## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>What’s New in SAS Optimization 8.2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Introduction</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Introduction to Optimization</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>Shared Concepts</td>
<td>19</td>
</tr>
<tr>
<td>5</td>
<td>The CLP Procedure</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>The OPTLP Procedure</td>
<td>107</td>
</tr>
<tr>
<td>7</td>
<td>The OPTMILP Procedure</td>
<td>161</td>
</tr>
<tr>
<td>8</td>
<td>The OPTQP Procedure</td>
<td>223</td>
</tr>
<tr>
<td>9</td>
<td>The MPS-Format Data Table</td>
<td>255</td>
</tr>
<tr>
<td>10</td>
<td>The OPTMODEL Procedure</td>
<td>271</td>
</tr>
<tr>
<td>11</td>
<td>The Constraint Programming Solver</td>
<td>441</td>
</tr>
<tr>
<td>12</td>
<td>The Linear Programming Solver</td>
<td>501</td>
</tr>
<tr>
<td>13</td>
<td>The Local Search Optimization Solver</td>
<td>567</td>
</tr>
<tr>
<td>14</td>
<td>The Mixed Integer Linear Programming Solver</td>
<td>607</td>
</tr>
<tr>
<td>15</td>
<td>The Network Solver</td>
<td>659</td>
</tr>
<tr>
<td>16</td>
<td>The Nonlinear Programming Solver</td>
<td>769</td>
</tr>
<tr>
<td>17</td>
<td>The Quadratic Programming Solver</td>
<td>825</td>
</tr>
<tr>
<td>18</td>
<td>The Decomposition Algorithm</td>
<td>849</td>
</tr>
</tbody>
</table>

### Subject Index

949

### Syntax Index

959
Chapter 1
What’s New in SAS Optimization 8.2

Contents

| Overview                               | 1 |
| Optimization Solver Updates           | 2 |
| Procedures                            | 2 |
| The Optimization Action Set           | 3 |

Overview

SAS Optimization 8.2 includes improvements to its procedures, optimization solvers, and CAS actions, including the following enhancements:

- The MILP, network, and NLP solvers and the decomposition algorithm (for LP and MILP) improve their performance.
- The LP and MILP solvers improve their numerical stability.
- The MILP solver adds a distributed branch-and-cut algorithm and the ability to report multiple solutions.
- PROC OPTNETWORK adds a path enumeration algorithm to find all paths between specified nodes. The network optimization action set adds the corresponding path action.
- The connected components algorithm in the network solver adds support for the thin internal graph format.
- The LSO (local search optimization) solver is added and is called by PROC OPTMODEL.
- PROC CLP (constraint logic programming) is added and runs on CAS. You can choose to run PROC CLP on a single machine or to distribute its execution across multiple nodes.
- The optimization action set adds two actions, loadMps and convertMps, that help convert an .mps file on the client to a CAS table that can be accepted as input by PROC OPTLP and PROC OPTMILP in SAS Optimization.
Chapter 1: What’s New in SAS Optimization 8.2

Optimization Solver Updates

Improving optimization solver performance is a continuing theme for SAS Optimization because users continue to discover larger and more challenging optimization problems. In SAS Optimization 8.2, the MILP (mixed integer linear programming) and network solvers deliver significant improvements in performance, shortening the time needed to reach optimality and enabling you to solve more complex problems. The decomposition algorithm, applicable to LP (linear programming) and MILP problems, delivers significant performance improvements in solving MILP problems. Improvements in the numerical stability of the LP and MILP solvers better equip them to handle problems with ill-conditioned constraint matrices.

The MILP solver expands its distributed computational features by adding a distributed branch-and-cut solution algorithm. In distributed mode, multiple nodes participate in solving a single MILP; this distributed computation can produce dramatic reductions in solution time, compared to solving on a single node.

The MILP solver also adds the ability to report more than one solution. You can use the MAXPOOLSOLS= option in the SOLVE statement for PROC OPTMODEL to specify the maximum number of solutions to return.

The network solver, a set of network optimization and analysis algorithms, is accessible via PROC OPTMODEL, PROC OPTNETWORK, and the network optimization action set. In SAS Optimization 8.2, this set adds a path enumeration algorithm, which finds all paths between the specified nodes in a network. You can choose to use one or all source (starting) nodes and one or all sink (ending) nodes. By default, the algorithm finds all paths between all pairs of nodes in the input network. This algorithm runs on CAS in distributed mode.

Also in the network solver, the connected components algorithm now supports the thin internal format for storing network structures. This is a simpler and less memory-intensive storage scheme than the default approach. The memory savings can be especially significant for large networks. A total of four algorithms in the network solver now support the thin format: connected components, minimum-cost network flow, minimum spanning tree, and path enumeration.

The LSO (local search optimization) solver, called by PROC OPTMODEL, is new in SAS Optimization 8.2. This heuristic solver executes multiple instances of global and local search algorithms in parallel on CAS, on a single node or distributed across multiple nodes. Local search optimization is typically used for nonlinear optimization problems in which the functions involved are likely be non-smooth, discontinuous, or computationally expensive to evaluate. The LSO solver can perform single- and multiobjective optimization.

Procedures

SAS Optimization 8.2 adds PROC CLP, which provides access to the same CLP solver as PROC OPTMODEL. In PROC CLP, the solution algorithm can run on a single node or distributed across multiple nodes, with two exceptions:

- The OBJECTIVE statement is not supported in distributed mode.
- If the EVALVARSEL= option is used and there are more specified strategies than available nodes, the strategies are executed consecutively on a single node.
Four other updates to SAS Optimization 8.2 procedures have already been mentioned:

- PROC OPTNETWORK adds a path enumeration algorithm.
- In PROC OPTNETWORK, the connected components algorithm adds support for the thin internal graph format.
- PROC OPTMODEL enables you to request that the MILP solver return multiple solutions.
- PROC OPTMODEL provides access to the new LSO solver.

The Optimization Action Set

Two new actions help users convert an .mps or .qps file on the client to a CAS table that can be accepted as input by PROC OPTLP, PROC OPTMILP, or PROC OPTQP (and their corresponding actions, solveLP, solveMilp, and solveQp) in SAS Optimization:

- **loadMps**: The user saves the file as a string by, for example, writing a few lines of Python code. The action accepts the string as input and produces a CAS table as output, adding an _ID_ column so that the original order of the rows in the input file can be retained even when the file is retrieved from distributed storage.

- **convertMps**: The user adds an _ID_ column and converts the file to a two-column CAS table that includes the _ID_ column and the .mps or .qps content in the remaining column. This can be done by, for example, converting to a .csv file and then using the loadTable action to load the .csv file into CAS. Then the convertMps action creates a CAS table that is suitable for input to the procedures and actions in SAS Optimization.

The loadMps action has the advantage of including file conversion, adding the _ID_ column, and loading to CAS in one step, but the process of saving a file as a string is usually slower than saving as a .csv file. Additionally, CAS actions impose a 2GB limit on the size of an input string. If the speed of the conversion process is critical, you should use the convertMps action, and if your .mps file is sufficiently large, you must use the convertMps action.
Chapter 2
Introduction

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to SAS Optimization</td>
<td>5</td>
</tr>
<tr>
<td>About This Book</td>
<td>5</td>
</tr>
<tr>
<td>Chapter Organization</td>
<td>6</td>
</tr>
<tr>
<td>Typographical Conventions</td>
<td>6</td>
</tr>
<tr>
<td>Options Used in Examples</td>
<td>7</td>
</tr>
<tr>
<td>Where to Turn for More Information</td>
<td>7</td>
</tr>
<tr>
<td>Online Documentation</td>
<td>7</td>
</tr>
<tr>
<td>SAS Technical Support Services</td>
<td>7</td>
</tr>
</tbody>
</table>

Introduction to SAS Optimization

SAS Optimization provides both general optimization procedures and network optimization procedures that take advantage of the distributed, cloud-based computational environment that SAS Viya provides. Optimization solvers are provided for linear (LP), mixed integer linear (MILP), quadratic (QP), nonlinear (NLP), network, and local search (LSO) problems. Constraint logic programming (CLP) is also supported. A decomposition algorithm can exploit several different types of structures in LP and MILP problems to decrease solution time. The network solver includes algorithms that solve optimization problems such as shortest path and minimum cost network flow, and also provide algorithms that analyze network structure, including connected components and cycle detection. The OPTMODEL procedure builds optimization models and calls the various solvers.

Optimization models are built in SAS, and solvers execute in CAS. The specialized OPTLP, OPTMILP, OPTQP, OPTNET, and CLP procedures accept input CAS tables that specify a problem (LP, MILP, QP, network, and CLP, respectively) and invoke the corresponding solver.

**NOTE:** When you license SAS Optimization, you also have access to SAS/OR software. For more information about SAS/OR procedures, see the SAS/OR documentation.

About This Book

This book assumes that you are familiar with Base SAS software and with the books *SAS Language Reference: Concepts* and *SAS Visual Data Management and Utility Procedures Guide*. It also assumes that you are familiar with basic SAS System concepts, such as using the DATA step to create SAS data sets and using Base SAS procedures (such as the PRINT and SORT procedures) to manipulate SAS data sets.
Chapter Organization

This book contains the following chapters:

- **Chapter 2**, this chapter, provides an overview of SAS Optimization and summarizes related information, products, and services.

- **Chapter 3** provides an overview of the operations research tools that SAS Optimization provides and shows which procedures you can use to tackle various programming problems and to build models.

- **Chapter 4** provides information about topics that are common to multiple procedures. Topics include how to use SAS Cloud Analytic Services (CAS) sessions, how to load a SAS data set onto a CAS server, and information about multithreaded parallel computing, memory limits, and numerical difficulties.

- The remaining chapters describe the SAS Optimization procedures and solvers, in addition to the MPS-format data table and the decomposition algorithm. These chapters are organized as follows:
  - The “Overview” section briefly describes the analysis provided by the procedure, solver, or algorithm.
  - The “Getting Started” section provides a quick introduction through a simple example.
  - The “Syntax” section describes the SAS statements and options that control the procedure, solver, or algorithm.
  - The “Details” section discusses methodology and other topics, such as ODS tables.
  - The “Examples” section contains examples.
  - The “References” section contains references for the methodology.

Typographical Conventions

This book uses several type styles for presenting information. The following list explains the meaning of the typographical conventions used in this book:

- **roman** is the standard type style used for most text.
- **UPPERCASE ROMAN** is used for SAS statements, options, and other SAS language elements when they appear in text. However, you can enter these elements in your own SAS programs in lowercase, uppercase, or a mixture of the two.
- **UPPERCASE BOLD** is used in the “Syntax” sections’ initial lists of SAS statements and options.
- **oblique** is used in the syntax definitions and in text to represent arguments for which you supply a value.
- **VariableName** is used for the names of variables and data sets when they appear in text.
- **bold** is used for matrices and vectors.
- **italic** is used for terms that are defined in text, for emphasis, and for references to publications.
- **monospace** is used for example code. In most cases, this book uses lowercase type for SAS code.
Options Used in Examples

The HTMLBLUE style is used to create the graphs and the HTML tables that appear in the online documentation. The PEARLJ style is used to create the PDF tables that appear in the documentation. A style template controls stylistic elements such as colors, fonts, and presentation attributes. You can specify a style template in an ODS destination statement as follows:

```sas
ods html style=HTMLBlue;
    . . .
ods html close;

ods pdf style=PearlJ;
    . . .
ods pdf close;
```

Most of the PDF tables are produced by using the following SAS System option:

```sas
options papersize=(6.5in 9in);
```

If you run the examples, you might get slightly different output. This is a function of the SAS System options that are used and the precision that your computer uses for floating-point calculations.

Where to Turn for More Information

Online Documentation

You can access the documentation by going to [http://support.sas.com/documentation](http://support.sas.com/documentation).

SAS Technical Support Services

The SAS Technical Support staff is available to respond to problems and answer technical questions regarding the use of procedures in this book. Go to [http://support.sas.com/techsup](http://support.sas.com/techsup) for more information.
Chapter 3
Introduction to Optimization

Contents

Overview ................................................................................................................. 9
Linear Programming Problems ............................................................................... 11
  The OPTLP Procedure ...................................................................................... 11
  The OPTMODEL Procedure ............................................................................. 11
Mixed Integer Linear Programming Problems ..................................................... 12
  The OPTMILP Procedure .................................................................................. 12
  The OPTMODEL Procedure ............................................................................. 12
Quadratic Programming Problems ....................................................................... 12
  The OPTQP Procedure ..................................................................................... 12
  The OPTMODEL Procedure ............................................................................. 13
Nonlinear Programming Problems ....................................................................... 13
  The OPTMODEL Procedure ............................................................................. 13
Model Building with PROC OPTMODEL ............................................................. 13
References ............................................................................................................ 17

Overview

Operations research tools are directed toward the solution of resource management and planning problems. Models in operations research are representations of the structure of a physical object or a conceptual or business process. Using the tools of operations research involves the following:

- defining a structural model of the system under investigation
- collecting the data for the model
- solving the model
- interpreting the results

SAS/OR software is a set of procedures for exploring models of distribution networks, production systems, resource allocation problems, and scheduling problems using the tools of operations research.
Chapter 3: Introduction to Optimization

The following list suggests some of the application areas in which optimization-based decision support systems have been used. In practice, models often contain elements of several applications listed here.

- **Product-mix problems** find the mix of products that generates the largest return when several products compete for limited resources.
- **Blending problems** find the mix of ingredients to be used in a product so that it meets minimum standards at minimum cost.
- **Time-staged problems** are models whose structure repeats as a function of time. Production and inventory models are classic examples of time-staged problems. In each period, production plus inventory minus current demand equals inventory carried to the next period.
- **Scheduling problems** assign people to times, places, or tasks so as to optimize people’s preferences or performance while satisfying the demands of the schedule.
- **Multiple objective problems** have multiple, possibly conflicting, objectives. Typically, the objectives are prioritized, and the problems are solved sequentially in a priority order.
- **Capital budgeting and project selection problems** ask for the project or set of projects that yield the greatest return.
- **Location problems** seek the set of locations that meets the distribution needs at minimum cost.
- **Cutting stock problems** find the partition of raw material that minimizes waste and fulfills demand.

The basic optimization problem is that of minimizing or maximizing an objective function subject to constraints imposed on the variables of that function. The objective function and constraints can be linear or nonlinear; the constraints can be bound constraints, equality or inequality constraints, or integer constraints. Traditionally, optimization problems are divided into various types depending on the sets of values that the variables are restricted to (real, integer, or binary, or a combination) and the nature of functional form of the constraints and objectives (linear, quadratic, or general nonlinear). An expression of an optimization problem in mathematical form is called a mathematical program.

When the complete description of a mathematical program is supplied to an appropriate algorithm (such as one of the solvers described in this book), the algorithm determines the optimal values for the decision variables so the objective is either maximized or minimized, the optimal values that are assigned to decision variables are on or between allowable bounds, and the constraints are obeyed. This process of solving mathematical programs is called mathematical programming, mathematical optimization, or just optimization.

When the constraints in an optimization problem are linear and the objective is either linear or quadratic, the optimization problem can be encapsulated in SAS data tables and then solved using the appropriate SAS/OR procedure: the OPTLP, OPTMILP, or OPTQP procedure.

Often optimization problems, and especially those with nonlinear elements, are formalized in an algebraic model that represents the problem. When formulated in its most abstract form, such an algebraic model is independent of problem data. A specific optimization problem instance (including the original problem) is then just an instantiation of the algebraic model with the specific data associated with that instance. An optimization modeling language (also called an algebraic modeling language) is a programming environment that has syntax, structures, and operations that enable you to express a mathematical program in a form that corresponds in a natural and transparent way to its algebraic model. The syntax, structures, and operations
also enable you to populate an algebraic model with a specific data instance and then solve the resulting optimization problem instance with an appropriate solver. The OPTMODEL procedure is such an algebraic modeling language in SAS/OR software and can be viewed as a single, unified environment to formulate and solve mathematical programming problems of many different types.

Whether mathematical programs are represented in SAS data tables or in an algebraic model in PROC OPTMODEL, they can be saved, easily changed, and solved again. The SAS/OR procedures also output SAS data tables that contain the solutions. These data tables can then be used to produce customized reports or as input to other SAS procedures. This structure enables you to use the tools of operations research and other SAS tools as building blocks to build decision support systems.

This chapter describes how to use SAS/OR software to solve a wide variety of optimization problems. It describes various types of optimization problems, indicates which SAS/OR procedures you can use, and shows how you provide data, run the procedure, and obtain optimal solutions. For additional examples that demonstrate the features of the OPTMODEL procedure, see SAS/OR User’s Guide: Mathematical Programming Examples.

The following sections broadly classify the SAS/OR procedures based on the types of mathematical programming problems that they can solve.

---

## Linear Programming Problems

### The OPTLP Procedure

The OPTLP procedure solves linear programming problems that are submitted in a SAS data table that uses a mathematical programming system (MPS) format.

The MPS file format is a format commonly used for describing linear programming (LP) and integer programming (IP) problems (Murtagh 1981; IBM 1988). MPS-format files are in text format and have specific conventions for the order in which the different pieces of the mathematical model are specified. The MPS-format SAS data table corresponds closely to the MPS file format and is used to describe linear programming problems for PROC OPTLP. For more information, see Chapter 9, “The MPS-Format Data Table.”

PROC OPTLP provides four solvers to solve general LPs: primal simplex, dual simplex, network simplex, and interior point. For more information about solving LPs with PROC OPTLP, see Chapter 6, “The OPTLP Procedure.” The OPTLP procedure also provides a decomposition algorithm for solving LPs. For more information about the decomposition algorithm, see Chapter 18, “The Decomposition Algorithm.”

### The OPTMODEL Procedure

The OPTMODEL procedure, a general purpose optimization modeling language, can also be used for concisely modeling linear programming problems. If an LP has special network structure, the structure is typically natural and evident in a well-formulated model of the problem in PROC OPTMODEL.
Within PROC OPTMODEL you can declare a model, pass it directly to various solvers, and review the solver result. You can also save an instance of a linear model in data table form for use by the OPTLP procedure. For more information, see Chapter 10, “The OPTMODEL Procedure.”

Mixed Integer Linear Programming Problems

The OPTMILP Procedure

The OPTMILP procedure solves general mixed integer linear programs (MILPs)—linear programs in which a subset of the decision variables are constrained to be integers. The OPTMILP procedure solves MILPs by using an LP-based branch-and-bound algorithm augmented by advanced techniques such as cutting planes and primal heuristics. For more information about the OPTMILP procedure, see Chapter 7, “The OPTMILP Procedure.” The OPTMILP procedure also provides a decomposition algorithm for solving MILPs. For more information about the decomposition algorithm, see Chapter 18, “The Decomposition Algorithm.”

The OPTMILP procedure requires a MILP to be specified by a SAS data table that adheres to the MPS format. For more information about the MPS-format data set, see Chapter 9, “The MPS-Format Data Table.”

The OPTMODEL Procedure

The OPTMODEL procedure, a general purpose optimization modeling language, can also be used for concisely modeling mixed integer linear programming problems. In fact, except for the declaration of some subset of variables to be integer or binary, modeling these problems is quite analogous to modeling LPs. Within OPTMODEL you can declare a model, pass it directly to various solvers, and review the solver result. You can also save an instance of a mixed integer linear model in data table form for use by PROC OPTMILP. For more details, see Chapter 10, “The OPTMODEL Procedure.”

Quadratic Programming Problems

The OPTQP Procedure

The OPTQP procedure uses the interior point algorithm to solve quadratic programs—problems that have a quadratic objective function and a collection of linear constraints, including general linear constraints along with lower or upper bounds (or both) on the decision variables.

You can specify the problem input data in one SAS data table that uses a quadratic programming system (QPS) format. For details about the QPS-format data specification, see Chapter 9, “The MPS-Format Data Table.” For more details about the OPTQP procedure, see Chapter 8, “The OPTQP Procedure.”
The OPTMODEL Procedure

The OPTMODEL procedure, a general-purpose optimization modeling language, can also be used for concisely modeling quadratic programming problems. Within OPTMODEL you can declare a model, pass it directly to various solvers, and review the solver result. You can also save an instance of a quadratic model in data table form for use by PROC OPTQP. For more details, see Chapter 10, “The OPTMODEL Procedure.”

Nonlinear Programming Problems

The OPTMODEL Procedure

The OPTMODEL procedure, a general-purpose optimization modeling language, can also be used for concisely modeling nonlinear programming (NLP) problems. Within OPTMODEL you can declare a nonlinear optimization model, pass it directly to various solvers, and review the solver result. For more details, see Chapter 10, “The OPTMODEL Procedure.”

You can solve many different types of NLP problems by using the nonlinear solver functionality of PROC OPTMODEL. The NLP solver implements two primal-dual based algorithms—interior point trust region line search and active-set trust region linear search—for finding a local minimum. For more information about the nonlinear programming solver, see Chapter 16, “The Nonlinear Programming Solver.”

Model Building with PROC OPTMODEL

Model generation and maintenance are often difficult and expensive aspects of applying mathematical programming techniques. The richly expressive syntax and features of PROC OPTMODEL, in addition to the flexible data input and output capabilities, simplify this task considerably. Although PROC OPTMODEL offers almost unlimited latitude in how a particular optimization problem is formulated, the most effective use of OPTMODEL is achieved when the model is abstracted away from the data. This aspect makes PROC OPTMODEL somewhat unusual among SAS procedures and is important enough to illustrate with a simple example.

A small product-mix problem serves as a starting point for a discussion of two different ways of modeling with PROC OPTMODEL.

A candy manufacturer makes two products: chocolate and toffee. What combination of chocolate and toffee should be produced in a day in order to maximize the company’s profit? Chocolate contributes $0.25 per pound to profit, and toffee contributes $0.75 per pound. The decision variables are chocolate and toffee.

Four processes are used to manufacture the candy:

1. Process 1 combines and cooks the basic ingredients for both chocolate and toffee.
2. Process 2 adds colors and flavors to the toffee, then cools and shapes the confection.
3. Process 3 chops and mixes nuts and raisins, adds them to the chocolate, and then cools and cuts the bars.

4. Process 4 is packaging: chocolate is placed in individual paper shells; toffee is wrapped in cellophane packages.

During the day, there are 7.5 hours (27,000 seconds) available for each process.

Firm time standards have been established for each process. For Process 1, mixing and cooking take 15 seconds for each pound of chocolate, and 40 seconds for each pound of toffee. Process 2 takes 56.25 seconds per pound of toffee. For Process 3, each pound of chocolate requires 18.75 seconds of processing. In packaging, a pound of chocolate can be wrapped in 12 seconds, whereas a pound of toffee requires 50 seconds. These data are summarized as follows:

<table>
<thead>
<tr>
<th>Process</th>
<th>Time (sec)</th>
<th>chocolate (sec)</th>
<th>toffee (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Cooking</td>
<td>27,000</td>
<td>15</td>
<td>40</td>
</tr>
<tr>
<td>2 Color/Flavor</td>
<td>27,000</td>
<td></td>
<td>56.25</td>
</tr>
<tr>
<td>3 Condiments</td>
<td>27,000</td>
<td>18.75</td>
<td></td>
</tr>
<tr>
<td>4 Packaging</td>
<td>27,000</td>
<td>12</td>
<td>50</td>
</tr>
</tbody>
</table>

The objective is to maximize the company’s total profit, which is represented as

Maximize: \(0.25(\text{chocolate}) + 0.75(\text{toffee})\)

The production of the candy is limited by the time available for each process. The limits placed on production by Process 1 are expressed by the following inequality:

Process 1: \(15(\text{chocolate}) + 40(\text{toffee}) \leq 27,000\)

Process 1 can handle any combination of chocolate and toffee that satisfies this inequality.

The limits on production by other processes generate constraints described by the following inequalities:

Process 2: \(56.25(\text{toffee}) \leq 27,000\)

Process 3: \(18.75(\text{chocolate}) \leq 27,000\)

Process 4: \(12(\text{chocolate}) + 50(\text{toffee}) \leq 27,000\)

This linear program illustrates an example of a product mix problem. The mix of products that maximizes the objective without violating the constraints is the solution.

First, the following statements demonstrate a way of representing the optimization model in PROC OPTMODEL that is almost a verbatim translation of the mathematical model:

```plaintext
proc optmodel;
   /* declare variables */
   var choco >= 0, toffee >= 0;

   /* maximize objective function (profit) */
   maximize profit = 0.25*choco + 0.75*toffee;
```
/* subject to constraints */
con process1: 15*choco + 40*toffee <= 27000;
con process2: 56.25*toffee <= 27000;
con process3: 18.75*choco <= 27000;
con process4: 12*choco + 50*toffee <= 27000;

/* solve LP using primal simplex algorithm */
solve with lp / algorithm = primal_spx;

/* display solution */
print choco toffee;
quit;

The optimal objective value and the optimal solution are displayed in Figure 3.1:

**Figure 3.1 Solution Summary**

**The OPTMODEL Procedure**

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>LP</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Primal Simplex</td>
</tr>
<tr>
<td>Objective Function</td>
<td>profit</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Optimal</td>
</tr>
<tr>
<td>Objective Value</td>
<td>475</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Dual Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Iterations</td>
<td>4</td>
</tr>
<tr>
<td>Presolve Time</td>
<td>0.00</td>
</tr>
<tr>
<td>Solution Time</td>
<td>0.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>choco toffee</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
</tr>
<tr>
<td>300</td>
</tr>
</tbody>
</table>

You can observe from the preceding example that PROC OPTMODEL provides an easy and very direct way of modeling and solving mathematical programming models. Although this way of modeling, where the data are intertwined heavily with model elements, is correct, has significant practical limitations. The model is not easy to explain, it is hard to generalize, and clearly this approach does not scale to large problems of the same type. To overcome these issues, you need to separate the data from the essential algebraic structure of the model. Along those lines, you can make the reasonable assumption that you have the following two data sets (one for the products and one for processes that capture the parameters and data elements of this product mix problem):

data Products;
   length Name $10.;
   input Name $ Profit;
datalines;
Chocolate 0.25
Chapter 3: Introduction to Optimization

Toffee 0.75
;

data Processes;
  length Name $15.;
  input Name $ Available_time Chocolate Toffee;
datalines;
Cooking 27000 15 40
Color/Flavor 27000 0 56.25
Condiments 27000 18.75 0
Packaging 27000 12 50
;

The following alternative model in PROC OPTMODEL can solve the same problem by taking these data sets as input:

proc optmodel;
  /* declare sets and data indexed by sets */
  set <string> Products;
  set <string> Processes;
  num Profit{Products};
  num AvailableTime{Processes};
  num RequiredTime{Products,Processes};
  /* declare the variable */
  var Amount{Products};
  /* maximize objective function (profit) */
  maximize TotalProfit = sum{p in Products} Profit[p]*Amount[p];
  /* subject to constraints */
  con Availability{r in Processes}:
    sum{p in Products} RequiredTime[p,r]*Amount[p] <= AvailableTime[r];
  /* abstract algebraic model that captures the structure of the */
  /* optimization problem has been defined without referring */
  /* to a single data constant */
  /* populate model by reading in the specific data instance */
  read data Products into Products=[name] Profit;
  read data Processes into Processes=[name] AvailableTime=Available_time
    {p in Products} <RequiredTime[p,name]= col(p)>;
  /* solve LP using primal simplex algorithm */
  solve with lp / algorithm = primal_spx;
  /* display solution */
  print Amount;
quit;

The details of the syntax and elements of the PROC OPTMODEL language are discussed in Chapter 10, “The OPTMODEL Procedure.” The key observation here is that the preceding version of the PROC OPTMODEL statements capture the essence of the optimization model concisely, but completely, and the model can be explained, modified, and maintained easily. It also achieves total separation of the data from the model in that the same PROC OPTMODEL statements can be applied to any other specific problem of this type (and of any size) by simply changing the data sets appropriately and rerunning the same PROC OPTMODEL statements. Also, because of PROC OPTMODEL’s ability to read data very flexibly and from any number of data sets, the problem data can be in its most natural form, making the model easier to explain and understand.


Chapter 4
Shared Concepts

Contents

<table>
<thead>
<tr>
<th>Introduction to Shared Concepts</th>
<th>19</th>
</tr>
</thead>
<tbody>
<tr>
<td>Using CAS Sessions and CAS Engine Librefs</td>
<td>19</td>
</tr>
<tr>
<td>Loading a SAS Data Set onto a CAS Server</td>
<td>20</td>
</tr>
<tr>
<td>Multithreaded Parallel Computing</td>
<td>21</td>
</tr>
<tr>
<td>Memory Limit</td>
<td>21</td>
</tr>
<tr>
<td>Numerical Difficulties</td>
<td>22</td>
</tr>
<tr>
<td>References</td>
<td>23</td>
</tr>
</tbody>
</table>

Introduction to Shared Concepts

SAS/OR procedures run on SAS Viya. One component of SAS Viya is SAS Cloud Analytic Services (CAS), which is the analytic server and associated cloud services. The following subsections describe how to set up and use CAS sessions. The sections after that then provide details on the concepts that are common to most of the procedures.

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```sas
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:
cas mysess;
libname mycas cas sessref=mysess;

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

cas mysess terminate;

For more information about the CAS statement and the LIBNAME statement, see SAS Cloud Analytic Services: Language Reference. For general information about CAS and CAS sessions, see SAS Cloud Analytic Services: Fundamentals.

---

### Loading a SAS Data Set onto a CAS Server

Procedures in this book require the input data to reside on a CAS server. To work with a SAS data set, you must first load the data set onto the CAS server. Data loaded on the CAS server are called *data tables*. This section lists three methods of loading a SAS data set onto a CAS server. In this section, mycas is the name of the caslib that is connected to the mysess CAS session.

- You can use a single DATA step to create a data table on the CAS server as follows:

  ```sas
  data mycas.Sample;
  input y x @@;
  datalines;
  .46 1 .47 2 .57 3 .61 4 .62 5 .68 6 .69 7
  ;
  ```

  Note that DATA step operations might not work as intended when you perform them on the CAS server instead of the SAS client.

- You can create a SAS data set first, and when it contains exactly what you want, you can use another DATA step to load it onto the CAS server as follows:

  ```sas
  data Sample;
  input y x @@;
  datalines;
  .46 1 .47 2 .57 3 .61 4 .62 5 .68 6 .69 7 .78 8
  ;
  data mycas.Sample;
  set Sample;
  run;
  ```

- You can use the CASUTIL procedure as follows:
The CASUTIL procedure can load data onto a CAS server more efficiently than the DATA step. For more information about the CASUTIL procedure, see SAS Cloud Analytic Services: Language Reference.

The mycas caslib stores the Sample data table, which can be distributed across many machine nodes. You must use a caslib reference in procedures in this book to enable the SAS client machine to communicate with the CAS session. For example, the following OPTLP procedure statements use a data table that resides in the mycas caslib:

```sas
proc optlp data = mycas.Sample;
    ...statements...
run;
```

You can delete your data table by using the DELETE procedure as follows:

```sas
proc delete data = mycas.Sample;
    run;
```

The Sample data table is accessible only in the mysess session. When you terminate the mysess session, the Sample data table is no longer accessible from the CAS server. If you want your Sample data table to be available to other CAS sessions, then you must promote your data table. For more information about data tables, see SAS Cloud Analytic Services: User’s Guide.

---

**Multithreaded Parallel Computing**

Although the speed of a single-core processor has increased considerably over the decades, further gains in computing power are possible through the use of multiple cores or processors. This practice is called parallel computing, in which certain computations are partitioned into independent smaller subcomputations. Each subcomputation is then processed on separate cores or processors simultaneously. Consumer-grade PCs and servers are often equipped with multicore processors; multiprocessor configurations are becoming relatively common and inexpensive. As a result, parallel computing is becoming increasingly important. One type of parallel computing is multithreaded computing, in which several threads use the processors of a single server to work concurrently on subtasks. These threads share the random access memory (RAM) of that server. In another type of parallel computing, distributed computing, computation is parallelized over several processors (possibly multithreaded), each of which owns an independent memory allocation.

---

**Memory Limit**

The system option MEMSIZE sets a limit on the amount of memory that the SAS System uses. If you do not specify a value for this option, then the SAS System sets a default memory limit. Your operating environment determines the actual size of the default memory limit set by the SAS System, which is sufficient for many
applications. However, the solution of many realistic optimization problems can require more memory than the default. It is therefore recommended that the memory limit be increased above the default when you are solving optimization problems. This reduces the chance of a procedure failing because of an out-of-memory error.

**NOTE:** The MEMSIZE system option is not available in some operating environments. See the documentation for your operating environment for more information.

You can specify -MEMSIZE 0 to indicate that all available memory can be used, but use this setting with caution. In most operating environments, it is better to specify an adequate amount of memory than to specify -MEMSIZE 0. For example, if you are running PROC OPTLP to solve LP problems with only a few hundred thousand variables and constraints, -MEMSIZE 500M might be sufficient to enable the procedure to run without an out-of-memory error. When a problem has millions of variables, -MEMSIZE 2G or higher might be needed. These are rules of thumb; problems with atypical structure, density, or other characteristics can increase the optimizer’s memory requirements.

No matter how much memory is installed, 32-bit Windows operating systems permit the SAS System to use at most 4 gigabytes of memory. This memory limit might be lower, depending on which version of Windows you are running. The limit is enforced by the Windows operating system, not the SAS System.

You can specify the MEMSIZE option at system invocation, on the SAS command line, or in a configuration file. The syntax is described in the *SAS Companion* book for your operating environment.

To report a procedure’s memory consumption, you can use the FULLSTIMER option. The syntax is described in the *SAS Companion* book for your operating environment.

---

**Numerical Difficulties**

Extremely large or extremely small numerical values might cause computational difficulties (singularities, stalled solution progress, false infeasibilities, and so on) for optimization solvers, but the occurrence of such difficulties is hard to predict. For this reason, solvers issue a data error message when they detect model data that exceed a specific threshold number. The value of the threshold number depends on your operating environment and is printed in the log as part of the data error message.

The following conditions produce a data error:

- The absolute value of an objective coefficient, constraint coefficient, or range (difference between the upper and lower bounds on a constraint) is greater than the threshold number.

- A variable’s lower bound, a ≤ or = constraint’s right-hand side, or a range constraint’s lower bound is greater than the threshold number.

- A variable’s upper bound, a ≥ or = constraint’s right-hand side, or a range constraint’s upper bound is smaller than the negative threshold number.

If a variable’s upper bound is greater than 1E20, then solvers treats the bound as ∞. Similarly, if a variable’s lower bound is less than −1E20, then LP solver treats the bound as −∞.

If a solver fails or experiences numerical difficulties when solving a problem, try one of the following remedies:
• Improve the input data: Rescale very large and very small numbers in constraints, objectives, right-hand sides, and variable bounds. It is recommended that the magnitudes of the largest and smallest constraint coefficients not exceed 1E6.

• Specify different algorithms or options (or both): For example, to solve a linear program, you can choose from the primal simplex, dual simplex, interior point, and network simplex algorithms. Using available options, you can tighten or relax feasibility or optimality tolerances.

References

# Chapter 5
The CLP Procedure

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview: CLP Procedure</td>
<td>26</td>
</tr>
<tr>
<td>The Constraint Satisfaction Problem</td>
<td>26</td>
</tr>
<tr>
<td>Techniques for Solving CSPs</td>
<td>27</td>
</tr>
<tr>
<td>The CLP Procedure</td>
<td>28</td>
</tr>
<tr>
<td>Getting Started: CLP Procedure</td>
<td>29</td>
</tr>
<tr>
<td>Send More Money</td>
<td>29</td>
</tr>
<tr>
<td>Eight Queens</td>
<td>31</td>
</tr>
<tr>
<td>Syntax: CLP Procedure</td>
<td>33</td>
</tr>
<tr>
<td>Functional Summary</td>
<td>33</td>
</tr>
<tr>
<td>PROC CLP Statement</td>
<td>35</td>
</tr>
<tr>
<td>ALLDIFF Statement</td>
<td>38</td>
</tr>
<tr>
<td>ARRAY Statement</td>
<td>38</td>
</tr>
<tr>
<td>ELEMENT Statement</td>
<td>39</td>
</tr>
<tr>
<td>FOREACH Statement</td>
<td>40</td>
</tr>
<tr>
<td>GCC Statement</td>
<td>40</td>
</tr>
<tr>
<td>LEXICO Statement</td>
<td>41</td>
</tr>
<tr>
<td>LINCON Statement</td>
<td>42</td>
</tr>
<tr>
<td>OBJ Statement</td>
<td>43</td>
</tr>
<tr>
<td>PACK Statement</td>
<td>44</td>
</tr>
<tr>
<td>REIFY Statement</td>
<td>45</td>
</tr>
<tr>
<td>VARIABLE Statement</td>
<td>46</td>
</tr>
<tr>
<td>Details: CLP Procedure</td>
<td>46</td>
</tr>
<tr>
<td>Type of Solvers</td>
<td>46</td>
</tr>
<tr>
<td>Constraint Data Table</td>
<td>47</td>
</tr>
<tr>
<td>Solution Data Table</td>
<td>49</td>
</tr>
<tr>
<td>Parallel Processing</td>
<td>49</td>
</tr>
<tr>
<td>ODS Tables</td>
<td>50</td>
</tr>
<tr>
<td>Macro Variable <em>ORCLP</em></td>
<td>52</td>
</tr>
<tr>
<td>Macro Variable <em>ORCLPEVS</em></td>
<td>53</td>
</tr>
<tr>
<td>Examples: CLP Procedure</td>
<td>53</td>
</tr>
<tr>
<td>Example 5.1: Logic-Based Puzzles</td>
<td>54</td>
</tr>
<tr>
<td>Example 5.2: Car Painting Problem</td>
<td>64</td>
</tr>
<tr>
<td>Example 5.3: Scene Allocation Problem</td>
<td>67</td>
</tr>
<tr>
<td>Example 5.4: Car Sequencing Problem</td>
<td>71</td>
</tr>
<tr>
<td>Example 5.5: Scheduling a Major Basketball Conference</td>
<td>78</td>
</tr>
<tr>
<td>Example 5.6: Balanced Incomplete Block Design</td>
<td>89</td>
</tr>
</tbody>
</table>
Overview: CLP Procedure

The CLP procedure is a finite-domain constraint programming solver for constraint satisfaction problems (CSPs) with linear, logical, and global constraints. In addition to having an expressive syntax for representing CSPs, the CLP procedure features powerful built-in consistency routines and constraint propagation algorithms, a choice of nondeterministic search strategies, and controls for guiding the search mechanism that enable you to solve a diverse array of combinatorial problems.

You can also solve standard constraint satisfaction problems via the constraint programming solver in PROC OPTMODEL. For more information, see the constraint programming solver chapter in *SAS Optimization: Mathematical Optimization Procedures*.

The Constraint Satisfaction Problem

Many important problems in areas such as artificial intelligence (AI) and operations research (OR) can be formulated as constraint satisfaction problems. A CSP is defined by a finite set of variables that take values from finite domains and by a finite set of constraints that restrict the values that the variables can simultaneously take.

More formally, a CSP can be defined as a triple \( \langle X, D, C \rangle \):

- \( X = \{x_1, \ldots, x_n\} \) is a finite set of variables.
- \( D = \{D_1, \ldots, D_n\} \) is a finite set of domains, where \( D_i \) is a finite set of possible values that the variable \( x_i \) can take. \( D_i \) is known as the domain of variable \( x_i \).
- \( C = \{c_1, \ldots, c_m\} \) is a finite set of constraints that restrict the values that the variables can simultaneously take.

The domains need not represent consecutive integers. For example, the domain of a variable could be the set of all even numbers in the interval \([0, 100]\). A domain does not even need to be totally numeric. In fact, in a scheduling problem with resources, the values are typically multidimensional. For example, an activity can be considered as a variable, and each element of the domain would be an \( n \)-tuple that represents a start time for the activity and one or more resources that must be assigned to the activity that corresponds to the start time.

A solution to a CSP is an assignment of values to the variables in order to satisfy all the constraints. The problem amounts to finding one or more solutions, or possibly determining that a solution does not exist.

The CLP procedure can be used to find one or more (and in some instances, all) solutions to a CSP with linear, logical, and global constraints. The numeric components of all variable domains are assumed to be integers.
Techniques for Solving CSPs

Several techniques for solving CSPs are available. Kumar (1992) and Tsang (1993) present a good overview of these techniques. It should be noted that the satisfiability problem (SAT) (Garey and Johnson 1979) can be regarded as a CSP. Consequently, most problems in this class are nondeterministic polynomial-time complete (NP-complete) problems, and a backtracking search mechanism is an important technique for solving them (Floyd 1967).

One of the most popular tree search mechanisms is chronological backtracking. However, a chronological backtracking approach is not very efficient because conflicts are detected late; that is, it is oriented toward recovering from failures rather than avoiding them to begin with. The search space is reduced only after detection of a failure, and the performance of this technique is drastically reduced with increasing problem size. Another drawback of using chronological backtracking, for the same reason, is encountering repeated failures, sometimes called “thrashing.” The presence of late detection and “thrashing” has led researchers to develop consistency techniques that can achieve superior pruning of the search tree. This strategy uses constraints actively, rather than passively.

Constraint Propagation

A more efficient technique than backtracking is constraint propagation, which uses consistency techniques to effectively prune the domains of variables. Consistency techniques are based on the idea of a priori pruning, which uses the constraint to reduce the domains of the variables. Consistency techniques are also known as relaxation algorithms (Tsang 1993), and the process is also called problem reduction, domain filtering, or pruning.

One of the earliest applications of consistency techniques was in the AI field in solving the scene labeling problem, which required recognizing objects in three-dimensional space by interpreting two-dimensional line drawings of the object. The Waltz filtering algorithm (Waltz 1975) analyzes line drawings by systematically labeling the edges and junctions while maintaining consistency between the labels.

An effective consistency technique for handling resource capacity constraints is edge finding (Applegate and Cook 1991). Edge-finding techniques reason about the processing order of a set of activities that require a given resource or set of resources. Some of the earliest work related to edge finding can be attributed to Carlier and Pinson (1989), who successfully solved MT10, a well-known 10×10 job shop problem that had remain unsolved for over 20 years (Muth and Thompson 1963).

Constraint propagation is characterized by the extent of propagation (also referred to as the level of consistency) and whether domain propagation or interval propagation is the domain pruning scheme that is followed. In practice, interval propagation is preferred over domain propagation because of its lower computational costs. This mechanism is discussed in detail in Van Hentenryck (1989). However, constraint propagation is not a complete solution technique and needs to be complemented by a search technique in order to ensure success (Kumar 1992).

Finite-Domain Constraint Programming

Finite-domain constraint programming is an effective and complete solution technique that embeds incomplete constraint propagation techniques into a nondeterministic backtracking search mechanism, implemented as follows. Whenever a node is visited, constraint propagation is carried out to attain a desired level of consistency. If the domain of each variable reduces to a singleton set, the node represents a solution to the CSP. If the domain of a variable becomes empty, the node is pruned. Otherwise a variable is selected, its
domain is distributed, and a new set of CSPs is generated, each of which is a child node of the current node. Several factors play a role in determining the outcome of this mechanism, such as the extent of propagation (or level of consistency enforced), the variable selection strategy, and the variable assignment or domain distribution strategy.

For example, the lack of any propagation reduces this technique to a simple generate-and-test, whereas performing consistency on variables already selected reduces this to chronological backtracking, one of the systematic search techniques. These are also known as look-back schemas, because they share the disadvantage of late conflict detection. Look-ahead schemas, on the other hand, work to prevent future conflicts. Some popular examples of look-ahead strategies, in increasing degree of consistency level, are forward checking (FC), partial look ahead (PLA), and full look ahead (LA) (Kumar 1992). Forward checking enforces consistency between the current variable and future variables; PLA and LA extend this even further to pairs of not yet instantiated variables.

Two important consequences of this technique are that inconsistencies are discovered early and that the current set of alternatives that are coherent with the existing partial solution is dynamically maintained. These consequences are powerful enough to prune large parts of the search tree, thereby reducing the “combinatorial explosion” of the search process. However, although constraint propagation at each node results in fewer nodes in the search tree, the processing at each node is more expensive. The ideal scenario is to strike a balance between the extent of propagation and the subsequent computation cost.

Variable selection is another strategy that can affect the solution process. The order in which variables are chosen for instantiation can have a substantial impact on the complexity of the backtrack search. Several heuristics have been developed and analyzed for selecting variable ordering. One of the more common ones is a dynamic heuristic based on the fail first principle (Haralick and Elliott 1980), which selects the variable whose domain has minimal size. Subsequent analysis of this heuristic by several researchers has validated this technique as providing substantial improvement for a significant class of problems. Another popular technique is to instantiate the most constrained variable first. Both these strategies are based on the principle of selecting the variable most likely to fail and to detect such failures as early as possible.

The domain distribution strategy for a selected variable is yet another area that can influence the performance of a backtracking search. However, good value-ordering heuristics are expected to be very problem-specific (Kumar 1992).

The CLP Procedure

The CLP procedure is a finite-domain constraint programming solver for CSPs. In the context of the CLP procedure, CSPs can be classified into the following two types:

- A standard CSP is characterized by integer variables, linear constraints, array-type constraints, global constraints, and reified constraints. In other words, $X$ is a finite set of integer variables, and $C$ can contain linear, array, global, or logical constraints.

- A scheduling CSP is characterized by activities, temporal constraints, and resource requirement constraints. In other words, $X$ is a finite set of activities, and $C$ is a set of temporal constraints and resource requirement constraints. This capability requires a SAS/OR license to run. For more information, see SAS/OR User’s Guide: Constraint Programming.
In order to solve a standard CSP, the CLP procedure requires a Solution data table for storing the solution(s) to the CSP; to specify this data table, you use the OUT= option in the PROC CLP statement. For more information about the format and layout of the Solution data table, see the section “Solution Data Table” on page 49. In addition, the procedure expects only ALLDIFF, ARRAY, ELEMENT, FOREACH, GCC, LEXICO, LINCON, PACK, REIFY, and VARIABLE statements. You can also specify a Constraint data table by using the CONDATA= option in the PROC CLP statement instead of, or in combination with, VARIABLE and LINCON statements.

**Consistency Techniques**

The CLP procedure features a full look-ahead algorithm for standard CSPs that follows a strategy of maintaining a version of generalized arc consistency that is based on the AC-3 consistency routine (Mackworth 1977). This strategy maintains consistency between the selected variables and the unassigned variables and also maintains consistency between unassigned variables.

**Selection Strategy**

A search algorithm for CSPs searches systematically through the possible assignments of values to variables. The order in which a variable is selected can be based on a static ordering, which is determined before the search begins, or on a dynamic ordering, in which the choice of the next variable depends on the current state of the search. The VARSELECT= option in the PROC CLP statement defines the variable selection strategy for a standard CSP. The default strategy is the dynamic MINR strategy, which selects the variable with the smallest range.

**Assignment Strategy**

After a variable has been selected, the assignment strategy dictates the value that is assigned to it. The assignment strategy is specified with the VARASSIGN= option in the PROC CLP statement. The default assignment strategy selects the minimum value from the domain of the selected variable.

---

**Getting Started: CLP Procedure**

The following examples illustrate the use of the CLP procedure in the formulation and solution of two well-known logical puzzles in the constraint programming community.

---

**Send More Money**

The Send More Money problem consists of finding unique digits for the letters D, E, M, N, O, R, S, and Y such that S and M are different from zero (no leading zeros) and the following equation is satisfied:
Chapter 5: The CLP Procedure

You can use the CLP procedure to formulate this problem as a CSP by representing each of the letters in the expression with an integer variable. The domain of each variable is the set of digits 0 through 9. The VARIABLE statement identifies the variables in the problem. The DOM= option defines the default domain for all the variables to be \([0,9]\). The OUT= option defines the Solution data table. The LINCON statement defines the linear constraint \(SEND + MORE = MONEY\), and the restrictions that \(S\) and \(M\) cannot take the value zero. (Alternatively, you can simply specify the domain for \(S\) and \(M\) as \([1,9]\) in the VARIABLE statement.) Finally, the ALLDIFF statement is specified to enforce the condition that the assignment of digits should be unique. The complete representation, using the CLP procedure, is as follows:

```plaintext
proc clp dom=[0,9] /* Define the default domain */
   out=mycas.out; /* Name the Solution data table */
var SEND MORE MONEY; /* Declare the variables */
lincon /* Linear constraints */
   /* SEND + MORE = MONEY */
   1000*S + 100*E + 10*N + D + 1000*M + 100*O + 10*R + E
   =
   10000*M + 1000*O + 100*N + 10*E + Y,
   S<>0, /* No leading zeros */
   M<>0;
   alldiff(); /* All variables have pairwise distinct values*/
run;
```

The Solution data table that is produced by the CLP procedure is shown in Output 5.1.

<table>
<thead>
<tr>
<th>SEND</th>
<th>M O N E Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>9567</td>
<td>1082</td>
</tr>
</tbody>
</table>

The unique solution to the problem that the CLP procedure finds is as follows:

\[
\begin{align*}
9 & 5 & 6 & 7 \\
+ & 1 & 0 & 8 & 5 \\
\hline
1 & 0 & 6 & 5 & 2
\end{align*}
\]
Eight Queens

The Eight Queens problem is a special instance of the \(N\)-Queens problem, where the objective is to position \(N\) queens on an \(N \times N\) chessboard such that no two queens attack each other. The CLP procedure provides an expressive constraint for variable arrays that can be used for solving this problem very efficiently.

You can model this problem by using a variable array \(A\) of dimension \(N\), where \(A[i]\) is the row number of the queen in column \(i\). Since no two queens can be in the same row, it follows that all the \(A[i]\)'s must be pairwise distinct.

In order to ensure that no two queens can be on the same diagonal, the following should be true for all \(i\) and \(j\):

\[
\]

and

\[
\]

In other words,

\[
\]

and

\[
\]

Hence, the \((A[i] + i)\)'s are pairwise distinct, and the \((A[i] - i)\)'s are pairwise distinct.

These two conditions, in addition to the one requiring that the \(A[i]\)'s be pairwise distinct, can be formulated using the FOREACH statement.

One possible such CLP formulation is presented as follows:

```
proc clp out=mycas.out
  varselect=fifo; /* Variable Selection Strategy */
  array A[8] (A1-A8); /* Define the array A */
  var (A1-A8)=[1,8]; /* Define each of the variables in the array */
  /* Initialize domains */
  foreach(A, DIFF, 0); /* A[i]'s are pairwise distinct */
  foreach(A, DIFF, -1); /* A[i] - i's are pairwise distinct */
  foreach(A, DIFF, 1); /* A[i] + i's are pairwise distinct */
run;
```

The ARRAY statement is required when you are using a FOREACH statement, and it defines the array \(A\) in terms of the eight variables \(A1-A8\). The domain of each of these variables is explicitly specified in the VARIABLE statement to be the digits 1 through 8 since they represent the row number on an 8×8 board. FOREACH(A, DIFF, 0) represents the constraint that the \(A[i]\)'s are different. FOREACH(A, DIFF, -1) represents the constraint that the \((A[i] - i)\)'s are different, and FOREACH(A, DIFF, 1) represents the constraint that the \((A[i] + i)\)'s are different. The VARSELECT= option specifies the variable selection strategy to be first-in-first-out, the order in which the variables are encountered by the CLP procedure.

The following statements display the Solution data table shown in Output 5.2:
proc print data=mycas.out noobs label;
  label A1=a A2=b A3=c A4=d
       A5=e A6=f A7=g A8=h;
run;

Output 5.2  A Solution to the Eight Queens Problem

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
<th>g</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>8</td>
<td>6</td>
<td>3</td>
<td>7</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

The corresponding solution to the Eight Queens problem is displayed in Figure 5.1.

Figure 5.1  A Solution to the Eight Queens Problem
Syntax: CLP Procedure

The following statements are used in PROC CLP:

```
PROC CLP options;
  ALLDIFF alldiff constraints;
  ARRAY array specifications;
  ELEMENT element constraints;
  FOREACH foreach constraints;
  GCC global cardinality constraints;
  LEXICO lexicographic ordering constraints;
  LINCON linear constraints;
  OBJ objective function options;
  PACK bin packing constraints;
  REIFY reified constraints;
  VARIABLE variable specifications;
```

Functional Summary

The statements and options available in PROC CLP are summarized by purpose in Table 5.1.

<table>
<thead>
<tr>
<th>Description</th>
<th>Statement</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment Strategy Options</td>
<td>PROC CLP</td>
<td>VARASSIGN=</td>
</tr>
<tr>
<td>Specifies the variable assignment strategy</td>
<td>PROC CLP V ARASSIGN=</td>
<td></td>
</tr>
<tr>
<td>Data Table Options</td>
<td>PROC CLP</td>
<td>CONDATA=</td>
</tr>
<tr>
<td>Specifies the Constraint input data table</td>
<td>PROC CLP</td>
<td>CONDATA=</td>
</tr>
<tr>
<td>Specifies the Solution output data table</td>
<td>PROC CLP</td>
<td>OUT=</td>
</tr>
<tr>
<td>Domain Options</td>
<td>PROC CLP</td>
<td>DOMAIN=</td>
</tr>
<tr>
<td>Specifies the global domain of all variables</td>
<td>PROC CLP V ARABLE</td>
<td></td>
</tr>
<tr>
<td>Specifies the domain for selected variables</td>
<td>PROC CLP V ARABLE</td>
<td></td>
</tr>
<tr>
<td>General Options</td>
<td>PROC CLP</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Specifies the upper bound on time (seconds)</td>
<td>PROC CLP</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Suppresses preprocessing</td>
<td>PROC CLP</td>
<td>NOPREPROCESS</td>
</tr>
<tr>
<td>Permits preprocessing</td>
<td>PROC CLP</td>
<td>PREPROCESS</td>
</tr>
<tr>
<td>Specifies the units of MAXTIME</td>
<td>PROC CLP</td>
<td>TIMETYPE=</td>
</tr>
<tr>
<td>Implicitly defines Constraint data table variables</td>
<td>PROC CLP</td>
<td>USECONDATAVARS=</td>
</tr>
<tr>
<td>Objective Function Options</td>
<td>OBJ</td>
<td>LB=</td>
</tr>
<tr>
<td>Specifies the lower bound for the objective</td>
<td>OBJ</td>
<td>LB=</td>
</tr>
<tr>
<td>Specifies the tolerance for the search</td>
<td>OBJ</td>
<td>TOL=</td>
</tr>
<tr>
<td>Specifies the upper bound for the objective</td>
<td>OBJ</td>
<td>UB=</td>
</tr>
</tbody>
</table>
### Table 5.1 continued

<table>
<thead>
<tr>
<th>Description</th>
<th>Statement</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Output Control Options</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finds all possible solutions</td>
<td>PROC CLP</td>
<td>FINDALLSOLNS</td>
</tr>
<tr>
<td>Specifies the number of solution attempts</td>
<td>PROC CLP</td>
<td>MAXSOLNS=</td>
</tr>
<tr>
<td>Indicates progress in log</td>
<td>PROC CLP</td>
<td>SHOWPROGRESS</td>
</tr>
<tr>
<td><strong>Selection Strategy Options</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the variable selection strategy</td>
<td>PROC CLP</td>
<td>VARSELECT=</td>
</tr>
<tr>
<td>Specifies variable selection strategies for evaluation</td>
<td>PROC CLP</td>
<td>EVALVARSEL=</td>
</tr>
<tr>
<td>Enables time limit updating for strategy evaluation</td>
<td>PROC CLP</td>
<td>DECRMAXTIME</td>
</tr>
<tr>
<td><strong>Standard CSP Statements</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the alldifferent constraints</td>
<td>ALLDIFF</td>
<td></td>
</tr>
<tr>
<td>Specifies the array specifications</td>
<td>ARRAY</td>
<td></td>
</tr>
<tr>
<td>Specifies the element constraints</td>
<td>ELEMENT</td>
<td></td>
</tr>
<tr>
<td>Specifies the foreach constraints</td>
<td>FOREACH</td>
<td></td>
</tr>
<tr>
<td>Specifies the global cardinality constraints</td>
<td>GCC</td>
<td></td>
</tr>
<tr>
<td>Specifies the lexicographic ordering constraints</td>
<td>LEXICO</td>
<td></td>
</tr>
<tr>
<td>Specifies the linear constraints</td>
<td>LINCON</td>
<td></td>
</tr>
<tr>
<td>Specifies the bin packing constraints</td>
<td>PACK</td>
<td></td>
</tr>
<tr>
<td>Specifies the reified constraints</td>
<td>REIFY</td>
<td></td>
</tr>
<tr>
<td>Defines the variable specifications</td>
<td>VARIABLE</td>
<td></td>
</tr>
</tbody>
</table>
The PROC CLP statement invokes the CLP procedure. You can specify options to control a variety of search parameters that include selection strategies, assignment strategies, maximum running time, and number of solution attempts. You can specify the following options:

**CONDATA=**CAS-libref.data-table
identifies the input data table that defines the constraints, variable types, and variable bounds and an objective function. The CONDATA= data table provides support for linear constraints only. **CAS-libref.data-table** is a two-level name, where **CAS-libref** refers to the caslib and session identifier, and **data-table** specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 19 in Chapter 4, “Shared Concepts.”

You can also specify the linear constraints in-line by using the LINCON statement. The CLP procedure enables you to define constraints by using a combination of the two specifications. When defining constraints, you must define the variables by using a VARIABLE statement or implicitly define them by specifying the USECONDATAVARS= option when using the CONDATA= data table. You can define variable bounds by using the VARIABLE statement, and any such definitions override the implicit definitions of variables in the CONDATA= data table.

**DECRMAXTIME**
dynamically decreases the maximum solution time in effect during evaluation of the variable selection strategy. The DECRMAXTIME option is effective only when the EVALVARSEL= option is specified. Initially, the maximum solution time is the value specified by the MAXTIME= option. Whenever a solution is found with the current variable selection strategy, the value of MAXTIME for future attempts is reduced to the current solution time. The DECRMAXTIME option thus provides a sense of which strategy is the fastest for a given problem. However, you must use caution when comparing the strategies, because the individual strategy results might pertain to different time limits.

By default, the DECRMAXTIME option is disabled; each variable selection strategy is given the amount of time specified by the MAXTIME= option.

**DOMAIN=**[lb, ub]
**DOM=[lb, ub]**
specifies the global domain of all variables to be the closed interval [lb, ub]. You can override the global domain for a variable with a VARIABLE statement or the CONDATA= data table. The default domain is [0, 1].

**EVALVARSEL=<**(keyword(s))**>**
evaluates specified variable selection strategies by attempting to find a solution with each strategy. You can specify any combination of valid variable selection strategies in a space-delimited list enclosed in parentheses. If you do not specify a list, all available strategies are evaluated in alphabetical order, except that the default strategy is evaluated first. Descriptions of the available selection strategies are provided in the discussion of the VARSELECT= option.

When the EVALVARSEL= option is in effect, the MAXTIME= option must also be specified. By default, the value specified for the MAXTIME= option is used as the maximum solution time for each
variable selection strategy. When the \texttt{DECRMAXTIME} option is specified and a solution has been found, the value of the \texttt{MAXTIME=} option is set to the solution time of the current solution.

After the CLP procedure has attempted to find a solution with a particular strategy, it proceeds to the next strategy in the list. For this reason, the \texttt{VARSELECT=}, \texttt{ALLSOLNS}, and \texttt{MAXSOLNS=} options are ignored when the \texttt{EVALVARSEL=} option is in effect. All solutions that are found during the evaluation process are saved in the output data table that you specify in the \texttt{OUT=} option.

The macro variable \texttt{_ORCLPEVS=} provides more information that is related to the evaluation of each variable selection strategy. The fastest variable selection strategy is indicated in the macro variable \texttt{_ORCLP=}, provided that either at least one solution is found or the problem is found to be infeasible. See “Macro Variable \texttt{_ORCLP=}” on page 52 for more information about the \texttt{_ORCLP=} macro variable; see “Macro Variable \texttt{_ORCLPEVS=}” on page 53 for more information about the \texttt{_ORCLPEVS=} macro variable.

The \texttt{EVALVARSEL=} option enables you to evaluate strategies concurrently under certain conditions in a distributed environment. For more information, see the section “Parallel Processing” on page 49.

\texttt{FINDALLSOLNS}
\texttt{ALLSOLNS}
\texttt{FINDALL}

attempts to find all possible solutions to the CSP.

\texttt{MAXSOLNS=}n

specifies the maximum number of solutions to attempt to find. The default value is 1. It is important to note that the specified number of solutions might not be attainable in the time allotted; therefore the total number of solutions found might be fewer than the number given by the \texttt{MAXSOLNS=} parameter.

\texttt{MAXTIME=}n

specifies an upper bound on the number of seconds that are allocated for solving the problem. The type of time, either CPU time or real time, is determined by the value of the \texttt{TIMETYPE=} option. The default type is CPU time. The time specified by the \texttt{MAXTIME=} option is checked only once at the end of each iteration. Therefore, the actual running time can be longer than that specified by the \texttt{MAXTIME=} option. The difference depends on how long the last iteration takes. The default value of \texttt{MAXTIME=} is $\infty$. If you do not specify this option, the procedure does not stop based on the amount of time elapsed.

\texttt{NOPREPROCESS}

suppresses any preprocessing that would typically be performed for the problem.

\texttt{OUT=}\texttt{CAS-libref.data-table}

identifies the output data table that contains one or more solutions to a standard CSP, if one exists. This option is required in order to run the CLP procedure. \texttt{CAS-libref.data-table} is a two-level name, where \texttt{CAS-libref} refers to the caslib and session identifier, and \texttt{data-table} specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 19 in Chapter 4, “Shared Concepts.” Each observation in the \texttt{OUT=} data table corresponds to a solution of the CSP. You can control the number of solutions that are generated by specifying the \texttt{MAXSOLNS=} option in the \texttt{PROC CLP} statement.
**PREPROCESS**
permits any preprocessing that would typically be performed for the problem.

**SHOWPROGRESS**
prints a message to the log whenever a solution has been found.

**TIMETYPE=CPU | REAL**
specifies whether CPU time or real time is used for evaluation of the MAXTIME= option and the reporting of timing values in the ODS tables and macro variables. The default value of this option is CPU.

**USECONDATAVARS=0 | 1**
specifies whether the numeric variables in the CONDATA= data table, with the exception of any reserved variables, are implicitly defined or not. A value of 1 indicates they are implicitly defined, in which case a VARIABLE statement is not necessary to define the variables specified in the data table. The default value is 0. Currently, _RHS_ is the only reserved numeric variable.

**VARASSIGN=MAX | MIN**
specifies the value selection strategy. Currently, there are two strategies:
- MAX, which selects the maximum value from the domain of the selected variable
- MIN, which selects the minimum value from the domain of the selected variable

The default value selection strategy is MIN.

**VARSELECT=keyword**
specifies the variable selection strategy. The strategy can be static, dynamic or conflict-directed. Typically, static strategies exploit information about the initial state of the search, whereas dynamic strategies exploit information about the current state of the search process. Conflict-directed strategies are strategies which exploit information from previous states of the search process as well as the current state (Boussemart et al. 2004)

Static strategies are as follows:
- FIFO, which uses the first-in-first-out ordering of the variables as encountered by the procedure
- MAXCS, which selects the variable with the maximum number of constraints (degree)

Dynamic strategies are as follows:
- DOMDDEG, which selects the variable with the smallest ratio of domain size by dynamic degree
- DOMDEG, which selects the variable with the smallest ratio of domain size by degree
- MAXC, which selects the variable with the largest number of active constraints (dynamic degree)
- MINR, which selects the variable with the smallest range (that is, the minimum value of upper bound minus lower bound)
- MINRMAXC, which selects the variable with the smallest range, breaking ties by selecting one with the largest number of active constraints

Conflict-directed strategies are as follows:
- DOMWDEG, which selects the variable with the smallest ratio of domain size by weighted degree
• WDEG, which selects the variable with the largest weighted degree

The dynamic strategies embody the “fail first principle” (FFP) of Haralick and Elliott (1980), which suggests that “To succeed, try first where you are most likely to fail.” The degree of a variable is the number of constraints in which the variable appears. The dynamic degree of a variable is the degree of the variable with respect to the current set of active constraints. Boussemart et al. (2004) introduced the concept of weighted degree, which takes into account dead ends during searches. Weights are associated with constraints and variables. The weight of a constraint is initially set to 1 and incremented each time the constraint is responsible for a dead end. The weight of a variable is equal to the sum of the weights of the active constraints that it is associated with. The default variable selection strategy is MINR.

**ALLDIFF Statement**

\[
\text{ALLDIFF (variable_list-1) < . . . (variable_list-n)> ;}
\]

\[
\text{ALLDIFFERENT (variable_list-1) < . . . (variable_list-n)> ;}
\]

The ALLDIFF statement can have multiple specifications. Each specification defines a unique global constraint on a set of variables, requiring all of them to be different from each other. A global constraint is equivalent to a conjunction of elementary constraints.

For example, the statements

\[
\begin{align*}
\text{var (X1-X3) A B;}
\text{alldiff (X1-X3) (A B);} \\
\end{align*}
\]

are equivalent to

\[
\begin{align*}
X1 & \neq X2 \text{ AND} \\
X2 & \neq X3 \text{ AND} \\
X1 & \neq X3 \text{ AND} \\
A & \neq B
\end{align*}
\]

If the variable list is empty, the ALLDIFF constraint applies to all the variables.

**ARRAY Statement**

\[
\text{ARRAY specification-1 < . . . specification-n>;}
\]

An ARRAY specification is in a form as follows:

\[
\text{name(dimension)(variables)}
\]

The ARRAY statement is used to associate a name with a list of variables. Each of the variables in the variable list must be defined using a VARIABLE statement or implicitly defined using the CONDATA= data table. The ARRAY statement is required when you are specifying a constraint by using the FOREACH statement.


**ELEMENT Statement**

**ELEMENT** `element_constraint-1 < ... element_constraint-n > ;`

An `element_constraint` is specified in the following form:

`(index variable, (integer list), variable)`

The ELEMENT statement specifies one or more element constraints. An element constraint enables you to define dependencies, not necessarily functional, between variables. The statement

**ELEMENT**`(I, (L), V)`

sets the variable `V` to equal to the `I`th element in the list `L`. The list of integers `L = (v_1, ..., v_n)` is a list of values that are potentially assigned to the variable `V` and are not necessarily distinct. The variable `I` is the index variable, and its domain is considered to be `[1, n]`. Each time the domain of `I` is modified, the domain of `V` is updated and vice versa.

An element constraint enforces the following propagation rules:

\[ V = v \iff I \in \{i_1, ..., i_m\} \]

where `v` is a value in the list `L` and `i_1, ..., i_m` are all the indices in `L` whose value is `v`.

The following statements use the element constraint to implement the quadratic function \( y = x^2 \):

```plaintext
proc clp out=mycas.clpout;
  var x=[1,5] y=[1,25];
  element (x,(1, 4, 9, 16, 25), y);
run;
```

An element constraint is equivalent to a conjunction of reified and linear constraints. For example, the preceding statements are equivalent to:

```plaintext
proc clp out=mycas.clpout;
  var x=[1,5] y=[1,25] (R1-R5)=[0,1];
  reify R1: (x=1);
  reify R1: (y=1);
  reify R2: (x=2);
  reify R2: (y=4);
  reify R3: (x=3);
  reify R3: (y=9);
  reify R4: (x=4);
  reify R4: (y=16);
  reify R5: (x=5);
  reify R5: (y=25);
  lincon R1 + R2 + R3 + R4 + R5 = 1;
run;
```

Element constraints can also be used to define positional mappings between two variables. For example, suppose the function \( y = x^2 \) is defined on only odd numbers in the interval \([-5, 5]\). You can model this by using two element constraints and an artificial index variable:
Chapter 5: The CLP Procedure

```
  element (i, ( -5, -3, -1, 1, 3, 5), x)
  (i, ( 25, 9, 1, 1, 9, 25), y);
```

The list of values \( L \) can also be specified by using a convenient syntax of the form `start TO end` or `start TO end BY increment`. For example, the previous element specification is equivalent to:

```
  element (i, ( -5 to 5 by 2), x)
  (i, ( 25, 9, 1, 1, 9, 25), y);
```

---

**FOREACH Statement**

```
FOREACH (array, type, <offset>);
```

where `array` must be defined by using an `ARRAY` statement, `type` is a keyword that determines the type of the constraint, and `offset` is an integer.

The FOREACH statement iteratively applies a constraint over an array of variables. The type of the constraint is determined by `type`. Currently, the only valid `type` keyword is `DIFF`. The optional `offset` parameter is an integer and is interpreted in the context of the constraint type. The default value of `offset` is zero.

The FOREACH statement that corresponds to the `DIFF` keyword iteratively applies the following constraint to each pair of variables in the array:

```
  variable_i + offset \times i \neq variable_j + offset \times j \ \forall \ i \neq j, \ i, j = 1, \ldots, \text{array\_dimension}
```

For example, the constraint that all \((A[i] - i)\)’s are pairwise distinct for an array \(A\) is expressed as

```
  foreach (A, diff, -1);
```

---

**GCC Statement**

```
GCC global\_cardinality\_constraint-1 <\ldots global\_cardinality\_constraint-n> ;
```

where `global\_cardinality\_constraint` is specified in the following form:

```
  (variables) = ((v_1, l_1, u_1) <\ldots (v_n, l_n, u_n) > <DL= dl> <DU= du>)
```

\(v_i\) is a value in the domain of one of the variables, and \(l_i\) and \(u_i\) are the lower and upper bounds on the number of variables assigned to \(v_i\). The values of \(dl\) and \(du\) are the lower and upper bounds on the number of values assigned to values outside of \(\{v_1, \ldots, v_n\}\).

The GCC statement specifies one or more global cardinality constraints. A `global\_cardinality\_constraint` (GCC) is a constraint that consists of a set of variables \(\{x_1, \ldots, x_n\}\) and for each value \(v\) in \(D = \bigcup_{i=1}^{n} \text{Dom}(x_i)\), a pair of numbers \(l_v\) and \(u_v\). A GCC is satisfied if and only if the number of times that a value \(v\) in \(D\) is assigned to the variables \(x_1, \ldots, x_n\) is at least \(l_v\) and at most \(u_v\).

For example, the constraint that is specified with the statements
var (x1-x6) = [1, 4];
gcc(x1-x6) = ((1, 1, 2) (2, 1, 3) (3, 1, 3) (4, 2, 3));
expresses that at least one but no more than two variables in $x_1, \ldots, x_6$ can have value 1, at least one and no more than three variables can have value 2 (or value 3), and at least two and no more than three variables can have value 4. For example, an assignment $x_1 = 1, x_2 = 1, x_3 = 2, x_4 = 3, x_5 = 4,$ and $x_6 = 4$ satisfies the constraint.

If a global cardinality constraint has common lower or upper bounds for many of the values in $D$, the DL= and DU= options can be used to specify the common lower and upper bounds.

For example, the previous specification could also be written as

gcc(x1-x6) = ((1, 1, 2) (4, 2, 3) DL=1 DU=3);

You can also specify missing values for the lower and upper bounds. The values of $dl$ and $du$ are substituted as appropriate. The previous example can also be expressed as

gcc(x1-x6) = ((1, ., 2) (4, 2, .) DL=1 DU=3);

The following statements specify that each of the values in $\{1, \ldots, 9\}$ can be assigned to at most one of the variables $x_1, \ldots, x_9$:

var (x1-x9) = [1, 9];
gcc(x1-x9) = (DL=0 DU=1);

Note that the preceding global cardinality constraint is equivalent to the alldifferent constraint that is expressed as:

var (x1-x9) = [1, 9];
alldiff(x1-x9);

If you do not specify the DL= and DU= options, the default lower and upper bound for any value in $D$ that does not appear in the $v,l,u$ format is 0 and the number of variables in the constraint, respectively.

The global cardinality constraint also provides a convenient way to define disjoint domains for a set of variables. For example, the following syntax limits assignment of the variables $x_1, \ldots, x_9$ to even numbers between 0 and 10:

var (x1-x9) = [0, 10];
gcc(x1-x9) = ((1, 0, 0) (3, 0, 0) (5, 0, 0) (7, 0, 0) (9, 0, 0));

If the variable list is empty, the GCC constraint applies to all the variables.
where \texttt{variable-list-1} and \texttt{variable-list-2} are variable lists of equal length. The keyword \texttt{order_type} signifies the type of ordering and can be one of two values: LEX\_LE, which indicates lexicographically less than or equal to ($\leq_{\text{lex}}$), or LEX\_LT, which indicates lexicographically less than ($<_{\text{lex}}$).

The LEXICO statement specifies one or more lexicographic constraints. The lexicographic constraint $\leq_{\text{lex}}$ and the strict lexicographic constraint $<_{\text{lex}}$ are defined as follows. Given two $n$-tuples $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_n)$, the $n$-tuple $x$ is lexicographically less than or equal to $y$ ($x \leq_{\text{lex}} y$) if and only if

\[(x_i = y_i \ \forall i = 1, \ldots, n) \lor (\exists j \text{ with } 1 \leq j \leq n \text{ such that } x_i = y_i \ \forall i = 1, \ldots, j-1 \text{ and } x_j < y_j)\]

The $n$-tuple $x$ is lexicographically less than $y$ ($x <_{\text{lex}} y$) if and only if $x \leq_{\text{lex}} y$ and $x \neq y$. Equivalently, $x <_{\text{lex}} y$ if and only if

\[\exists j \text{ with } 1 \leq j \leq n \text{ such that } x_i = y_i \ \forall i = 1, \ldots, j-1 \text{ and } x_j < y_j\]

Frisch et al. (2002) introduced an optimal algorithm to establish generalized arc consistency (GAC) for the $\leq_{\text{lex}}$ constraint between two vectors of variables. Informally you can think of the lexicographic constraint $\leq_{\text{lex}}$ as sorting the $n$-tuples in alphabetical order. Mathematically, $\leq_{\text{lex}}$ is a partial order on a given subset of $n$-tuples, and $<_{\text{lex}}$ is a strict partial order on a given subset of $n$-tuples (Brualdi 2010).

For example, you can express the lexicographic constraint $(x_1, \ldots, x_6) \leq_{\text{lex}} (y_1, \ldots, y_6)$ by using a LEXICO statement as follows:

\[
\text{lexico}((x_1-x_6) \text{ lex_le } (y_1-y_6));
\]

The assignment $x_1 = 1, x_2 = 2, x_3 = 2, x_4 = 1, x_5 = 2, x_6 = 5, y_1 = 1, y_2 = 2, y_3 = 2, y_4 = 1, y_5 = 4, \text{ and } y_6 = 3$ satisfies this constraint because $x_i = y_i \text{ for } i = 1, \ldots, 4$ and $x_5 < y_5$. The fact that $x_6 > y_6$ is irrelevant in this ordering.

Lexicographic ordering constraints can be useful for breaking a certain kind of symmetry that arises in CSPs with matrices of decision variables.

---

**LINCON Statement**

\[
\text{LINCON linear_constraint-1 < \ldots, linear_constraint-n> ;}
\]

\[
\text{LINEAR linear_constraint-1 < \ldots, linear_constraint-n> ;}
\]

where \texttt{linear_constraint} has the form

\[
\text{linear_expression-l type linear_expression-r}
\]

where \texttt{linear_expression} has the form

\[
<+|-> \text{linear_term-1 < \ldots, (+|-) linear_term-n>}
\]

where \texttt{linear_term} has the form

\[
\text{(variable | number< * variable>)}
\]

The keyword \texttt{type} can be one of the following:
The LINCON statement allows for a very general specification of linear constraints. In particular, it allows for specification of the following types of equality or inequality constraints:

\[
\sum_{j=1}^{n} a_{ij} x_j \{ \leq, <, =, \geq, >, \neq \} b_i \quad \text{for } i = 1, \ldots, m
\]

For example, the constraint \(4x_1 - 3x_2 = 5\) can be expressed as

```plaintext
var x1 x2;
lincon 4 * x1 - 3 * x2 = 5;
```

and the constraints

\[
\begin{align*}
10x_1 - x_2 & \geq 10 \\
x_1 + 5x_2 & \neq 15
\end{align*}
\]

can be expressed as

```plaintext
var x1 x2;
lincon 10 <= 10 * x1 - x2, 
      x1 + 5 * x2 <> 15;
```

Note that variables can be specified on either side of an equality or inequality in a LINCON statement. Linear constraints can also be specified by using the CONDATA= data table.

Regardless of the specification, you must define the variables by using a VARIABLE statement or implicitly by specifying the USECONDATAVARS= option.

User-specified scalar values are subject to rounding based upon a platform-dependent tolerance.

---

### OBJ Statement

**OBJ options**

The OBJ statement enables you to set upper and lower bounds on the value of an objective function that is specified in the Constraint data table. You can also use the OBJ statement to specify the tolerance used for finding a locally optimal objective value.

If upper and lower bounds for the objective value are not specified, the CLP procedure tries to derive bounds from the domains of the variables that appear in the objective function. The procedure terminates with an error message if the objective is unbounded.

You can specify the following options in the OBJ statement:

- **LB=m**
  
  specifies the lower bound of the objective value.
TOL=m
specifies the tolerance of the objective value. The tolerance must not be less than 1E–6, which is the
default.

UB=m
specifies the upper bound of the objective value.

For more information about using an objective function, see the section “Objective Function” on page 48.

**PACK Statement**

PACK bin_packing_constraint-1 < ... bin_packing_constraint-n> ;

where bin_packing_constraint is specified by the form

\((b_1 < ... b_k >) (s_1 < ... s_k >) (l_1 < ... l_m >)\)

The PACK constraint is used to assign \(k\) items to \(m\) bins, subject to the sizes of the items and the capacities of the bins. The item variable \(b_i\) assigns a bin to the \(i\)th item so that the domain for the item variables is 1 to \(m\). Optionally, the domain for the item variables can be any contiguous set up to size \(m\), where the smallest member represents the first bin, the second-smallest member represents the second bin, and so on. The constant \(s_i\) gives the size or weight of the \(i\)th item. The value of load variable \(l_j\) is determined by the total size of the contents of the corresponding bin.

For example, suppose there are three bins with capacities 3, 4, and 5. There are five items with sizes 4, 3, 2, 2, and 1 to be assigned to these three bins. The following statements formulate the problem and find all solutions:

```
proc clp out=mycas.out findallsolns;
  var bin1 = [0,3];
  var bin2 = [0,4];
  var bin3 = [0,5];
  var (item1-item5) = [1,3];
  pack ((item1-item5) (4,3,2,2,1) (bin1-bin3));
run;
```

Each row of Table 5.2 represents a solution to the problem. The number in each item column is the number of the bin to which the corresponding item is assigned.

<table>
<thead>
<tr>
<th>Item Variable</th>
<th>item1</th>
<th>item2</th>
<th>item3</th>
<th>item4</th>
<th>item5</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

**NOTE:** In specifying a PACK constraint, it can be more efficient to list the item variables in order by nonincreasing size and to specify VARSELECT=FIFO in the PROC CLP statement so that load variables are selected last.
REIFY Statement

REIFY reify_constraint-1 < ... reify_constraint-n> ;

where reify_constraint is specified in the following form:

variable : constraint

The REIFY statement associates a binary variable with a constraint. The value of the binary variable is 1 or 0 depending on whether the constraint is satisfied or not, respectively. The constraint is said to be reified, and the binary variable is referred to as the control variable. Currently, the only type of constraint that can be reified is the linear constraint, which should have the same form as linear_constraint defined in the LINCON statement. As with the other variables, the control variable must also be defined in a VARIABLE statement or in the CONDATA= data table.

The REIFY statement provides a convenient mechanism for expressing logical constraints, such as disjunctive and implicative constraints. For example, the disjunctive constraint

\[(3x + 4y < 20) \lor (5x - 2y > 50)\]

can be expressed with the following statements:

```
var x y p q;
reify p: (3 * x + 4 * y < 20) q: (5 * x - 2 * y > 50);
lincon p + q >= 1;
```

The binary variables \(p\) and \(q\) reify the linear constraints

\[3x + 4y < 20\]

and

\[5x - 2y > 50\]

respectively. The following linear constraint enforces the desired disjunction:

\[p + q \geq 1\]

The implication constraint

\[(3x + 4y < 20) \Rightarrow (5x - 2y > 50)\]

can be enforced with the linear constraint

\[q - p \geq 0\]

The REIFY constraint can also be used to express a constraint that involves the absolute value of a variable. For example, the constraint

\[|X| = 5\]

can be expressed with the following statements:
Chapter 5: The CLP Procedure

```plaintext
var x p q;
reify p: (x = 5) q: (x = -5);
lincon p + q = 1;
```

### VARIABLE Statement

```plaintext
VARIABLE var_specification-1 < . . . var_specification-n > ;
VAR var_specification-1 < . . . var_specification-n > ;
```

A `var_specification` can be one of the following types:

- `variable <= [lower-bound < , upper-bound >]`
- `(variables) <= [lower-bound < , upper-bound >]`

The VARIABLE statement declares all variables that are to be considered in the CSP and, optionally, defines their domains. Any variable domains defined in a VARIABLE statement override the global variable domains that are defined by using the `DOMAIN=` option in the PROC CLP statement in addition to any bounds that are defined by using the `CONDATA=` data table. The values of `lower-bound` and `upper-bound` can also be specified as missing, in which case the appropriate values from the `DOMAIN=` specification are substituted.

### Details: CLP Procedure

#### Type of Solvers

The CLP procedure features two different solvers. The choice of which one to use is determined by the nature of the CSP that you are attempting to solve.

- The **standard solver** is used to solve CSPs that are defined by ALLDIFFERENT constraints, element constraints, GCC constraints, lexicographic constraints, linear constraints, pack constraints, reified constraints, ARRAY statements, and FOREACH statements. When you invoke the standard solver, the decision variables are one-dimensional; a variable is assigned an integer in a solution.

- The **scheduling solver** is used to solve CSPs that are defined by scheduling-specific constraints, such as temporal constraints (precedence and time) and resource constraints. When you use the scheduling solver, the variables are typically multidimensional; a variable is assigned a start time and possibly a set of resources in a solution. Furthermore, the variables are referred to as activities, and the solution is referred to as a schedule.

The scheduling solver runs only on the SAS client and requires a SAS/OR license.
**Constraint Data Table**

The Constraint data table defines linear constraints, variable types, bounds on variable domains, and an objective function. You can use a Constraint data table in lieu of, or in combination with, a LINCON or a VARIABLE statement (or both) in order to define linear constraints, variable types, and variable bounds. You can use the Constraint data table in lieu of, or in combination with, the OBJ statement to specify an objective function. The Constraint data table is specified by using the CONDATA= option in the PROC CLP statement.

The Constraint data table must be in dense input format. In this format, a model’s columns appear as variables in the input data table and the data table must contain the _TYPE_ variable, at least one numeric variable, and any reserved variables. Currently, the only reserved variable is the _RHS_ variable. If this requirement is not met, the CLP procedure terminates. The _TYPE_ variable is a character variable that tells the CLP procedure how to interpret each observation. The CLP procedure recognizes the following keywords as valid values for the _TYPE_ variable: EQ, LE, GE, NE, LT, GT, LOWERBD, UPPERBD, BINARY, FIXED, MAX, and MIN. An optional character variable, _ID_, can be used to name each row in the Constraint data table.

**Linear Constraints**

For the _TYPE_ values EQ, LE, GE, NE, LT, and GT, the corresponding observation is interpreted as a linear constraint. The _RHS_ variable is a numeric variable that contains the right-hand-side coefficient of the linear constraint. Any numeric variable other than _RHS_ that appears in a VARIABLE statement is interpreted as a structural variable for the linear constraint.

The _TYPE_ values are defined as follows:

- **EQ (=)** defines a linear equality of the form
  \[ \sum_{j=1}^{n} a_{ij}x_j = b_i \]

- **LE (<=)** defines a linear inequality of the form
  \[ \sum_{j=1}^{n} a_{ij}x_j \leq b_i \]

- **GE (>=)** defines a linear inequality of the form
  \[ \sum_{j=1}^{n} a_{ij}x_j \geq b_i \]

- **NE (<>)** defines a linear inequality of the form
  \[ \sum_{j=1}^{n} a_{ij}x_j \neq b_i \]

- **LT (<)** defines a linear inequality of the form
  \[ \sum_{j=1}^{n} a_{ij}x_j < b_i \]
GT (>) defines a linear inequality of the form

\[ \sum_{j=1}^{n} a_{ij} x_j > b_i \]

**Domain Bounds**

The keywords LOWERBD and UPPERBD specify additional lower bounds and upper bounds, respectively, on the variable domains. In an observation where the _TYPE_ variable is equal to LOWERBD, a non-missing value for a decision variable is considered to be a lower bound for that variable. Similarly, in an observation where the _TYPE_ variable is equal to UPPERBD, a non-missing value for a decision variable is considered to be an upper bound for that variable. Note that lower and upper bounds defined in the Constraint data table are overridden by lower and upper bounds that are defined by using a VARIABLE statement.

**Variable Types**

The keywords BINARY and FIXED specify numeric variable types. If the value of _TYPE_ is BINARY for an observation, then any decision variable with a non-missing entry for the observation is interpreted as being a binary variable with domain \{0, 1\}. If the value of _TYPE_ is FIXED for an observation, then any decision variable with a non-missing entry for the observation is interpreted as being assigned to that non-missing value. In other words, if the value of the variable X is c in an observation for which _TYPE_ is FIXED, then the domain of X is considered to be the singleton \{c\}. The value c should belong to the domain of X, or the problem is deemed infeasible.

**Objective Function**

The keywords MAX and MIN specify the objective function of a maximization or a minimization problem, respectively. In an observation where the _TYPE_ variable is equal to MAX or MIN, a non-missing value for a decision variable is the coefficient of this variable in the objective function. The value specified for _RHS_ is ignored in this case.

The bisection method is used to find the optimal objective value within the specified or derived lower and upper bounds. A solution is considered optimal if the difference between consecutive objective values is less than or equal to the tolerance. You can use the OBJ statement to specify the tolerance in addition to upper and lower bounds on the objective value. The minimum objective tolerance is 1E–6, which is the default.

When an optimal solution is found, the solution is stored in the output data table and the resulting objective value is stored in the macro variable _ORCLP_. The objective value is not necessarily optimal when it is computed within a time limit specified by the MAXTIME= option. In this case, the last valid solution computed within the time limit appears in the output data table. See the macro variable _ORCLP_ for more information about solution status.

The MAX and MIN functions are defined as follows:

MAX defines an objective function of the form

\[ \max \sum_{j=1}^{n} c_j x_j \]
MIN defines an objective function of the form
\[ \min \sum_{j=1}^{n} c_j x_j \]

Variables in the CONDATA= Data Table

Table 5.3 lists all the variables that are associated with the Constraint data table and their interpretations by the CLP procedure. For each variable, the table also lists its type (C for character, N for numeric), the possible values it can assume, and its default value.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Description</th>
<th>Allowed Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>TYPE</em></td>
<td>C</td>
<td>Observation type</td>
<td>EQ, LE, GE, NE, LT, GT, LOWERBD, UPPERBD, BINARY, FIXED, MAX, MIN</td>
<td></td>
</tr>
<tr>
<td><em>RHS</em></td>
<td>N</td>
<td>Right-hand-side coefficient</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td><em>ID</em></td>
<td>C</td>
<td>Observation name</td>
<td>(optional)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Any numeric variable other</td>
<td>Structural variable</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>than <em>RHS</em></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Solution Data Table

In order to solve a standard CSP, you need to specify a Solution data table by using the OUT= option in the PROC CLP statement. The Solution data table contains all the solutions that have been determined by the CLP procedure. You can specify an upper bound on the number of solutions by using the MAXSOLNS= option in the PROC CLP statement. If you prefer that PROC CLP determine all possible solutions instead, you can specify the FINDALL option in the PROC CLP statement.

The Solution data table contains as many decision variables as have been defined in the CLP procedure invocation. Every observation in the Solution data table corresponds to a solution to the CSP. If a Constraint data table has been specified, then any variable formats and variable labels from the Constraint data table carry over to the Solution data table.

Parallel Processing

You can solve a CSP in single-machine mode or distributed mode. In single-machine mode, the computation is executed on a single computer.

In distributed mode, the CSP is statically decomposed into a set of mutually exclusive and exhaustive CSPs, in that the union of their solution sets represents the solution set to the original CSP. Each of the subproblem
CSPs is solved on one of the computing nodes in a distributed computing environment by using the same variable selection and variable assignment strategies that were specified for the original problem. Because of the nature of distributed computing, the order in which the solutions are found will not necessarily be the same as when you are solving the original problem on a single node.

CSPs that have an objective function will run only in single-machine mode.

In addition, the EVALVARSEL= option enables you to take advantage of a distributed system to run concurrently, provided that there are at least as many computing nodes as there are strategies for evaluation. The original problem is executed on grid nodes by using a different node for each variable selection strategy. The DECRMAXTIME option causes the procedure to terminate when a solution is found or the problem is determined to be infeasible. If the number of computing nodes is less than the number of strategies for evaluation, the problem will run in single-machine mode.

**ODS Tables**

PROC CLP creates two Output Delivery System (ODS) tables by default. The first table, ProblemSummary, is a summary of the input CSP. The second table, SolutionSummary, is a brief summary of the solution status. You can use the ODS OUTPUT statement with table names to select tables and create output data sets. For more information about ODS, see *SAS Output Delivery System: User’s Guide*.

Output 5.3 shows an example of a ProblemSummary table.

**Output 5.3** PROC CLP Output: Problem Summary

<table>
<thead>
<tr>
<th>The CLP Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Summary</td>
</tr>
<tr>
<td>Number of Variables</td>
</tr>
<tr>
<td>Number of Constraints</td>
</tr>
<tr>
<td>ALLDIFF</td>
</tr>
<tr>
<td>ELEMENT</td>
</tr>
<tr>
<td>FOREACH</td>
</tr>
<tr>
<td>GCC</td>
</tr>
<tr>
<td>LEXICO</td>
</tr>
<tr>
<td>LINCON</td>
</tr>
<tr>
<td>PACK</td>
</tr>
<tr>
<td>REIFY</td>
</tr>
</tbody>
</table>

Output 5.4 shows an example of a SolutionSummary table.
Output 5.4  PROC CLP Output: Solution Summary

The CLP Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solution Status</strong></td>
</tr>
<tr>
<td><strong>Variable Selection</strong></td>
</tr>
<tr>
<td><strong>Value Selection</strong></td>
</tr>
</tbody>
</table>

| Solutions Found | 5361 |
| Solution Time   | 27.78 |

When you specify the EVALVARSEL= option, an EvalInfo ODS table is created. The EvalInfo table provides information that is related to the evaluation of each variable selection strategy. It contains four columns, which identify the selection strategy, evaluation time, solution index, and number of variables selected.

Output 5.5 shows an example of an EvalInfo table.

Output 5.5  PROC CLP Output: Strategy Evaluation

The CLP Procedure

<table>
<thead>
<tr>
<th>Variable Selection</th>
<th>Strategy Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection Strategy</td>
<td>Evaluation Time</td>
</tr>
<tr>
<td>MINR</td>
<td>1</td>
</tr>
<tr>
<td>FIFO</td>
<td>0.55</td>
</tr>
<tr>
<td>MAXC</td>
<td>0.56</td>
</tr>
<tr>
<td>MAXCS</td>
<td>0.55</td>
</tr>
<tr>
<td>MINRMAXC</td>
<td>1</td>
</tr>
<tr>
<td>WDEG</td>
<td>0.03</td>
</tr>
<tr>
<td>DOMDDEG</td>
<td>1</td>
</tr>
<tr>
<td>DOMWDEG</td>
<td>1</td>
</tr>
<tr>
<td>DOMDEG</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.4 summarizes all the ODS tables that the CLP procedure can produce, along with the statement and option specifications that are required in order to produce each table.

Table 5.4  ODS Tables Produced by PROC CLP

<table>
<thead>
<tr>
<th>ODS Table Name</th>
<th>Description</th>
<th>Statement</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProblemSummary</td>
<td>Summary of the input CSP</td>
<td>PROC CLP</td>
<td>(default)</td>
</tr>
<tr>
<td>SolutionSummary</td>
<td>Summary of the solution status</td>
<td>PROC CLP</td>
<td>(default)</td>
</tr>
<tr>
<td>EvalInfo</td>
<td>Detailed EVALVARSEL= option statistics</td>
<td>PROC CLP</td>
<td>EVALVARSEL=</td>
</tr>
</tbody>
</table>
Macro Variable _ORCLP_

The CLP procedure defines a macro variable named _ORCLP_. This variable contains a character string that indicates the status of the CLP procedure upon termination. The various terms of the macro variable are interpreted as follows.

**STATUS**

indicates the procedure status at termination. It can take one of the following values:

- **OK**: The procedure terminated successfully.
- **DATA_ERROR**: An input data error occurred.
- **IO_ERROR**: A problem in reading or writing data occurred.
- **MEMORY_ERROR**: Insufficient memory is allocated to the procedure.
- **SEMANTIC_ERROR**: The use of semantic action is incorrect.
- **SYNTAX_ERROR**: The use of syntax is incorrect.
- **ERROR**: The status cannot be classified into any of the preceding categories.

If PROC CLP terminates normally or if an I/O error is detected while the procedure is closing a data table, the following terms are added to the macro variable.

**SOLUTION_STATUS**

indicates the solution status at termination. It can take one of the following values:

- **ALL_SOLUTIONS**: All solutions are found.
- **INFEASIBLE**: The problem is infeasible.
- **OPTIMAL**: The solution is optimal.
- **SOLN_LIMIT_REACHED**: The required number of solutions specified with the MAXSOLNS= option is reached.
- **TIME_LIMIT_REACHED**: The execution time limit specified with the MAXTIME= option is reached.
- **EVALUATION_COMPLETE**: The evaluation process is finished. This solution status appears only when the EVALVARESEL= option is specified.
- **ABORT**: The procedure is stopped by the user before any other stopping criterion is reached.

**SOLUTIONS_FOUND**

indicates the number of solutions that are found. This term is not applicable if **SOLUTION_STATUS=INFEASIBLE**.

**SOLUTION_TIME**

indicates the time taken to solve the problem. By default, the time reported is CPU time; see the **TIMETYPE**= option for more information.
Macro Variable _ORCLPEVS_

When you specify the EVALVARSEL= option to evaluate variable selection strategies for a standard CSP, the CLP procedure defines a macro variable named _ORCLPEVS_. This variable contains a character string that describes the solution attempt made with each selection strategy. The macro variable contains four terms for each selection strategy that is evaluated as follows:

**VARSELTYPE**
indicates the variable selection strategy being evaluated.

**EVAL_TIME**
indicates the amount of time taken to evaluate the variable selection strategy. By default, the time reported is CPU time; see the TIMETYPE= option for more information.

**SOLUTION**
indicates the index of the solution if a solution is found. Otherwise, SOLUTION=INFEASIBLE if the problem is found to be infeasible, or SOLUTION=NOT_FOUND.

**MAX_VARSEL**
indicates the maximum number of variables selected within the evaluation time.

Examples: CLP Procedure

This section contains several examples that illustrate the capabilities of the different logical constraints and showcase a variety of problems that the CLP procedure can solve. The following examples feature a standard constraint satisfaction problem (CSP):

- “Example 5.1: Logic-Based Puzzles” illustrates the capabilities of the alldifferent constraint in solving a popular logical puzzle, the sudoku. This example also contains a variant of sudoku that illustrates the capabilities of the GCC constraint and demonstrates the use of the EVALVARSEL= option.

- “Example 5.2: Car Painting Problem” involves limited sequencing of cars in an assembly process in order to minimize the number of paint purgings; it features the reified constraint.

- “Example 5.3: Scene Allocation Problem” illustrates how to schedule the shooting of different movie scenes in order to minimize production costs. This problem uses the GCC and linear constraints.
• “Example 5.4: Car Sequencing Problem” relates to sequencing the cars on an assembly line with workstations for installing specific options subject to the demand constraints for each set of options and the capacity constraints of each workstation.

• “Example 5.5: Scheduling a Major Basketball Conference” illustrates modeling the problem of scheduling a major basketball conference. The element constraint plays a key role in this particular example.

• “Example 5.6: Balanced Incomplete Block Design” illustrates how to use the lexicographic constraint to break symmetries in generating a balanced incomplete block design (BIBD), a standard combinatorial problem from design theory.

• “Example 5.7: Progressive Party Problem” illustrates how to use the pack constraint to solve the progressive party problem, a well-known constraint programming problem in which crews of various sizes must be assigned to boats of various capacities for several rounds of parties.

• “Example 5.8: Radio Link Frequency Assignment Problem” illustrates how to use the element and reified constraints to solve a radio link frequency assignment problem.

---

**Example 5.1: Logic-Based Puzzles**

Many logic-based puzzles can be formulated as CSPs. Several such instances are shown in this example.

**Sudoku**

A sudoku is a logic-based, combinatorial number-placement puzzle that uses a partially filled 9×9 grid. The objective is to fill the grid with the digits 1 to 9, so that each column, each row, and each of the nine 3×3 blocks contain only one of each digit. **Output 5.1.1** shows an example of a sudoku grid.

**Output 5.1.1** An Example of an Unsolved Sudoku Grid

```
<table>
<thead>
<tr>
<th>5</th>
<th>7</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
```

This example illustrates the use of the ALLDIFFERENT constraint to solve the preceding sudoku problem.
The data set `indata` contains the partially filled values for the grid and is used to create the set of macro variables $C_{ij}$ ($i = 1\ldots9, j = 1\ldots9$), where $C_{ij}$ is the value of cell $(i, j)$ in the grid when specified, and missing otherwise.

```plaintext
data indata;
  input C1-C9;
datalines;
  . . 5 . . 7 . . 1
  . 7 . 9 . 3 .
  . . 6 . . .
  . 3 . 1 . 5
  . 9 . 8 . 2 .
  1 . . 2 . . 4 .
  . . 3 . . 1 . . 5
  . 9 . . 8 . . 2 .
  1 . . 2 . . 4 .
  . . 3 . . 1 . . 5;
%macro store_initial_values;
  /* store initial values into macro variable C_i_j */
data _null_; 
  set indata;
  array C{9};
  do j = 1 to 9;
    i = _N_; 
    call symput(compress('C_'||put(i,best.)||'_'||put(j,best.)), put(C[j],best.));
  end;
run;
%mend store_initial_values;
%store_initial_values;
```

Let the variable $X_{ij}$ ($i = 1\ldots9, j = 1\ldots9$) represent the value of cell $(i, j)$ in the grid. The domain of each of these variables is $[1, 9]$. Three sets of alldifferent constraints are used to set the required rules for each row, each column, and each of the 3x3 blocks. The constraint `ALLDIFF(X_{i1} - X_{i9})` forces all values in row $i$ to be different, the constraint `ALLDIFF(X_{j1} - X_{j9})` forces all values in column $j$ to be different, and the constraint `ALLDIFF(X_{ij}) ((i = 1, 2, 3; j = 1, 2, 3), (i = 1, 2, 3; j = 4, 5, 6), \ldots, (i = 7, 8, 9; j = 7, 8, 9))` forces all values in each block to be different.

The following statements solve the sudoku puzzle:

```plaintext
%macro solve;
  proc clp out=mycas.outdata;
    /* Declare variables */
    /* Nine row constraints */
    %do i = 1 %to 9;
      var (X_&i._1-X_&i._9) = [1,9];
      alldiff(X_&i._1-X_&i._9);
    %end;
    /* Nine column constraints */
    %do j = 1 %to 9;
      alldiff(
        %do i = 1 %to 9;
          X_&i._&j
        %end;
      )
    %end;
  run;
%mend solve;
```
 Chapter 5: The CLP Procedure

 /************************************************************************
 /* Nine 3x3 block constraints */
%do s = 0 %to 2;
%do t = 0 %to 2;
   alldiff(
      %do i = 3*s + 1 %to 3*s + 3;
      %do j = 3*t + 1 %to 3*t + 3;
       X_&i._&j
      %end;
      %end;
   %end;
%end;
%end;
%end;
%end;

 /************************************************************************
 /* Initialize variables to cell values */
 /* X_i_j = C_i_j if C_i_j is non-missing */
 %do i = 1 %to 9;
   %do j = 1 %to 9;
      %if &&C_&i._&j ne . %then %do;
       lincon X_&i._&j = &&C_&i._&j;
      %end;
   %end;
%end;
%end;

run;
%put &_ORCLP_;
%mend solve;
%solve
Example 5.1: Logic-Based Puzzles

Output 5.1.2 shows the solution.

Output 5.1.2  Solution of the Sudoku Grid

<table>
<thead>
<tr>
<th>9</th>
<th>8</th>
<th>5</th>
<th>3</th>
<th>2</th>
<th>7</th>
<th>6</th>
<th>4</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>8</td>
<td>9</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>8</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>9</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

Finally, substituting the following PROC CLP statement and invoking the CLP procedure finds all 5,299,405 solutions to the partially filled sudoku grid shown in Output 5.1.3:

```
proc clp out=mycas.outdata findall timetype=real;
```

Output 5.1.3  An Example of an Unsolved Sudoku Grid

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>6</td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The solution summaries shown in Output 5.1.4 and Output 5.1.5 correspond to solutions of the CSP in two different modes. The first solution summary corresponds to solving the CSP in single-machine mode, and the second solution summary corresponds to solving the CSP in distributed mode by using a grid that contains 142 nodes.
Pi Day Sudoku

The basic structure of the classical sudoku problem can easily be extended to formulate more complex puzzles. One such example is the Pi Day sudoku puzzle. Pi Day is a celebration of the number $\pi$ that occurs every March 14. In honor of Pi Day, Brainfreeze Puzzles (Riley and Taalman 2008) celebrated this day with a special 12×12 grid sudoku puzzle. The 2008 Pi Day sudoku puzzle is shown in Output 5.1.6.

Output 5.1.4  Single-Machine Mode

The CLP Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Variable Selection</td>
</tr>
<tr>
<td>Value Selection</td>
</tr>
</tbody>
</table>

Solutions Found  5299405
Solution Time    704.775119

Output 5.1.5  Distributed Mode

The CLP Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Variable Selection</td>
</tr>
<tr>
<td>Value Selection</td>
</tr>
</tbody>
</table>

Solutions Found  5299405
Solution Time    47.156476

Output 5.1.6  Pi Day Sudoku 2008

```
3 1 5 4 1 9 5
1 3 1 3 6

4 3 8 2

5 1 9 2 5 1

9 5 5

5 8 1 9 3 6

5 8 2 5 5 3

5 6 1

2 5 1 5 5 9

6 4 1 3

1 5 1 5 5

5 5 4 3 1 6 8
```
The rules for this puzzle are a little different from the rules for standard sudoku:

1. Rather than using regular $3 \times 3$ blocks, this puzzle uses jigsaw regions such that highlighted regions in the middle resemble the Greek letter $\pi$. Each jigsaw region consists of 12 contiguous cells.

2. The first 12 digits of $\pi$ are used instead of the digits 1–9. Each row, column, and jigsaw region contains the first 12 digits of $\pi$ (314159265358) in some order. In other words, there are no 7s; one each of 2, 4, 6, 8, and 9; two each of 1 and 3; and three 5s.

The data set raw contains the partially filled values for the grid and, similar to the sudoku problem, is used to create the set of macro variables $C_{ij}$ ($i = 1, \ldots, 12, j = 1, \ldots, 12$) where $C_{ij}$ is the value of cell $(i, j)$ in the grid when specified, and missing otherwise.

```sas
data raw;
  input C1-C12;
datalines;
3 . . 1 5 4 . . 1 . 9 5
. 1 . . 3 . . . 1 3 6
. . 4 . . 3 . 8 . . 2 .
5 . . 1 . 9 2 5 . . 1
. 9 . . 5 . . 5 . . .
5 8 1 . . 9 . . 3 . 6 .
. 5 . 8 . . 2 . . 5 5 3
. . . 5 . . 6 . . 1 .
2 . . 5 1 5 . . 9 .
. 6 . . 4 . 1 . . 3 .
1 5 1 . . . 5 . . 5 .
5 5 . 4 . . 3 1 6 . . 8
;
%macro cdata;
  /* store each pre-filled value into macro variable C_i_j */
data _null_; set raw;
  array C{12};
  do j = 1 to 12;
    i = _N_;
    call symput(compress('C_'||put(i,best.)||'_'||put(j,best.)), put(C[j],best.));
  end;
run;
%mend cdata;
%cdata;
```

As in the sudoku problem, let the variable $X_{ij}$ represent the value of the cell that corresponds to row $i$ and column $j$. The domain of each of these variables is $[1, 9]$.

For each row, column, and jigsaw region, a GCC statement is specified to enforce the condition that it contain exactly the first twelve digits of $\pi$.

In particular, the variables in row $r$, $r = 1, \ldots, 12$ are $X_{r1}, \ldots, X_{r12}$. The SAS macro `%CONS_ROW(R)` enforces the GCC constraint that row $r$ contain exactly two 1s, two 3s, three 5s, no 7s, and one of each of the other values:
Chapter 5: The CLP Procedure

%macro cons_row(r);
    /* Row r must contain two 1's, two 3's, three 5's, no 7's, */
    /* and one for each of other values from 1 to 9. */
    gcc(X_&r._1-X_&r._12) =
        ( (1, 2, 2) (3, 2, 2) (5, 3, 3) (7, 0, 0) DL=1 DU=1 );
%mend cons_row;

The variables in column c are $X_{1c}, \ldots, X_{12c}$. The SAS macro %CONS_COL(C) enforces a similar GCC constraint for each column c.

%macro cons_col(c);
    /* Column c must contain two 1's, two 3's, three 5's, */
    /* no 7's, and one for each of other values from 1 to 9. */
    gcc( %do r = 1 %to 12;
        X_&r._&c.
    %end; ) = ((1, 2, 2) (3, 2, 2) (5, 3, 3) (7, 0, 0) DL=1 DU=1);
%mend cons_col;

Generalizing this concept further, the SAS macro %CONS_REGION(VARS) enforces the GCC constraint for the jigsaw region that is defined by the macro variable VARS.

%macro cons_region(vars);
    /* Jigsaw region that contains &vars must contain two 1's, */
    /* two 3's, three 5's, no 7's, and one for each of other */
    /* values from 1 to 9. */
    gcc(&vars.) = ((1, 2, 2) (3, 2, 2) (5, 3, 3) (7, 0, 0) DL=1 DU=1);
%mend cons_region;

The following SAS statements incorporate the preceding macros to define the GCC constraints in order to find all solutions of the Pi Day sudoku 2008 puzzle:

%macro pds(fn=mycas.pdsout,solns=allsolns,varsel=MINR,maxtime=900);
    proc clp out=&fn &solns
        varselect=&varsel /* Variable selection strategy */
        maxtime=&maxtime; /* Time limit */
        /* Variable $X_{i,j}$ represents the grid of ith row and jth column. */
        var ( %do i = 1 %to 12;
            X_&i._1 - X_&i._12 %end; ) = [1,9];

        /* $X_{i,j} = C_{i,j}$ if $C_{i,j}$ is non-missing */
        %do i = 1 %to 12;
            %do j = 1 %to 12;
                %if &&C_&i._&j ne . %then %do;
                    lincon X_&i._&j = &&C_&i._&j;
                %end;
            %end;
        %end;
%mend pds;
Example 5.1: Logic-Based Puzzles

/* 12 Row constraints: */
%do r = 1 %to 12;
   %cons_row(&r);
%end;

/* 12 Column constraints: */
%do c = 1 %to 12;
   %cons_col(&c);
%end;

/* 12 Jigsaw region constraints: */
/* Each jigsaw region is defined by the macro variable &vars. */

/* Region 1: */
%let vars = X_1_1 - X_1_3 X_2_1 - X_2_3
         X_3_1 X_3_2 X_4_1 X_4_2 X_5_1 X_5_2;
%cons_region(&vars.);

/* Region 2: */
%let vars = X_1_4 - X_1_9 X_2_4 - X_2_9;
%cons_region(&vars.);

/* Region 3: */
%let vars = X_1_10 - X_1_12 X_2_10 - X_2_12
         X_3_11 X_3_12 X_4_11 X_4_12 X_5_11 X_5_12;
%cons_region(&vars.);

/* Region 4: */
%let vars = X_3_3 - X_3_6 X_4_3 - X_4_6 X_5_3 - X_5_6;
%cons_region(&vars.);

/* Region 5: */
%let vars = X_3_7 - X_3_10 X_4_7 - X_4_10 X_5_7 - X_5_10;
%cons_region(&vars.);

/* Region 6: */
%let vars = X_6_1 - X_6_3 X_7_1 - X_7_3
         X_8_1 - X_8_3 X_9_1 - X_9_3;
%cons_region(&vars.);

/* Region 7: */
%let vars = X_6_4 X_6_5 X_7_4 X_7_5 X_8_4 X_8_5
         X_9_4 X_9_5 X_10_4 X_10_5 X_11_4 X_11_5;
%cons_region(&vars.);

/* Region 8: */
%let vars = X_6_6 X_6_7 X_7_6 X_7_7 X_8_6 X_8_7
         X_9_6 X_9_7 X_10_6 X_10_7 X_11_6 X_11_7;
%cons_region(&vars.);

/* Region 9: */
%let vars = X_6_8 X_6_9 X_7_8 X_7_9 X_8_8 X_8_9
         X_9_8 X_9_9 X_10_8 X_10_9 X_11_8 X_11_9;
%cons_region(&vars.);
The only solution of the 2008 Pi Day sudoku puzzle is shown in Output 5.1.7.

**Output 5.1.7**  Solution to Pi Day Sudoku 2008

**Pi Day Sudoku  2008**

<table>
<thead>
<tr>
<th>Obs</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
<th>C7</th>
<th>C8</th>
<th>C9</th>
<th>C10</th>
<th>C11</th>
<th>C12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>8</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>9</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>5</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>2</td>
<td>9</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>5</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

The corresponding completed grid is shown in Output 5.1.8.
Output 5.1.8  Solution to Pi Day Sudoku 2008

Next, a variation of this problem illustrates the use of the EVALVARSEL= option to help determine an appropriate selection strategy. Consider the same Pi Day sudoku puzzle with a few more missing cells, as shown in Output 5.1.9. The objective is to find all the possible solutions, subject to the same constraints as before.

Output 5.1.9  Pi Day Sudoku 2008

When you specify the EVALVARSEL= option without including a keyword list, the CLP procedure evaluates each available variable selection strategy for the amount of time that you specify in the MAXTIME= option. When you are running in distributed mode, in which there are at least as many machines as strategies, the CSPs are solved concurrently, with each selection strategy being evaluated on a different machine. The EvalInfo ODS table, which is shown in Output 5.1.10, identifies the variable selection strategies that find a
solution within a second in distributed mode on a 10-machine network. This information can also be found in the _ORCLPEVS_ macro variable.

**Output 5.1.10** EvalInfo ODS Table

<table>
<thead>
<tr>
<th>The CLP Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variable Selection Strategy Evaluation</strong></td>
</tr>
<tr>
<td><strong>Selection Strategy</strong></td>
</tr>
<tr>
<td>MINR</td>
</tr>
<tr>
<td>FIFO</td>
</tr>
<tr>
<td>MAXC</td>
</tr>
<tr>
<td>MAXCS</td>
</tr>
<tr>
<td>MINRMAXC</td>
</tr>
<tr>
<td>WDEG</td>
</tr>
<tr>
<td>DOMDDEG</td>
</tr>
<tr>
<td>DOMWDEG</td>
</tr>
<tr>
<td>DOMDEG</td>
</tr>
</tbody>
</table>

Finally, the CLP procedure is invoked in distributed mode by using the FINDALL option and the WDEG selection strategy. The resulting solution summary, which is shown in **Output 5.1.11**, indicates that all 5,361 solutions have been found, along with the solution time.

**Output 5.1.11** WDEG Selection Strategy

<table>
<thead>
<tr>
<th>The CLP Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solution Summary</strong></td>
</tr>
<tr>
<td><strong>Solution Status</strong></td>
</tr>
<tr>
<td><strong>Variable Selection</strong></td>
</tr>
<tr>
<td><strong>Value Selection</strong></td>
</tr>
<tr>
<td><strong>Solutions Found</strong></td>
</tr>
<tr>
<td><strong>Solution Time</strong></td>
</tr>
</tbody>
</table>

**Example 5.2: Car Painting Problem**

The car painting process is an important part of the automobile manufacturing industry. Purging (the act of changing colors during assembly) is expensive because of the added cost of wasted paint and solvents from each color change and the extra time that the purging process requires. The objective of the car painting problem is to sequence the cars in the assembly line in order to minimize the number of paint color changes (Sokol 2002; Trick 2004).

Suppose an assembly line contains 10 cars, which are ordered 1, 2, . . . , 10. A car must be painted within three positions of its assembly order. For example, car 5 can be painted in positions 2 through 8. Cars 1, 5, and 9 are red; 2, 6, and 10 are blue; 3 and 7 green; and 4 and 8 are yellow. The initial sequence 1, 2, . . . , 10 corresponds to the color pattern RBGYRBGYRB and has nine purgings. The objective is to find a solution that minimizes the number of purgings.

This problem can be formulated as a CSP as follows:
Example 5.2: Car Painting Problem

The variables $S_i$ and $C_i$ represent the ID and color, respectively, of the car in slot $i$.

- An element constraint relates the car ID to its color.
- An alldifferent constraint ensures that every slot is associated with a unique car ID.
- Two linear constraints represent the constraint that a car must be painted within three positions of its assembly order.
- The binary variable $P_i$ indicates whether a paint purge takes place after the car in slot $i$ is painted.
- A linear constraint is used to limit the total number of purgings to the required number.

The following %CAR_PAINTING macro determines all feasible solutions for a given number of purgings, which is specified as a parameter to the macro:

```plaintext
%macro car_painting(purgings);
proc clp out=mycas.car_ds findall;
  %do i = 1 %to 10;
    var S&i = [1, 10]; /* which car is in slot &i.*/
    var C&i = [1, 4]; /* which color the car in slot &i is.*/
    /* Red=1; Blue=2; Green=3; Yellow=4 */
    element (S&i, (1, 2, 3, 4, 1, 2, 3, 4, 1, 2), C&i);
  %end;

  /* A car can be painted only once. */
  alldiff (S1-S10);

  /* A car must be painted within 3 positions of its assembly order. */
  %do i = 1 %to 10;
    lincon S&i-&i>=-3;
    lincon S&i-&i<=3;
  %end;

  %do i = 1 %to 9;
    var P&i = [0, 1]; /* Whether there is a purge after slot &i*/
    reify P&i: (C&i <> C%eval(&i+1));
  %end;

  /* Calculate the number of purgings. */
  lincon 0
  %do i = 1 %to 9;
    + P&i
  %end;
  <=&purgings ;
run;
%mend;
%car_painting(5)
```
The problem is infeasible for four purgings. The CLP procedure finds 87 possible solutions for the five-purgings problem. The solutions are sorted by the total distance all cars are moved in the sequencing, which ranges from 12 to 22 slots. The first 15 solutions are displayed in the Gantt chart in Output 5.2.1. Each row represents a solution, and each color transition represents a paint purging.

**Output 5.2.1** Car Painting Schedule with Five Purgings

<table>
<thead>
<tr>
<th>Solution</th>
<th>Distance</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
</table>
Example 5.3: Scene Allocation Problem

The scene allocation problem consists of deciding when to shoot each scene of a movie in order to minimize the total production cost (Van Hentenryck 2002). Each scene involves a number of actors, and at most five scenes a day can be shot. All actors who appear in a scene must be present in the studio on the day the scene is shot. Each actor earns a daily rate for each day spent in the studio, regardless of the number of scenes in which he or she appears on that day. The goal is to shoot the movie for the lowest possible production cost.

The actors’ names, their daily fees, and the scenes in which they appear are contained in the SCENE data set, which is shown in Output 5.3.1. The data table variables S_Var1, . . . , S_Var9 indicate the scenes in which the actor appears. For example, the first observation indicates that Patt’s daily fee is 26,481 and that Patt appears in scenes 2, 5, 7, 10, 11, 13, 15, and 17.

Output 5.3.1 The Scene Data Set

<table>
<thead>
<tr>
<th>Obs</th>
<th>Number</th>
<th>Actor</th>
<th>DailyFee</th>
<th>S_Var1</th>
<th>S_Var2</th>
<th>S_Var3</th>
<th>S_Var4</th>
<th>S_Var5</th>
<th>S_Var6</th>
<th>S_Var7</th>
<th>S_Var8</th>
<th>S_Var9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Patt</td>
<td>26481</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>17</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Casta</td>
<td>25043</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>13</td>
<td>16</td>
<td>19</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Scolar</td>
<td>30310</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>10</td>
<td>14</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Murphy</td>
<td>4085</td>
<td>2</td>
<td>8</td>
<td>12</td>
<td>13</td>
<td>15</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>Brown</td>
<td>7562</td>
<td>2</td>
<td>3</td>
<td>12</td>
<td>17</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>Hacket</td>
<td>9381</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>13</td>
<td>18</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>Anderson</td>
<td>8770</td>
<td>5</td>
<td>6</td>
<td>14</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>McDougal</td>
<td>5788</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>15</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>Mercer</td>
<td>7423</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>16</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>Spring</td>
<td>3303</td>
<td>5</td>
<td>6</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>Thompson</td>
<td>9593</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

There are 19 scenes. At most five scenes can be filmed in one day, so at least four days are needed to schedule all the scenes ($\lceil \frac{19}{5} \rceil = 4$). Let $S_j . k$ be a binary variable that equals 1 if scene $j$ is shot on day $k$. Let $A_i . k$ be another binary variable that equals 1 if actor $i$ is present on day $k$. The variable $Name_i$ is the name of the $i$th actor; $Cost_i$ is the daily cost of the $i$th actor. $A_i . S_j = 1$ if actor $i$ appears in scene $j$, and 0 otherwise.

The objective function representing the total production cost is given by

$$\min \sum_{i=1}^{11} \sum_{k=1}^{4} Cost_i \times A_i . k$$

The %SCENE macro first reads the data set scene and produces three sets of macro variables: $Name_i$, $Cost_i$, and $A_i . S_j$. The data set cost is created next to specify the objective function. Finally, the CLP procedure is invoked. There are two sets of GCC constraints in the CLP call: one to make sure each scene is shot exactly once, and one to limit the number of scenes shot per day to be at least four and at most five. There are two sets of LINCON constraints: one to indicate that an actor must be present if any of his or her scenes are shot that day, and one for breaking symmetries to improve efficiency. Additionally, an OBJ statement specifies upper and lower bounds on the objective function to be minimized.
%macro scene;
/* Ai_Sj=1 if actor i appears in scene j */
/* Ai_Sj=0 otherwise */
/* Initialize to 0 */
%do i=1 %to 11; /* 11 actors */
  %do j=1 %to 19; /* 19 scenes */
    %let A&i._S&j=0;
  %end;
%end;

data scene_cost;
set scene;
keep DailyFee A;
retain DailyFee A;
do day=1 to 4;
  A='A'||left(strip(_N_))||'_'||left(strip(day));
  output;
end;
call symput("Name"||strip(_n_), Actor); /* read actor name */
call symput("Cost"||strip(_n_), DailyFee); /* read actor cost */
/* read whether an actor appears in a scene */
%do i=1 %to 9; /* 9 scene variables in the data set */
  if S_Var&i > 0 then
    call symput("A"||strip(_n_)||"_S"||strip(S_Var&i), 1);
%end;
run;

/* Create constraint data table which defines the objective function */
proc transpose data=scene_cost out=cost(drop=_name_);
  var DailyFee;
  id A;
run;
data mycas.cost;
set cost;
_TYPE_='MIN';
_RHS_=.;
run;

/* Find the minimum objective value. */
proc clp condata=mycas.cost out=mycas.out varselect=maxc;
  /* Set lower and upper bounds for the objective value */
  /* Lower bound: every actor appears on one day. */
  /* Upper bound: every actor appears on all four days. */
  obj lb=137739 ub=550956;

  /* Declare variables. */
  %do k=1 %to 4; /* 4 days */
    %do j=1 %to 19; /* 19 scenes */
      var S&j._&k=[0,1]; /* Indicates if scene j is shot on day k. */
    %end;
  %do i=1 %to 11; /* 11 actors */
    var A&i._&k=[0,1]; /* Indicates if actor i is present on day k. */
  %end;
%end;
Example 5.3: Scene Allocation Problem

Example 5.3: Scene Allocation Problem

/* Every scene is shot exactly once. */
%do j=1 %to 19;
    gcc (
        %do k=1 %to 4;
            S&j._&k
        %end;
    )=( (1, 1, 1) );
%end;

/* At least 4 and at most 5 scenes are shot per day. */
%do k=1 %to 4;
    gcc (
        %do j=1 %to 19;
            S&j._&k
        %end;
    )=( (1, 4, 5) );
%end;

/* Actors for a scene must be present on day of shooting. */
%do k=1 %to 4;
    %do j=1 %to 19;
        %do i=1 %to 11;
            %if (&&A&i._S&j>0) %then %do;
                lincon S&j._&k <= A&i._&k;
            %end;
        %end;
    %end;
%end;

/* Symmetry breaking constraints. Without loss of any generality, we */
/* can assume Scene1 to be shot on day 1, Scene2 to be shot on day 1 or */
/* or day 2, and Scene3 to be shot on either day 1, day 2 or day 3. */
lincon S1_1 = 1, S1_2 = 0, S1_3 = 0, S1_4 = 0,
    S2_3 = 0, S2_4 = 0, S3_4 = 0;

/* If Scene2 is shot on day 1, */
/* then Scene3 can be shot on day 1 or day 2. */
lincon S2_1 + S3_3 <= 1;
run;
%put &_ORCLP_;
%mend scene;

The optimal production cost is 334,144, as reported in the _ORCLP_ macro variable. The corresponding actor schedules and scene schedules are displayed in Output 5.3.2 and Output 5.3.3, respectively.
### Output 5.3.2 Scene Allocation Problem: Actor Schedules

#### Scene Allocation Problem

Optimal Schedule by Actor

<table>
<thead>
<tr>
<th>Actor</th>
<th>Day 1 Scenes</th>
<th>Day 2 Scenes</th>
<th>Day 3 Scenes</th>
<th>Day 4 Scenes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 6 14 18</td>
<td>2 5 8 12 15</td>
<td>3 9 10 16 17</td>
<td>4 7 11 13 19</td>
</tr>
<tr>
<td>Patt</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Casta</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scolaro</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Murphy</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Brown</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hacket</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Anderson</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>McDougal</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mercer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Spring</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thompson</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example 5.4: Car Sequencing Problem

This example is an instance of a category of problems known as the car sequencing problem. There is a considerable amount of literature related to this problem (Dincbas, Simonis, and Van Hentenryck 1988; Gravel, Gagne, and Price 2005; Solnon et al. 2008).

A number of cars are to be produced on an assembly line. Each car is customized to include a specific set of options, such as air conditioning, a sunroof, a navigation system, and so on. The assembly line moves through several workstations for installation of these options. The cars cannot be positioned randomly, because each workstation has limited capacity and needs time to set itself up to install the options as the car moves in front of the station. These capacity constraints are formalized using constraints of the form \( m \) out of \( N \), which indicates that the workstation can install the option on \( m \) out of every sequence of \( N \) cars. The car sequencing problem is to determine a sequencing of the cars on the assembly line that satisfies the demand constraints for each set of car options and the capacity constraints for each workstation.

This example comes from Dincbas, Simonis, and Van Hentenryck (1988). Ten cars need to be customized with five possible options. A class of car is defined by a specific set of options; there are six classes of cars. The instance data are presented in Table 5.5.
Table 5.5  The Instance Data

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Capacity m/N</th>
<th>Car Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option 1</td>
<td>1/2</td>
<td>1 0 0 1 1</td>
</tr>
<tr>
<td>Option 2</td>
<td>2/3</td>
<td>0 0 1 1 0</td>
</tr>
<tr>
<td>Option 3</td>
<td>1/3</td>
<td>1 0 0 1 0</td>
</tr>
<tr>
<td>Option 4</td>
<td>2/5</td>
<td>1 1 0 1 0</td>
</tr>
<tr>
<td>Option 5</td>
<td>1/5</td>
<td>0 0 1 0 0</td>
</tr>
</tbody>
</table>

Number of Cars 1 1 2 2 2 2

For example, car class 4 requires installation of option 2 and option 4, and two cars of this class are required. The workstation for option 2 can process only two out of every sequence of three cars. The workstation for option 4 has even less capacity—two out of every five cars.

The data for this problem are used to create a SAS data set, which in turn is processed to generate the SAS macro variables shown in Table 5.6 that are used in the CLP procedure. The assembly line is treated as a sequence of slots, and each car must be allocated to a single slot.

Table 5.6  SAS Macro Variables

<table>
<thead>
<tr>
<th>Macro Variable</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ncars</td>
<td>Number of cars (slots)</td>
<td>10</td>
</tr>
<tr>
<td>Nops</td>
<td>Number of options</td>
<td>5</td>
</tr>
<tr>
<td>Nclss</td>
<td>Number of classes</td>
<td>6</td>
</tr>
<tr>
<td>Max_1–Max_5</td>
<td>For each option, the maximum number of cars with</td>
<td>1 2 1 2 1</td>
</tr>
<tr>
<td></td>
<td>that option in a block</td>
<td></td>
</tr>
<tr>
<td>Blsz_1–Blsz_5</td>
<td>For each option, the block size to which the</td>
<td>2 3 3 5 5</td>
</tr>
<tr>
<td></td>
<td>maximum number refers</td>
<td></td>
</tr>
<tr>
<td>class_1–class_6</td>
<td>Index number of each class</td>
<td>1 2 3 4 5 6</td>
</tr>
<tr>
<td>cars_cls_1–cars_cls_6</td>
<td>Number of cars in each class</td>
<td>1 1 2 2 2 2</td>
</tr>
<tr>
<td>list_1–list_5</td>
<td>Class indicator list for each option; for</td>
<td>list_1=1,0,0,0,1,1</td>
</tr>
<tr>
<td></td>
<td>example, classes 1, 5, and 6 that require option</td>
<td>list_2=0,0,1,1,0,1</td>
</tr>
<tr>
<td></td>
<td>1 (list_1)</td>
<td>list_3=1,0,0,0,1,0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>list_4=1,1,0,1,0,0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>list_5=0,0,1,0,0,0</td>
</tr>
<tr>
<td>cars_opts_1–cars_opts_5</td>
<td>Number of cars for each option</td>
<td>5 6 3 4 2</td>
</tr>
<tr>
<td></td>
<td>(cars_opts_1 represents the number of cars that</td>
<td></td>
</tr>
<tr>
<td></td>
<td>require option 1)</td>
<td></td>
</tr>
</tbody>
</table>

The decision variables for this problem are shown in Table 5.7.
Table 5.7 The Decision Variables

<table>
<thead>
<tr>
<th>Variable Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_{1}$–$S_{10}$=[1,6]</td>
<td>$S_i$ is the class of cars assigned to slot $i$.</td>
</tr>
<tr>
<td>$O_{1_{1}}$–$O_{1_{5}}$=[0,1]</td>
<td>$O_{i_{j}}$=1 if the class assigned to slot $i$ needs option $j$.</td>
</tr>
</tbody>
</table>

The following SAS statements express the workstation capacity constraints by using a set of linear constraints for each workstation. A single GCC constraint expresses the demand constraints for each car class. An element constraint for each option variable expresses the relationships between slot variables and option variables.

Finally, a set of redundant constraints are introduced to improve the efficiency of propagation. The idea behind the redundant constraint in this model is the following realization: if the workstation for option $j$ has capacity $r$ out of $s$, then at most $r$ cars in the sequence $(n - s + 1), \ldots, n$ can have option $j$, where $n$ is the total number of cars. Consequently, at least $n_j - r$ cars in the sequence $1, \ldots, n - s$ must have option $j$, where $n_j$ is the number of cars that have option $j$. Generalizing this further, at least $n_j - k \times r$ cars in the sequence $1, \ldots, (n - k \times s)$ must have option $j$, $k = 1, \ldots, \lceil n/s \rceil$.

```sas
%macro car_sequencing(outdata);

proc clp out=&outdata varselect=minrmaxc findall;

   /* Declare Variables */
   var
      /* Slot variables: $S_i$ - class of car assigned to Slot $i$ */
      %do i = 1 %to &Ncars;
         $S_{&i} = [1, &Nclss]
      %end;

      /* Option variables: $O_{i_{j}}$ - indicates if class assigned to Slot $i$ needs Option $j$ */
      %do i = 1 %to &Ncars;
         %do j = 1 %to &Nops;
            $O_{&i_{j}} = [0, 1]
         %end;
      %end;

   /* Capacity Constraints: for each option $j$ */
   /* Installed in at most $Max_{j}$ cars out of every sequence of $BlSz_{j}$ cars */
   %do j = 1 %to &Nops;
      %do i = 0 %to %eval(&Ncars-&BlSz_{&j});
         lincon 0
      %end;
      %do k=1 %to &BlSz_{&j};
         + O_&i_{&i+k_{j}}
      %end;
      <=&Max_{&j};

%end;
```

%end;
%end;

/* Demand Constraints: for each class i */
/* Exactly cars_cls_i cars */
gcc (S_1-S_&Ncars) = ( 
  %do i = 1 %to &Nclss;
    (&&class_&i, &&cars_cls_&i, &&cars_cls_&i)
  %end;
); 

/* Element Constraints: For each slot i and each option j */
/* relate the slot variable to the option variables. */
/* O_i.j is the S_i th element in list_j. */
%do i = 1 %to &Ncars;
  %do j = 1 %to &Nops;
    element (S_&i, (&&list_&j), O_&i._&j);
  %end;
%end;

/* Redundant Constraints to improve efficiency - for every */
/* option j. */
/* At most &&Max_&j out of every sequence of &&BlSz_&j cars */
/* requires option j. */
/* All the other slots contain at least cars_opt_&j - Max_&j */
/* cars with option j */
%do j = 1 %to &Nops;
  %do i = 1 %to %eval(&Ncars/&&BlSz_&j);
    lincon 0 
    %do k=1 %to %eval(&Ncars-&i*&&BlSz_&j);
      + O_&k._&j
    %end;
    >= %eval(&&cars_opts_&j-&i*&&Max_&j);
  %end;
%end;
run;
%mend;
%car_sequencing(mycas.sequence_out);

This problem has six solutions, as shown in Output 5.4.1.
Example 5.4: Car Sequencing Problem

Output 5.4.1 Car Sequencing

Table 5.8 Instance Data for 100 Cars, 5 Options, 18 Classes

Although constraints can usually help performance, this is not necessarily always the case. Consider the instance shown in Table 5.8 with 100 cars, 5 options, and 18 classes from Dincbas, Simonis, and Van Hentenryck (1988).

Using the same formulation as before, the CLP procedure was unable to find a solution in 10 seconds. However, removing the redundant constraints resulted in a solution being found almost immediately.

Two different views of the solution that was found are shown in Output 5.4.2 and Output 5.4.3. The horizontal axis in each chart represents the assembly sequence. The assembly sequence by class is shown in Output 5.4.2. For example, a Class 1 car is in slots 1, 11, 31, 59, and 79.
Output 5.4.2  Car Sequencing by Class

The assembly sequence by option is shown in Output 5.4.3. You can use this chart to quickly verify that the capacity constraints are satisfied. For example, from the corresponding block patterns in each row, it is evident that option 1 is never processed on consecutive cars, option 2 is never processed on three consecutive cars, and so on.
Finally, the two solution summaries shown in Output 5.4.4 and Output 5.4.5 contrast the number of solutions that are found in two different modes in the same time period, using the FINDALL option.

**Output 5.4.4**  Single-Machine Mode

<table>
<thead>
<tr>
<th>Car Sequencing Problem</th>
<th>Single-Machine Mode</th>
</tr>
</thead>
</table>

The CLP Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution Status</td>
<td>Time Limit Reached</td>
</tr>
<tr>
<td>Variable Selection</td>
<td>WDEG</td>
</tr>
<tr>
<td>Value Selection</td>
<td>MIN</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solutions Found</th>
<th>878</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution Time</td>
<td>10.000292</td>
</tr>
</tbody>
</table>
This example illustrates modeling and solving a well-known double round-robin problem using the CLP procedure. Round-robin tournaments (and variations of them) are a popular format in the scheduling of many sports league tournaments. In a single round-robin tournament, each team plays every other team just once. In a double round-robin (DRR) tournament, each team plays every other team twice: once at home and once away.

This example is based on the work of Nemhauser and Trick (1998) and deals with scheduling the Atlantic Coast Conference (ACC) Men’s Basketball games for the 1997–1998 season.

A temporally dense double round-robin (DDRR) for \( n \) teams is a double round-robin in which the \( n(n - 1) \) games are played over a minimal number of dates or time slots. If \( n \) is even, the number of slots is \( 2(n - 1) \) and each team plays in every time slot. If \( n \) is odd, the number of slots is \( 2n \) and \( (n - 1) \) teams play in each time slot. In the latter case, each time slot has a team with a bye, and each team has two byes for the season.

The Atlantic Coast Conference (ACC) 1997–1998 men’s basketball scheduling problem as described in Nemhauser and Trick (1998) and Henz (2001) is a DRRR that consists of the following nine teams with their abbreviated team name and team number shown in parentheses: Clemson (Clem 1), Duke (Duke 2), Florida State (FSU 3), Georgia Tech (GT 4), Maryland (UMD 5), University of North Carolina (UNC 6), NC State (NCSU 7), Virginia (UVA 8), and Wake Forest (Wake 9).

The general objective is to schedule the DRRR to span the months of January and February and possibly include a game in December or March or both. In general, each team plays twice a week—typically Wednesday and Saturday. Although the actual day might differ, these two time slots are referred to as the “weekday slot” and the “weekend slot.” Since there are an odd number of teams, there is a team with a bye in each slot and four games in each slot, resulting in a schedule that requires 18 time slots or nine weeks. The last time slot must be a weekend slot, which implies the first slot is a weekday slot. The first slot, denoted slot 1, corresponds to the last weekday slot of December 1997, and the final slot, slot 18, corresponds to the first weekend slot of March 1998. Each team plays eight home games and eight away games, and has two byes.

In addition several other constraints must be satisfied. This example uses the following criteria employed by Nemhauser and Trick (1998) as presented by Henz (2001).

1. **Mirroring**: The dates are grouped into pairs \((r1, r2)\), such that each team gets to play against the same team in dates \(r1\) and \(r2\). Such a grouping is called a mirroring scheme. A separation of nine
slots can be achieved by mirroring a round-robin schedule; although this separation is desirable, it is not possible for this problem.

Nemhauser and Trick fix the mirroring scheme to

$$m = (1, 8), (2, 9), (3, 12), (4, 13), (5, 14), (6, 15), (7, 16), (10, 17), (11, 18)$$

in order to satisfy the constraints that UNC and Duke play in time slots 11 and 18. (See criterion 9.)

2. **Initial and final home and away games:** Every team must play at home on at least one of the first three dates. Every team must play at home on at least one of the last three dates. No team can play away on both of the last two dates.

3. **Home/away/bye pattern:** No team can have more than two away games in a row. No team can have more than two home games in a row. No team can have more than three away games or byes in a row. No team can have more than four home games or byes in a row.

4. **Weekend pattern:** Of the nine weekends, each team plays four at home, four away, and has one bye.

5. **First weekends:** Each team must have home games or byes on at least two of the first five weekends.

6. **Rival matches:** Every team except FSU has a traditional rival. The rival pairs are Clem-GT, Duke-UNC, UMD-UVA, and NCSU-Wake. In the last date, every team except FSU plays against its rival, unless it plays against FSU or has a bye.

7. **Popular matches in February:** The following pairings must occur at least once in dates 11 to 18: Duke-GT, Duke-Wake, GT-UNC, UNC-Wake.

8. **Opponent sequence:** No team plays in two consecutive away dates against Duke and UNC. No team plays in three consecutive dates against Duke, UNC, and Wake (independent of the home or away status).

9. **Idiosyncrasies:** UNC plays its rival Duke in the last date and in date 11. UNC plays Clem in the second date. Duke has a bye in date 16. Wake does not play home in date 17. Wake has a bye in the first date. Clem, Duke, UMD and Wake do not play away in the last date. Clem, FSU, and GT do not play away in the first date. Neither FSU nor NCSU has a bye in last date. UNC does not have a bye in the first date.

Previous work for solving round-robin problems, including that of Nemhauser and Trick (1998) and Henz (2001), have used a general three-phase framework for finding good schedules.

1. pattern generation

2. pattern set generation

3. timetable generation

A pattern is a valid sequence of home, away, and bye games for a given team for the entire season. For example, the following is a valid pattern:

```
A H B A H A H A A H B H A A H H A
```

For this example, patterns that satisfy criterion 1 through criterion 5 and some constraints in criterion 9 are generated using the CLP procedure with the SAS macro `%PATTERNS`. 
Chapter 5: The CLP Procedure

/*****************************************************************
/* First, find all possible patterns. Consider only time */
/* constraints at this point. A pattern should be suitable */
/* for any team. Do not consider individual teams yet. */
/*******************************************************************/
%macro patterns();
proc clp out=mycas.all_patterns findall;
   %do j = 1 %to 18;
      var h&j = [0, 1]; /* home */
      var a&j = [0, 1]; /* away */
      var b&j = [0, 1]; /* bye */
     /* A team is either home, away, or bye. */
      lincon h&j + a&j + b&j=1;
   %end;
   /* For date 1 to 18. */
   %do j = 1 %to 18;
      var h&j = [0, 1]; /* home */
      var a&j = [0, 1]; /* away */
      var b&j = [0, 1]; /* bye */
      /* A team is either home, away, or bye. */
      lincon h&j + a&j + b&j=1;
   %end;
   /* Criterion 1 - Mirroring Scheme */
   /* The dates are grouped into pairs (j, j1), such that each */
   /* team plays the same opponent on dates j and j1. */
   /* A home game on date j will be an away game on date j1 */
   %do j = 1 %to 16;
      %do j1 = %eval(&j+1) %to 18;
         %if ( &j=1 and &j1=8 ) or ( &j=2 and &j1=9 ) or
            ( &j=3 and &j1=12 ) or ( &j=4 and &j1=13 ) or
            ( &j=5 and &j1=14 ) or ( &j=6 and &j1=15 ) or
            ( &j=7 and &j1=16 ) or ( &j=10 and &j1=17 ) or
            ( &j=11 and &j1=18 ) %then
            lincon h&j = a&j1, a&j = h&j1, b&j = b&j1;
      %end;
    %end;
   /* Criterion 2 - Initial and Final Home and Away Games */
   /* Every team must play home on at least one of the first three dates. */
   lincon h1 + h2 + h3 >= 1;
   /* Every team must play home on at least one of the last three dates. */
   lincon h16 + h17 + h18 >= 1;
   /* No team can play away on both last two dates. */
   lincon a17 + a18 < 2;
   /* Criterion 3 - Home/Away/Bye Pattern */
   %do j = 1 %to 16;
      /* No team can have more than two away matches in a row. */
      lincon a&j + a%eval(&j+1) + a%eval(&j+2) < 3;
%end;
/* No team can have more than two home matches in a row. */
lincon h&j + h%eval(&j+1) + h%eval(&j+2) < 3;
%end;

/* No team can have more than three away matches or byes in a row. */
%do j = 1 %to 15;
lincon a&j + b&j + a%eval(&j+1) + b%eval(&j+1) + a%eval(&j+2) + b%eval(&j+2) + a%eval(&j+3) + b%eval(&j+3) + h%eval(&j+4) + b%eval(&j+4) < 4;
%end;

/* No team can have more than four home matches or byes in a row. */
%do j = 1 %to 14;
lincon h&j + b&j + h%eval(&j+1) + h%eval(&j+1) + h%eval(&j+2) + b%eval(&j+2) + h%eval(&j+3) + h%eval(&j+3) + h%eval(&j+4) + b%eval(&j+4) < 5;
%end;

/*------------------------------------------------------------*/
/* Criterion 4 - Weekend Pattern */
/*------------------------------------------------------------*/
/* Each team plays four weekends at home. */
lincon 0 %do j = 2 %to 18 %by 2; +h&j %end; =4;
/* Each team plays four weekends away. */
lincon 0 %do j = 2 %to 18 %by 2; +a&j %end; =4;
/* Each team has 1 weekend with a bye */
lincon 0 %do j = 2 %to 18 %by 2; +b&j %end; =1;

/*------------------------------------------------------------*/
/* Criterion 5 - First Weekends */
/*------------------------------------------------------------*/
/* Each team must have home games or byes on at least two */
/* of the first five weekends. */
lincon 0 %do j = 2 %to 10 %by 2; +h&j + b&j %end; >=2;

/*------------------------------------------------------------*/
/* Criterion 9 - (Partial) */
/*------------------------------------------------------------*/
/* The team with a bye in date 1 does not play away on the */
/* last date or home in date 17 (Wake) */
/* The team with a bye in date 16 does not play away in */
/* date 18 (Duke) */
lincon b1 + a18 < 2, b1 + h17 < 2, b16 + a18 < 2;
run;
%mend;
%patterns;

The %PATTERNS macro generates 38 patterns. The next step is to find a subset of patterns with cardinality equal to the number of teams that would collectively support a potential assignment to all of the teams. For example, each of the 18 time slots must correspond to four home games, four away games, and one bye. Furthermore, pairs of patterns that do not support a potential meeting date between the two corresponding
teams are excluded. The following %PATTERN_SETS macro uses the CLP procedure with the preceding constraints to generate all possible pattern sets.

```sas
/* Determine all possible "pattern sets" considering only time constraints. */
/* Individual teams are not considered at this stage. */
/* xi - binary variable indicates pattern i is in pattern set */
*********************************************************************/
%macro pattern_sets();

/* Determine all possible "pattern sets" considering only time constraints. */
/* Individual teams are not considered at this stage. */
/* xi - binary variable indicates pattern i is in pattern set */
*********************************************************************/

%macro pattern_sets();

data all_patterns;
  set mycas.all_patterns;
  %do i=1 %to 38;
    if _n_=&i then do;
      %do j=1 %to 18;
        call symput("h&i._&j", put(h&j,best.));
        call symput("a&i._&j", put(a&j,best.));
        call symput("b&i._&j", put(b&j,best.));
      %end;
    end;
  %end;
run;

proc clp out=mycas.pattern_sets findall;
  /* xi=1 if pattern i belongs to pattern set */
  var (x1-x38)= [0, 1];

  /* Exactly nine patterns per patterns set */
  lincon 0 %do i = 1 %to 38; + x&i %end; =9;

  /* time slot constraints */
  %do j = 1 %to 18;
    /* Four home games per time slot */
    lincon 0 %do i = 1 %to 38; + &&h&i._&j*x&i %end; =4;
    /* Four away games per time slot */
    lincon 0 %do i = 1 %to 38; + &&a&i._&j*x&i %end; =4;
    /* One bye per time slot */
    lincon 0 %do i = 1 %to 38; + &&b&i._&j*x&i %end; =1;
  %end;

  /* Exclude pattern pairs that do not support a meeting date */
  %do i = 1 %to 38;
    %do i1 = %eval(&i+1) %to 38;
      %let count=0;
      %do j=1 %to 18;
        %if ( (&&h&i._&j=0 or &&a&i._&j=0) and
            (&&a&i._&j=0 or &&h&i._&j=0)) %then %do;
          %let count=%eval(&count+1);
        %end;
      %end;
    %end;
    %if (&count=18) %then %do;
      lincon x&i+x&i1<=1;
    %end;
  %end;
%end;
```

```sas
/* Determine all possible "pattern sets" considering only time constraints. */
/* Individual teams are not considered at this stage. */
/* xi - binary variable indicates pattern i is in pattern set */
*********************************************************************/
%macro pattern_sets();

data all_patterns;
  set mycas.all_patterns;
  %do i=1 %to 38;
    if _n_=&i then do;
      %do j=1 %to 18;
        call symput("h&i._&j", put(h&j,best.));
        call symput("a&i._&j", put(a&j,best.));
        call symput("b&i._&j", put(b&j,best.));
      %end;
    end;
  %end;
run;

proc clp out=mycas.pattern_sets findall;
  /* xi=1 if pattern i belongs to pattern set */
  var (x1-x38)= [0, 1];

  /* Exactly nine patterns per patterns set */
  lincon 0 %do i = 1 %to 38; + x&i %end; =9;

  /* time slot constraints */
  %do j = 1 %to 18;
    /* Four home games per time slot */
    lincon 0 %do i = 1 %to 38; + &&h&i._&j*x&i %end; =4;
    /* Four away games per time slot */
    lincon 0 %do i = 1 %to 38; + &&a&i._&j*x&i %end; =4;
    /* One bye per time slot */
    lincon 0 %do i = 1 %to 38; + &&b&i._&j*x&i %end; =1;
  %end;

  /* Exclude pattern pairs that do not support a meeting date */
  %do i = 1 %to 38;
    %do i1 = %eval(&i+1) %to 38;
      %let count=0;
      %do j=1 %to 18;
        %if ( (&&h&i._&j=0 or &&a&i._&j=0) and
            (&&a&i._&j=0 or &&h&i._&j=0)) %then %do;
          %let count=%eval(&count+1);
        %end;
      %end;
    %end;
    %if (&count=18) %then %do;
      lincon x&i+x&i1<=1;
    %end;
  %end;
```
The `%PATTERN_SETS` macro generates 17 pattern sets. The final step is to add the individual team constraints and match up teams to the pattern sets in order to come up with a schedule for each team. The schedule for each team indicates the opponent for each time slot (0 for a bye) and whether it corresponds to a home game, away game, or a bye.

The `%FIND_SCHEDULES` macro determines the team schedules by using the CLP procedure to match teams to pattern sets while satisfying the individual team constraints.

```plaintext
/* Try all possible pattern sets to find all valid schedules. */
%macro find_schedules;
proc transpose data=mycas.pattern_sets out=trans_good; run;

data _temp;
   set trans_good;
   set all_patterns;
run;

proc sql noprint;
   %do k = 1 %to 17; /* For each pattern */
      %do j=1 %to 18; /* For each date */
         select h&j into :col&k._h_&j
              separated by ',' from _temp where col&k=1;
         select a&j into :col&k._a_&j
              separated by ',' from _temp where col&k=1;
         select b&j into :col&k._b_&j
              separated by ',' from _temp where col&k=1;
      %end;
   %end;
run;

/* Assign individual teams to pattern sets */
/* Teams: 1 Clem, 2 Duke, 3 FSU, 4 GT, 5 UMD, 6 UNC, 7 NCSU, 8 UVA, */
/* 9 Wake */
%macro find_schedules;
proc clp out=mycas.ACC_ds varselect=maxcs findall;
   var k = [1,17]; /* pattern set index */
   %do j = 1 %to 18;
```

```plaintext
...
/* alpha(i,j): Team i's opponent on date j ( 0 = bye ). */
%do i = 1 %to 9;
  var alpha&i._&j = [0, 9];
%end;

/* Timetable constraint 1 */
/* Opponents in a time slot must be distinct */
alldiff ( %do i = 1 %to 9; alpha&i._&j %end; );

/* Timetable constraint 2 */
%do i = 1 %to 9;
  %do i1 = 1 %to 9;
    /* indicates if teams i and i1 play in time slot j */
    var X&i._&i1._&j = [0, 1];
    reify X&i._&i1._&j: (alpha&i._&j = &i1);
    /* team i plays i1 iff team i1 plays i */
    %if (&i1 > &i ) %then %do;
      lincon X&i._&i1._&j = X&i1._&i._&j;
    %end;
  %end;
%end;

/* Mirroring Scheme at team level. */
/* The dates are grouped into pairs (j, j1) such that each */
/* team plays the same opponent in dates j and j1. */
/* One of these should be a home game for each team. */
%do i = 1 %to 9;
  %do j = 1 %to 18;
    %do j1 = %eval(&j+1) %to 18;
      %if ( &j=1 and &j1=8 ) or ( &j=2 and &j1=9 ) or
        ( &j=3 and &j1=12 ) or ( &j=4 and &j1=13 ) or
        ( &j=5 and &j1=14 ) or ( &j=6 and &j1=15 ) or
        ( &j=7 and &j1=16 ) or ( &j=10 and &j1=17 ) or
        ( &j=11 and &j1=18 ) %then %do;
        lincon alpha&i._&j=alpha&i._&j1,
        /* H and A are matrices that indicate home */
        /* and away games */
        H&i._&j=A&i._&j1,
        H&i._&j1=A&i._&j;
      %end;
    %end;
  %end;
%end;

/* Timetable constraint 3 */
/* Each team plays every other team twice */
%do i = 1 %to 9;
  %do i1 = 1 %to 9;
    %if &i1 ne &i %then %do;
      lincon 0 %do j = 1 %to 18; + X&i._&i1._&j %end; = 2;
    %end;
Example 5.5: Scheduling a Major Basketball Conference

/* Timetable constraint 4 */
/* Teams do not play against themselves */
%do j = 1 %to 18;
    %do i = 1 %to 9;
        lincon alpha&i._&j<>&i;
        lincon X&i._&i._&j = 0; /* redundant */
    %end;
%end;

/* Timetable constraint 5 */
/* Setup Bye Matrix */
/* alpha&i._&j=0 means team &i has a bye on date &j. */
%do j = 1 %to 18;
    %do i = 1 %to 9;
        var B&i._&j = [0, 1]; /*Bye matrix*/
        reify B&i._&j: ( alpha&i._&j = 0 );
    %end;
%end;

/* Timetable constraint 6 */
/* alpha&i._&j=&i1 implies teams &i and &i1 play on date &j */
/* It must be a home game for one, away game for the other */
%do j = 1 %to 18;
    %do i = 1 %to 9;
        %do i1 = 1 %to 9;
            /* reify control variables.*/
            var U&i._&i1._&j = [0, 1] V&i._&i1._&j = [0, 1];
            /* if &i is home and &i1 is away. */
            reify U&i._&i1._&j: ( H&i._&j + A&i1._&j = 2);
            /* if &i1 is home and &i is away. */
            reify V&i._&i1._&j: ( A&i._&j + H&i1._&j = 2);
            /* Necessary condition if &i plays &i1 on date j */
            lincon X&i._&i1._&j <= U&i._&i1._&j + V&i._&i1._&j;
        %end;
    %end;
%end;

/* Timetable constraint 7 */
/* Each team must be home, away or have a bye on a given date */
%do j = 1 %to 18;
    %do i = 1 %to 9;
        /* Team &i is home (away) at date &j. */
        var H&i._&j = [0, 1] A&i._&j = [0, 1];
        lincon H&i._&j + A&i._&j + B&i._&j = 1;
    %end;
%end;

%do i = 1 %to 9;
    %do i1 = %eval(&i+1) %to 9;
Chapter 5: The CLP Procedure

/* Timetable constraint 8 */
/*Criterion 6 - Rival Matches */

/* The final weekend is reserved for 'rival games' */
/* unless the team plays FSU or has a bye */
%if ( &i=1 and &i1=4 ) or ( &i=2 and &i1=6 ) or
   ( &i=5 and &i1=8 ) or ( &i=7 and &i1=9 ) %then %do;
   lincon X&i._&i1._18 + B&i._18 + X&i._3_18 = 1;
/* redundant */
   lincon X&i1._&i._18 + B&i1._18 + X&i1._3_18 = 1;
%end;

/* Timetable constraint 9 */
/*Criterion 7 - Popular Matches */

/* The following pairings are specified to occur at */
/* least once in February. */
%if ( &i=2 and &i1=4 ) or ( &i=2 and &i1=9 ) or
   ( &i=4 and &i1=6 ) or ( &i=6 and &i1=9 ) %then %do;
   lincon 0 %do j = 11 %to 18; + X&i._&i1._&j %end; >= 1;
/* redundant */
   lincon 0 %do j = 11 %to 18; + X&i1._&i._&j %end; >= 1;
%end;

/* Timetable constraint 10 */
/*Criterion 8 - Opponent Sequence */

%do i = 1 %to 9;
/* No team plays two consecutive away dates against */
/* Duke (2) and UNC (6) */
%do j = 1 %to 17;
   var Q&i._26._&j = [0, 1] P&i._26._&j = [0, 1];
   reify Q&i._26._&j: ( X&i._2._&j + X&i._6._&j = 1 );
   reify P&i._26._&j: ( X&i._2._%eval(&j+1) + X&i._6._%eval(&j+1) =1 );
   lincon Q&i._26._&j + A&i._&j + P&i._26._&j + A&i._%eval(&j+1) < 4;
%end;
/* No team plays three consecutive dates against */
/* Duke(2), UNC(6) and Wake(9). */
%do j = 1 %to 16;
   var L&i._269._&j = [0, 1] M&i._269._&j = [0, 1]
   N&i._269._&j = [0, 1];
   reify L&i._269._&j: ( X&i._2._&j + X&i._6._&j + X&i._9._&j = 1 );
   reify M&i._269._&j: ( X&i._2._%eval(&j+1) + X&i._6._%eval(&j+1) +
   X&i._9._%eval(&j+1) =1 );
   reify N&i._269._&j: ( X&i._2._%eval(&j+2) + X&i._6._%eval(&j+2) +
   X&i._9._%eval(&j+2) =1 );
%end;
Example 5.5: Scheduling a Major Basketball Conference

\[ X_i \_9 \cdot \text{eval}(\&j + 2) = 1; \]
\[ \text{lincon } L_i \_269 \_\&j + M_i \_269 \_\&j + N_i \_269 \_\&j < 3; \]
\[ \% \text{end; } \]
\[ \% \text{end; } \]

/* Timetable constraint 11 */
/*---------------------------------------*/
/* Criterion 9 - Idiosyncratic Constraints */
/*---------------------------------------*/
/* UNC plays Duke in date 11 and 18 */
lincon alpha6_11 = 2;
lincon alpha6_18 = 2;
/* UNC plays Clem in the second date. */
lincon alpha6_2 = 1;
/* Duke has a bye in date 16. */
lincon B2_16 = 1;
/* Wake does not play home in date 17. */
lincon H9_17 = 0;
/* Wake has a bye in the first date. */
lincon B9_1 = 1;
/* Clem, Duke, UMD and Wake do not play away in the last date. */
lincon A1_18 = 0;
lincon A2_18 = 0;
lincon A5_18 = 0;
lincon A9_18 = 0;
/* Clem, FSU, and GT do not play away in the first date. */
lincon A1_1 = 0;
lincon A3_1 = 0;
lincon A4_1 = 0;
/* FSU and NCSU do not have a bye in the last date. */
lincon B3_18 = 0;
lincon B7_18 = 0;
/* UNC does not have a bye in the first date. */
lincon B6_1 = 0;

/* Timetable constraint 12 */
/*---------------------------------------*/
/* Match teams with patterns. */
/*---------------------------------------*/
%do i = 1 %to 9; /* For each team */
  var p&i=[1,9];
  var pp&i = [1,153];
  lincon pp&i = 9 * k - 9 + p&i;
%do j=1 %to 18; /* For each date */
  element ( pp&i, ( %do k = 1 %to 17; &&col&k._h_&j %end; ), H&i._&j );
  element ( pp&i, ( %do k = 1 %to 17; &&col&k._a_&j %end; ), A&i._&j );
The %FIND_SCHEDULES macro generates 179 possible schedules including the one that was eventually used by the ACC, which is displayed in Output 5.5.1.
### Example 5.6: Balanced Incomplete Block Design

Balanced incomplete block design (BIBD) generation is a standard combinatorial problem from design theory. The concept was originally developed in the design of statistical experiments; applications have expanded to other fields, such as coding theory, network reliability, and cryptography. A BIBD is an arrangement of \( v \) distinct objects into \( b \) blocks such that the following conditions are met:

- Each block contains exactly \( k \) distinct objects.
- Each object occurs in exactly \( r \) different blocks.
- Every two distinct objects occur together in exactly \( \lambda \) blocks.

A BIBD is therefore specified by its parameters \( (v, b, r, k, \lambda) \). It can be proved that when a BIBD exists, its parameters must satisfy the following conditions:

- \( rv = bk \)
- \( \lambda(v - 1) = r(k - 1) \)
- \( b \geq v \)

### ACC Basketball Tournament Schedule

<table>
<thead>
<tr>
<th>Team</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>UMD</td>
<td>UNC</td>
<td>Wake</td>
<td>FSU</td>
<td>Duke</td>
<td>UVA</td>
<td>UMD</td>
<td>UNC</td>
<td>NCSU</td>
<td>GT</td>
<td>Wake</td>
<td>FSU</td>
<td>Duke</td>
<td>UVA</td>
<td>NCSU</td>
<td>GT</td>
<td></td>
</tr>
<tr>
<td>UNC</td>
<td>NCSU</td>
<td>UMD</td>
<td>FSU</td>
<td>Wake</td>
<td>Clem</td>
<td>UVA</td>
<td>UMD</td>
<td>GT</td>
<td>UNC</td>
<td>NCSU</td>
<td>FSU</td>
<td>Wake</td>
<td>Clem</td>
<td>GT</td>
<td>UNC</td>
<td></td>
</tr>
<tr>
<td>FSU</td>
<td>NCSU</td>
<td>UMD</td>
<td>Duke</td>
<td>Clem</td>
<td>GT</td>
<td>Wake</td>
<td>UNC</td>
<td>NCSU</td>
<td>UVA</td>
<td>UMD</td>
<td>Duke</td>
<td>Clem</td>
<td>GT</td>
<td>Wake</td>
<td>UVA</td>
<td></td>
</tr>
<tr>
<td>GT</td>
<td>UNC</td>
<td>Wake</td>
<td>UVA</td>
<td>FSU</td>
<td>UMD</td>
<td>NCSU</td>
<td>UMD</td>
<td>Duke</td>
<td>Clem</td>
<td>UNC</td>
<td>Wake</td>
<td>UVA</td>
<td>FSU</td>
<td>UMD</td>
<td>Duke</td>
<td></td>
</tr>
<tr>
<td>UMD</td>
<td>FSU</td>
<td>Clem</td>
<td>GT</td>
<td>UVA</td>
<td>UMD</td>
<td>NCSU</td>
<td>FSU</td>
<td>Clem</td>
<td>Wake</td>
<td>Duke</td>
<td>GT</td>
<td>UVA</td>
<td>UMD</td>
<td>NCSU</td>
<td>Wake</td>
<td></td>
</tr>
<tr>
<td>UNC</td>
<td>GT</td>
<td>FSU</td>
<td>Duke</td>
<td>UMD</td>
<td>UVA</td>
<td>UNC</td>
<td>GT</td>
<td>FSU</td>
<td>Clem</td>
<td>Wake</td>
<td>Duke</td>
<td>UMD</td>
<td>UVA</td>
<td>UNC</td>
<td>Clem</td>
<td></td>
</tr>
<tr>
<td>NCSU</td>
<td>GT</td>
<td>FSU</td>
<td>Duke</td>
<td>UMD</td>
<td>UVA</td>
<td>UNC</td>
<td>GT</td>
<td>FSU</td>
<td>Clem</td>
<td>Wake</td>
<td>Duke</td>
<td>UMD</td>
<td>UVA</td>
<td>UNC</td>
<td>Clem</td>
<td></td>
</tr>
<tr>
<td>UVA</td>
<td>Wake</td>
<td>UNC</td>
<td>GT</td>
<td>NCSU</td>
<td>Clem</td>
<td>Duke</td>
<td>Wake</td>
<td>UMD</td>
<td>FSU</td>
<td>UNC</td>
<td>GT</td>
<td>NCSU</td>
<td>Clem</td>
<td>UMD</td>
<td>FSU</td>
<td></td>
</tr>
<tr>
<td>Wake</td>
<td>UVA</td>
<td>Clem</td>
<td>GT</td>
<td>Duke</td>
<td>UMD</td>
<td>FSU</td>
<td>UVA</td>
<td>UNC</td>
<td>NCSU</td>
<td>Clem</td>
<td>GT</td>
<td>Duke</td>
<td>UMD</td>
<td>FSU</td>
<td>NCSU</td>
<td></td>
</tr>
</tbody>
</table>
The preceding conditions are not sufficient to guarantee the existence of a BIBD (Prestwich 2001). For example, the parameters \((15, 21, 7, 5, 2)\) satisfy the preceding conditions, but a BIBD that has these parameters does not exist. Computational methods of BIBD generation usually suffer from combinatorial explosion, in part because of the large number of symmetries: for any solution, any two objects or blocks can be exchanged to obtain another solution.

This example demonstrates how to express a BIBD problem as a CSP and how to use lexicographic ordering constraints to break symmetries. (Note that this example is for illustration only. SAS provides an autocall macro, MKTBIBD, for solving BIBD problems.) The most direct CSP model for BIBD, as described in Meseguer and Torras (2001), represents a BIBD as a \(v \times b\) matrix \(X\). Each matrix entry is a Boolean decision variable \(X_{i,c}\) that satisfies \(X_{i,c} = 1\) if and only if block \(c\) contains object \(i\). The condition that each object occurs in exactly \(r\) blocks (or, equivalently, that there are \(r\) 1s per row) can be expressed as \(v\) linear constraints:

\[
\sum_{c=1}^{b} X_{i,c} = r \quad \text{for } i = 1, \ldots, v
\]

Alternatively, you can use global cardinality constraints to ensure that there are exactly \(b - r\) 0s and \(r\) 1s in \(X_{i,1}, \ldots, X_{i,b}\) for each object \(i\):

\[
\text{gcc}(X_{i,1}, \ldots, X_{i,b}) = ((0, 0, b - r)(1, 0, r)) \quad \text{for } i = 1, \ldots, v
\]

Similarly, the condition that each block contains exactly \(k\) objects (there are \(k\) 1s per column) can be specified by the following constraints:

\[
\text{gcc}(X_{1,c}, \ldots, X_{v,c}) = ((0, 0, v - k)(1, 0, k)) \quad \text{for } c = 1, \ldots, b
\]

To enforce the final condition that every two distinct objects occur together in exactly \(\lambda\) blocks (equivalently, that the scalar product of every pair of rows equal \(\lambda\)), you can introduce the auxiliary variables \(P_{i,j,c}\) for every \(i < j\), which indicate whether objects \(i\) and \(j\) both occur in block \(c\). The following reified constraint ensures that \(P_{i,j,c} = 1\) if and only if block \(c\) contains both objects \(i\) and \(j\):

\[
\text{reify } P_{i,j,c} : (X_{i,c} + X_{j,c} = 2)
\]

The following constraints ensure that the final condition holds:

\[
\text{gcc}(P_{i,j,1}, \ldots, P_{i,j,b}) = ((0, 0, b - \lambda)(1, 0, \lambda)) \quad \text{for } i = 1, \ldots, v - 1 \text{ and } j = i + 1, \ldots, v
\]

The objects and the blocks are interchangeable, so the matrix \(X\) has total row symmetry and total column symmetry. Because of the constraints on the rows, no pair of rows can be equal unless \(r = \lambda\). To break the row symmetry, you can impose strict lexicographical ordering on the rows of \(X\) as follows:

\[
(X_{1,1}, \ldots, X_{i,b}) \preceq_{\text{lex}} (X_{i-1,1}, \ldots, X_{i-1,b}) \quad \text{for } i = 2, \ldots, v
\]

To break the column symmetry, you can impose lexicographical ordering on the columns of \(X\) as follows:

\[
(X_{1,c}, \ldots, X_{v,c}) \preceq_{\text{lex}} (X_{1,c-1}, \ldots, X_{v,c-1}) \quad \text{for } c = 2, \ldots, b
\]

The following SAS macro incorporates all the preceding constraints. For the specified parameters \((v, b, r, k, \lambda)\), the macro either finds BIBDs or proves that a BIBD does not exist.
Example 5.6: Balanced Incomplete Block Design

%macro bibd(v, b, r, k, lambda, out=bibdout, soln=);
/* Arrange v objects into b blocks such that:
   (i) each object occurs in exactly r blocks,
   (ii) each block contains exactly k objects,
   (iii) every pair of objects occur together in exactly lambda blocks.

Equivalently, create a binary matrix with v rows and b columns,
with r 1s per row, k 1s per column,
and scalar product lambda between any pair of distinct rows. */
/* Check necessary conditions */
%if (%eval(&r * &v) ne %eval(&b * &k)) or
   (%eval(&lambda * (&v - 1)) ne %eval(&r * (&k - 1))) or
   (&v > &b) %then %do;
   %put BIBD necessary conditions are not met.;
   %goto EXIT;
%end;
proc clp out=&out domain=[0,1] varselect=FIFO varassign=MAX &soln;
/* Decision variables: */
/* Decision variable Xi_c = 1 iff object i occurs in block c. */
var (
   %do i=1 %to &v;
   x&i._1-x&i._&b.
   %end;
) = [0,1];
/* Mandatory constraints: */
/* (i) Each object occurs in exactly r blocks. */
%let q = %eval(&b.-&r.); /* each row has &q 0s and &r 1s */
%do i=1 %to &v;
   gcc( x&i._1-x&i._&b. ) = ((0,0,&q.) (1,0,&r.));
%end;
/* (ii) Each block contains exactly k objects. */
%let h = %eval(&v.-&k.); /* each column has &h 0s and &k 1s */
%do c=1 %to &b;
   gcc( %do i=1 %to &v;
       x&i._&c.
   %end;
) = ((0,0,&h.) (1,0,&k.));
/* (iii) Every pair of objects occurs in exactly lambda blocks. */
%let t = %eval(&b.-&lambda.);
%do i=1 %to %eval(&v.-1);
   %do j=%eval(&i.+1) %to &v;
      /* auxiliary variable p_i_j_c = 1 iff both i and j occur in c */
      var ( p&i._&j._1-p&i._&j._&b. ) = [0,1];
      %do c=1 %to &b;
         reify p&i._&j._&c.: (x&i._&c. + x&j._&c. = 2);
   %end;
%end;
Chapter 5: The CLP Procedure

%end;

\[
gcc(p\&i._.\&j._.1-p\&i._.\&j._.\&b.) = ((0,0,\&t.) (1,0,\&lambda.));
\]
%end;
%end;

/* Symmetry breaking constraints: */
/* Break row symmetry via lexicographic ordering constraints. */
%do i = 2 %to &v.;
  %let i1 = %eval(&i.-1);
  lexico( (x\&i._1-x&i._&b.) LEX_LT (x&i1._1-x&i1._&b.) );
%end;

/* Break column symmetry via lexicographic ordering constraints. */
%do c = 2 %to &b.;
  %let c1 = %eval(&c.-1);
  lexico( ( %do i = 1 %to &v.;
           x&i._.\&c.
          %end; )
       LEX_LE
       ( %do i = 1 %to &v.;
          x&i._.\&c1.
         %end; ) ) );
%end;

run;
%put &_orclp_;
%EXIT:
%mend bibd;

The following statement invokes the macro to find a balanced incomplete block design (BIBD) for the parameters (19, 19, 9, 9, 4):

%bibd(19,19,9,9,4, out=mycas.bibdout);

The output is displayed in Output 5.6.1.
Example 5.6: Balanced Incomplete Block Design

Output 5.6.1  Balanced Incomplete Block Design for (19,19,9,9,4)

Balanced Incomplete Block Design Problem
(19, 19, 9, 9, 4)

Finally, the following statement invokes the macro to find all solutions for the instance:

%bibd(8,14,7,4,3,out=mycas.bibdout, soln=findall);

The solution summary shown in Output 5.6.2 verifies the number of solutions for this instance when you use lexicographic ordering to break symmetries (Flener et al. 2002).

Output 5.6.2  Balanced Incomplete Block Design for (8,14,7,4,3)

The CLP Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Variable Selection</td>
</tr>
<tr>
<td>Value Selection</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solutions Found</th>
<th>92</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution Time</td>
<td>0.12</td>
</tr>
</tbody>
</table>
Example 5.7: Progressive Party Problem

This example demonstrates the use of the PACK constraint to solve an instance of the progressive party problem (Smith et al. 1996). In the original progressive party problem, a number of yacht crews and their boats congregate at a yachting rally. In order for each crew to socialize with as many other crews as possible, some of the boats are selected to serve as “host boats” for six rounds of parties. The crews of the host boats stay with their boats for all six rounds. The crews of the remaining boats, called “guest crews,” are assigned to visit a different host boat in each round.

Given the number of boats at the rally, the capacity of each boat, and the size of each crew, the objective of the original problem is to assign all the guest crews to host boats for each of the six rounds, using the minimum number of host boats. The partitioning of crews into guests and hosts is fixed throughout all rounds. No two crews should meet more than once. The assignments are constrained by the spare capacities (total capacity minus crew size) of the host boats and the crew sizes of the guest boats. Some boats cannot be hosts (zero spare capacity), and other boats must be hosts.

In this instance of the problem, the designation of the minimum requirement of thirteen hosts is assumed (boats 1 through 12 and 14). The formulation solves up to nine rounds in distributed mode with 142 workers. The total capacities and crew sizes of the boats are shown in Figure 5.2.

<table>
<thead>
<tr>
<th>boatnum</th>
<th>capacity</th>
<th>crewsize</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>17</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>22</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>23</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>24</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>26</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>27</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>29</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>31</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>33</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>34</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>35</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>36</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>37</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>38</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>39</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>41</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>42</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>
The following statements and DATA steps process the data and designate host boats:

```sas
data hostability;
  set capacities;
  spareCapacity = capacity - crewsize;
run;

data hosts guests;
  set hostability;
  if (boatnum <= 12 or boatnum eq 14) then do;
    output hosts;
  end;
  else do;
    output guests;
  end;
run;

/* sort so guest boats with larger crews appear first */
proc sort data=guests;
  by descending crewsize;
run;

data capacities;
  format boatnum capacity 2.;
  set hosts guests;
  seqno = _n_;
run;
```

To model the progressive party problem for the CLP solver, first define the following sets of variables:

- **Item variables** $x_{it}$ contain the host boat number for the assignment of guest boat $i$ in round $t$.
- **Load variables** $L_{ht}$ contain the load of host boat $h$ in round $t$.
- **Variable** $m_{ijt}$ are binary variables that take a value of 1 if and only if guest boats $i$ and $j$ are assigned to the same host boat in round $t$.

Next, describe the set of constraints that are used in the model:

- **Alldifferent constraints** ensure that a guest boat is not assigned to the same host boat in different rounds.
- **Reify constraints** regulate the values that are assigned to the aforementioned indicator variables $m_{ijt}$.
- **The reified indicator variables** appear in linear constraints to enforce the requirement to meet no more than once.
- **One pack constraint** per round maintains the capacity limits of the host boats.
- **Finally**, a symmetry-breaking linear constraint orders the host boat assignments for the highest-numbered guest boat across rounds.

The following statements call the CLP procedure to define the variables, specify the constraints, and solve the problem.
%let rounds=9;
%let numhosts=13;
%global maxcap;

%macro ppp(maxtime=180);
proc sql noprint;
    select count(*) into :numboats from capacities;
    select max(capacity) into :maxcap from capacities;
    %do i = 0 %to &maxcap;
        select count(*) into :numclass_&i from capacities where capacity = &i;
    %end;
    select crewsise, spareCapacity into
        :crewsise_1-:crewsise_%scan(&numboats,1),
        :cap_1-:cap_%scan(&numboats,1) from capacities order by seqno;
quit;

proc clp out=mycas.out varselect=FIFO
    maxtime=&maxtime timetype=real;
/* assume first &numhosts boats are hosts */
/* process each round in turn */
    %do t = 1 %to &rounds;
        %do i = &numhosts+1 %to &numboats;
            /* boat i assigned host value for round t */
            var x_&i._&t = [1,&numhosts];
        %end;
        %do h = 1 %to &numhosts;
            var L_&h._&t = [0,&&cap_&h]; /* load of host boat */
        %end;
    %end;

    %do i = &numhosts+1 %to &numboats;
        /* assign different host each round */
        alldiff (x_&i._1-x_&i._&rounds);
    %end;

    %do t = 1 %to &rounds;
        %do i = &numhosts+1 %to &numboats-1;
            /* boat i assigned host value for round t */
            %do j = &i+1 %to &numboats;
                var m_&i._&j._&t = [0,1];
                reify m_&i._&j._&t : (x_&i._&t = x_&j._&t);
            %end;
        %end;
    %end;
    %do i = &numhosts+1 %to &numboats-1;
        %do j = &i+1 %to &numboats;
            lincon 1 >= 0
                %do t = 1 %to &rounds;
                    + m_&i._&j._&t
                %end;
        %end;
    %end;
/* honor capacities */
%do t = 1 %to &rounds;
Packing(
    %do i = &numhosts+1 %to &numboats;
        x_&_i._&t
    %end;
) (  
    %do i = &numhosts+1 %to &numboats;
        &&crewsize_&_i
    %end;
) (  
    %do h = 1 %to &numhosts;
        L_&_h._&t
    %end;
));
%end;

/* break symmetries */
%do t = 1 %to &rounds-1;
    Lincon x_%scan(&numboats,1)_&t < x_%scan(&numboats,1)_&eval(&t+1);
%end;

run;
%mend ppp;

%ppp;

The two charts in Output 5.7.1 show the boat assignments for the first two rounds.
The first segment of each row indicates the host boat, and the remaining labeled segments indicate the guest boats that visit it. The horizontal axis represents host boat capacity. The length of each labeled segment indicates the crew size of the boat. The length of an unlabeled segment indicates excess capacity of the host boat.
Output 5.7.1  Gantt Chart: Host Boat Schedule by Round

Host Boat Schedules by Round
Round=1

<table>
<thead>
<tr>
<th>Host Boat</th>
<th>Cap</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>1</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>3</td>
<td>39</td>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>4</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>5</td>
<td></td>
<td>19</td>
<td></td>
<td></td>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>6</td>
<td></td>
<td>23</td>
<td></td>
<td></td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>7</td>
<td></td>
<td>27</td>
<td></td>
<td></td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>8</td>
<td></td>
<td>37</td>
<td></td>
<td></td>
<td>42</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>9</td>
<td></td>
<td>15</td>
<td></td>
<td></td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>10</td>
<td></td>
<td>16</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>11</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td>25</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>34</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>35</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>36</td>
</tr>
</tbody>
</table>
The chart in **Output 5.7.2** shows all the guest boats that visit host boat 3 in each of the nine rounds. Once again, the horizontal axis represents capacity, and segment length indicates the crew size of the boat.
Finally, the chart in **Output 5.7.3** displays the visitation schedule for all guest boats. The horizontal axis indicates the round, and the segment number identifies the host boat that is visited in that round.
Example 5.8: Radio Link Frequency Assignment Problem

This example uses element and reified constraints to solve a radio link frequency assignment problem (Cabon et al. 1999), in which frequencies must be assigned to communication links from limited spectral resources in such a way that no interference occurs. The simplest case is considered, with only two types of constraints:

- soft constraints that specify that the absolute difference between two frequencies should be greater than a given distance $k$; that is, $|f_x - f_y| > k$
- hard constraints that specify that the absolute difference between two frequencies should be exactly equal to a given distance $k$; that is, $|f_x - f_y| = k$

Violation of a soft constraint incurs a cost that depends on the cost category that is associated with the constraint.

The data for the CELAR6-SUB0 instance come from MiniZinc Challenge 2016 (Stuckey, Becket, and Fischer 2010), a partial description of which is shown in Table 5.9.

### Table 5.9: CELAR6-SUB0 Instance

<table>
<thead>
<tr>
<th>Guest Boat</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>6</td>
<td>5</td>
<td>8</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>7</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>11</td>
<td>12</td>
<td>5</td>
<td>3</td>
<td>14</td>
<td>8</td>
<td>4</td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>10</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>11</td>
<td>16</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>5</td>
<td>10</td>
<td>14</td>
<td>3</td>
<td>11</td>
<td>12</td>
<td>1</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>10</td>
<td>14</td>
<td>7</td>
<td>12</td>
<td>11</td>
<td>2</td>
<td>1</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>9</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>11</td>
<td>4</td>
<td>10</td>
<td>16</td>
<td>12</td>
<td>3</td>
<td>8</td>
<td>7</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>5</td>
<td>6</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>3</td>
<td>2</td>
<td>11</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>10</td>
<td>12</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>6</td>
<td>8</td>
<td>7</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>4</td>
<td>11</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>11</td>
<td>10</td>
<td>9</td>
<td>13</td>
<td>2</td>
<td>14</td>
<td>8</td>
<td>3</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>11</td>
<td>14</td>
<td>8</td>
<td>10</td>
<td>5</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>7</td>
<td>2</td>
<td>9</td>
<td>10</td>
<td>4</td>
<td>11</td>
<td>2</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>6</td>
<td>12</td>
<td>8</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>14</td>
<td>12</td>
<td>9</td>
<td>4</td>
<td>10</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>7</td>
<td>9</td>
<td>8</td>
<td>11</td>
<td>12</td>
<td>10</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>12</td>
<td>10</td>
<td>11</td>
<td>9</td>
<td>14</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>12</td>
<td>11</td>
<td>14</td>
<td>10</td>
<td>5</td>
<td>9</td>
<td>7</td>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>12</td>
<td>14</td>
<td>9</td>
<td>11</td>
<td>10</td>
<td>7</td>
<td>6</td>
<td>8</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>16</td>
<td>10</td>
<td>12</td>
<td>11</td>
<td>9</td>
<td>3</td>
<td>7</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>16</td>
<td>11</td>
<td>10</td>
<td>12</td>
<td>9</td>
<td>4</td>
<td>1</td>
<td>8</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>14</td>
<td>12</td>
<td>11</td>
<td>10</td>
<td>3</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>8</td>
<td>7</td>
<td>10</td>
<td>9</td>
<td>11</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>3</td>
<td>12</td>
<td>14</td>
<td>8</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>9</td>
<td>12</td>
<td>6</td>
<td>7</td>
<td>16</td>
<td>11</td>
<td>13</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>8</td>
<td>9</td>
<td>11</td>
<td>16</td>
<td>10</td>
<td>4</td>
<td>7</td>
<td>3</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

Output 5.7.3  Gantt Chart: Guest Boat Schedules

Segment label indicates host boat visited in each round

---

**Example 5.8: Radio Link Frequency Assignment Problem**

This example uses element and reified constraints to solve a radio link frequency assignment problem (Cabon et al. 1999), in which frequencies must be assigned to communication links from limited spectral resources in such a way that no interference occurs. The simplest case is considered, with only two types of constraints:

- soft constraints that specify that the absolute difference between two frequencies should be greater than a given distance $k$; that is, $|f_x - f_y| > k$
- hard constraints that specify that the absolute difference between two frequencies should be exactly equal to a given distance $k$; that is, $|f_x - f_y| = k$

Violation of a soft constraint incurs a cost that depends on the cost category that is associated with the constraint.

The data for the CELAR6-SUB0 instance come from MiniZinc Challenge 2016 (Stuckey, Becket, and Fischer 2010), a partial description of which is shown in Table 5.9.
### Table 5.9 Instance Data Description

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>cost</td>
<td>Objective function weights</td>
<td>[1000, 100, 10, 1]</td>
</tr>
<tr>
<td>num_categories</td>
<td>Number of categories</td>
<td>2</td>
</tr>
<tr>
<td>categories</td>
<td>Domain categories for the frequency variables</td>
<td>[{16,30,44,...,792}, {30,44,58,...,778}]</td>
</tr>
<tr>
<td>min_freq</td>
<td>Minimum frequency</td>
<td>16</td>
</tr>
<tr>
<td>max_freq</td>
<td>Maximum frequency</td>
<td>792</td>
</tr>
<tr>
<td>num_variables</td>
<td>Number of frequency variables</td>
<td>32</td>
</tr>
<tr>
<td>domains</td>
<td>Domain category index for each variable</td>
<td>[1,1,1,1,1,1,1,1,2,2,...,1]</td>
</tr>
<tr>
<td>num_hardconstraints</td>
<td>Number of hard constraints</td>
<td>16</td>
</tr>
<tr>
<td>hardctrx</td>
<td>Index x of each hard constraint</td>
<td>[1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31]</td>
</tr>
<tr>
<td>hardctry</td>
<td>Index y of each hard constraint</td>
<td>[2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32]</td>
</tr>
<tr>
<td>hardctrk</td>
<td>Given distance $k$ between two frequencies for each hard constraint</td>
<td>[238,238,238,...,238]</td>
</tr>
<tr>
<td>num_softconstraints</td>
<td>Number of soft constraints</td>
<td>207</td>
</tr>
<tr>
<td>softctrx</td>
<td>Index x of each soft constraint</td>
<td>[1,1,1,1,1,1,1,1,1,1,1,1,1,2.....,30]</td>
</tr>
<tr>
<td>softctry</td>
<td>Index y of each soft constraint</td>
<td>[3,4,5,6,7,8,11,12,17,18,25,26,.....,31]</td>
</tr>
<tr>
<td>softctrk</td>
<td>Given distance $k$ between two frequencies for each soft constraint</td>
<td>[59,186,10,186,168,56,91,265,26,.....,259]</td>
</tr>
<tr>
<td>softctrw</td>
<td>Cost categories for each soft constraint</td>
<td>[1,2,1,2,1,4,3,4,4,2.....,3]</td>
</tr>
</tbody>
</table>

The instance data are used to generate the SAS macro variables that the CLP procedure uses. In the following SAS statements, f1-f&num_variables are the frequency variables. Other variables include TotalCost and several sets of binary variables for the element and reified constraints. A set of element constraints is created to ensure that each frequency variable takes a value from its noncontiguous domain. A set of linear and element constraints specifies the absolute difference between two frequencies to be the exact given distance. The soft constraints are implemented using binary variables and reified constraints. The binary variable ri indicates whether the corresponding soft constraint is violated and is used in the linear expression that represents the total cost.

```sas
%macro celar(maxtime);
  proc clp out=mycas.out varselect=wdeg maxtime=&maxtime timetype=real;
    /* Declare variables */
    var (f1-f&num_variables) = [&min_freq, &max_freq];
    var (TotalCost) = [0, 159];
    var eley = [1, 1];
    /* Each variable takes a value in its domain */
    %do i=1 %to &num_variables;
      %do j=1 %to &num_categories;
        %if (&domain&i. eq &j.) %then %do;
          element(f&i., (&elecategory&j.), eley);
        %end;
      %end;
    %end;
  %end;
```
Example 5.8: Radio Link Frequency Assignment Problem

/* Hard constraints */
%do i=1 %to &num_hardconstraints;
  %let fx = f&&hardctrx&i.;
  %let fy = f&&hardctry&i.;
  %let k = &&hardctrk&i.;
  var (hx&i. hy&i.) = [%eval(-&k), &k];
  lincon hx&i. = &fx - &fy;
  element (hx&i. = (%eval(-&k), &k), (&k, &k), hy&i.);
%end;

/* Soft constraints */
%do i=1 %to &num_softconstraints;
  %let fx = f&&softctrx&i.;
  %let fy = f&&softctry&i.
  var (sa1_&i. sa2_&i. sb1_&i. sb2_&i. ss&i. st&i. r&i.)=[0, 1];
  reify sa1_&i.: (&fx >= &fy);
  reify sa2_&i.: (&fx-&fy <= &&softctrk&i.);
  reify sb1_&i.: (&fx < &fy);
  reify sb2_&i.: (&fy-&fx <= &&softctrk&i.);
  reify ss&i.: (sa1_&i. + sa2_&i. = 2);
  reify st&i.: (sb1_&i. + sb2_&i. = 2);
  reify r&i.: (ss&i. + st&i. = 1);
%end;

/* Cost constraint */
lincon TotalCost = &costw1 * r1
  %do i=2 %to &num_softconstraints;
    + &&costw&i. * r&i.
  %end;
run;
%put &_orclp_
%mend celar;

The following statement invokes the %celar macro with a maximum time of 60 seconds to find a feasible solution that corresponds to the known minimum cost of 159:

%celar(maxtime=60);

The solution summary that corresponds to running in distributed mode with 100 workers is shown in Output 5.8.1. By contrast, running in single-machine mode with a maximum time of 600 seconds failed to yield a solution.

Output 5.8.1  Radio Link Frequency Assignment Problem

The CLP Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution Status</td>
<td>Solution Limit Reached</td>
</tr>
<tr>
<td>Variable Selection</td>
<td>WDEG</td>
</tr>
<tr>
<td>Value Selection</td>
<td>MIN</td>
</tr>
<tr>
<td>Solutions Found</td>
<td>1</td>
</tr>
<tr>
<td>Solution Time</td>
<td>45.674288</td>
</tr>
</tbody>
</table>
Statement and Option Cross-Reference Table

Table 5.10 shows which examples in this section use each of the statements and options in the CLP procedure.

Table 5.10  Statements and Options Specified in Examples
4.1–4.8

<table>
<thead>
<tr>
<th>Statement</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLDIFF</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELEMENT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GCC</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LEXICO</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LINCON</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OBJ</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PACK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REIFY</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARIABLE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONDATA=</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOMAIN=</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVALVARSEL=</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FINDALLSOLNS</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LB=</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MAXTIME=</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT=</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>UB=</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>VARASSIGN=</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>VARSELECT=</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

References


# Chapter 6
## The OPTLP Procedure

## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview: OPTLP Procedure</td>
<td>108</td>
</tr>
<tr>
<td>Getting Started: OPTLP Procedure</td>
<td>108</td>
</tr>
<tr>
<td>Syntax: OPTLP Procedure</td>
<td>111</td>
</tr>
<tr>
<td>1. Functional Summary</td>
<td>111</td>
</tr>
<tr>
<td>2. PROC OPTLP Statement</td>
<td>112</td>
</tr>
<tr>
<td>3. Decomposition Algorithm Statements</td>
<td>118</td>
</tr>
<tr>
<td>Details: OPTLP Procedure</td>
<td>119</td>
</tr>
<tr>
<td>1. Data Input and Output</td>
<td>119</td>
</tr>
<tr>
<td>2. Presolve</td>
<td>122</td>
</tr>
<tr>
<td>3. Pricing Strategies for the Primal and Dual Simplex Algorithms</td>
<td>123</td>
</tr>
<tr>
<td>4. Warm Start for the Primal and Dual Simplex Algorithms</td>
<td>123</td>
</tr>
<tr>
<td>5. The Network Simplex Algorithm</td>
<td>124</td>
</tr>
<tr>
<td>6. The Interior Point Algorithm</td>
<td>124</td>
</tr>
<tr>
<td>7. Iteration Log for the Primal and Dual Simplex Algorithms</td>
<td>126</td>
</tr>
<tr>
<td>8. Iteration Log for the Network Simplex Algorithm</td>
<td>127</td>
</tr>
<tr>
<td>9. Iteration Log for the Interior Point Algorithm</td>
<td>128</td>
</tr>
<tr>
<td>10. Iteration Log for the Crossover Algorithm</td>
<td>128</td>
</tr>
<tr>
<td>11. Concurrent LP</td>
<td>129</td>
</tr>
<tr>
<td>12. Parallel Processing</td>
<td>129</td>
</tr>
<tr>
<td>13. ODS Tables</td>
<td>129</td>
</tr>
<tr>
<td>14. Irreducible Infeasible Set</td>
<td>133</td>
</tr>
<tr>
<td>15. Macro Variable <em>OROPTLP</em></td>
<td>135</td>
</tr>
<tr>
<td>Examples: OPTLP Procedure</td>
<td>137</td>
</tr>
<tr>
<td>1. Example 6.1: Oil Refinery Problem</td>
<td>137</td>
</tr>
<tr>
<td>2. Example 6.2: Using the Interior Point Algorithm</td>
<td>141</td>
</tr>
<tr>
<td>3. Example 6.3: The Diet Problem</td>
<td>142</td>
</tr>
<tr>
<td>4. Example 6.4: Reoptimizing after Modifying the Objective Function</td>
<td>145</td>
</tr>
<tr>
<td>5. Example 6.5: Reoptimizing after Modifying the Right-Hand Side</td>
<td>147</td>
</tr>
<tr>
<td>6. Example 6.6: Reoptimizing after Adding a New Constraint</td>
<td>149</td>
</tr>
<tr>
<td>7. Example 6.7: Finding an Irreducible Infeasible Set</td>
<td>153</td>
</tr>
<tr>
<td>8. Example 6.8: Using the Network Simplex Algorithm</td>
<td>156</td>
</tr>
<tr>
<td>References</td>
<td>160</td>
</tr>
</tbody>
</table>
Overview: OPTLP Procedure

The OPTLP procedure provides four methods of solving linear programs (LPs). A linear program has the following formulation:

\[
\begin{align*}
\text{min} & \quad c^T x \\
\text{subject to} & \quad Ax \{\geq, =, \leq\} b \\
& \quad l \leq x \leq u
\end{align*}
\]

where

- \( x \in \mathbb{R}^n \) is the vector of decision variables
- \( A \in \mathbb{R}^{m \times n} \) is the matrix of constraints
- \( c \in \mathbb{R}^n \) is the vector of objective function coefficients
- \( b \in \mathbb{R}^m \) is the vector of constraints’ right-hand sides (RHS)
- \( l \in \mathbb{R}^n \) is the vector of lower bounds on variables
- \( u \in \mathbb{R}^n \) is the vector of upper bounds on variables

The following LP algorithms are available in the OPTLP procedure:

- primal simplex algorithm
- dual simplex algorithm
- network simplex algorithm
- interior point algorithm

The primal and dual simplex algorithms implement the two-phase simplex method. In phase I, the algorithm tries to find a feasible solution. If no feasible solution is found, the LP is infeasible; otherwise, the algorithm enters phase II to solve the original LP. The network simplex algorithm extracts a network substructure, solves this using network simplex, and then constructs an advanced basis to feed to either primal or dual simplex. The interior point algorithm implements a primal-dual predictor-corrector interior point algorithm.

PROC OPTLP requires a linear program to be specified using a data table that adheres to the MPS format, a widely accepted format in the optimization community. For details about the MPS format see Chapter 9, “The MPS-Format Data Table.”

Getting Started: OPTLP Procedure

The following example illustrates how you can use the OPTLP procedure to solve linear programs. Suppose you want to solve the following problem:

\[
\begin{align*}
\text{min} & \quad 2x_1 - 3x_2 - 4x_3 \\
\text{subject to} & \quad -2x_2 - 3x_3 \geq -5 \quad (R1) \\
& \quad x_1 + x_2 + 2x_3 \leq 4 \quad (R2) \\
& \quad x_1 + 2x_2 + 3x_3 \leq 7 \quad (R3) \\
& \quad x_1, x_2, x_3 \geq 0
\end{align*}
\]
The corresponding MPS-format SAS data set is as follows:

```sas
data example;
   input field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
   NAME . EXAMPLE . . .
   ROWS . . . . .
   N COST . . . .
   G R1 . . . .
   L R2 . . . .
   L R3 . . . .
   COLUMNS . . . . .
   . X1 COST 2 R2 1
   . X1 R3 1 . .
   . X2 COST -3 R1 -2
   . X2 R2 1 R3 2
   . X3 COST -4 R1 -3
   . X3 R2 2 R3 3
   RHS . . . . .
   . RHS R1 -5 R2 4
   . RHS R3 7 . .
   ENDATA . . . . .
;
```

You can also create this data set from an MPS-format flat file (examp.mps) by using the following SAS macro:

```sas
%mps2sasd(mpsfile = "examp.mps", outdata = example);
```

**NOTE:** The SAS macro `%MPS2SASD` is provided in SAS/OR software. See “Converting an MPS/QPS-Format File: `%MPS2SASD`” on page 264 for details.

To make the created data set available on the CAS server, you need to upload it. At the same time, you also add an `_id_` column to ensure that the MPS reader can read the file in the correct order even if it is distributed across several machines and hence loses the original ordering. For more information about the MPS-format, see Chapter 9, “The MPS-Format Data Table.”

```sas
data mycas.example;
   set example;
   _id_ = _n_;
run;
```

Alternatively, you can create the MPS data table directly in CAS and include the `_id_` column in the datalines input.

You can use the following statement to call the OPTLP procedure:

```sas
title1 'The OPTLP Procedure';
proc optlp data = mycas.example
   objsense = min
   presolver = automatic
   algorithm = primal
   primalout = mycas.expout
dualout = mycas.exdout;
run;
```

NOTE: The “N” designation for “COST” in the rows section of the data table example also specifies a minimization problem. See the section “ROWS Section” on page 257 for details.

The optimal primal and dual solutions are stored in the data tables expout and exdout, respectively, and are displayed in Figure 6.1.

```
title2 'Primal Solution';
proc print data=mycas.expout label;
run;

title2 'Dual Solution';
proc print data=mycas.exdout label;
run;
```

![Figure 6.1 Primal and Dual Solution Output](image)

For details about the type and status codes displayed for variables and constraints, see the section “Data Input and Output” on page 119.
Syntax: OPTLP Procedure

The following statements are available in the OPTLP procedure:

```
PROC OPTLP < options >;
  DECOMP < options >;
  DECOMPMASTER < options >;
  DECOMPSUBPROB < options >;
```

Functional Summary

Table 6.1 summarizes the list of options available for the OPTLP procedure, classified by function.

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Table Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the input data table</td>
<td>DATA=</td>
</tr>
<tr>
<td>Specifies the dual input data table for warm start</td>
<td>DUALIN=</td>
</tr>
<tr>
<td>Specifies the dual solution output data table</td>
<td>DUALOUT=</td>
</tr>
<tr>
<td>Specifies whether the LP model is a maximization or minimization problem</td>
<td>OBJSENSE=</td>
</tr>
<tr>
<td>Specifies the primal input data table for warm start</td>
<td>PRIMALIN=</td>
</tr>
<tr>
<td>Specifies the primal solution output data table</td>
<td>PRIMALOUT=</td>
</tr>
<tr>
<td>Saves output data tables only if optimal</td>
<td>SAVE_ONLY_IF_OPTIMAL</td>
</tr>
<tr>
<td><strong>Solver Options</strong></td>
<td></td>
</tr>
<tr>
<td>Enables or disables IIS detection</td>
<td>IIS=</td>
</tr>
<tr>
<td>Specifies the type of algorithm</td>
<td>ALGORITHM=</td>
</tr>
<tr>
<td>Specifies the type of algorithm called after network simplex</td>
<td>ALGORITHM2=</td>
</tr>
<tr>
<td><strong>Presolve Option</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the type of presolve</td>
<td>PRESOLVER=</td>
</tr>
<tr>
<td>Controls the dualization of the problem</td>
<td>DUALIZE=</td>
</tr>
<tr>
<td><strong>Control Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the feasibility tolerance</td>
<td>FEASTOL=</td>
</tr>
<tr>
<td>Specifies the frequency of printing solution progress</td>
<td>LOGFREQ=</td>
</tr>
<tr>
<td>Specifies the detail of solution progress printed in log</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the maximum number of iterations</td>
<td>MAXITER=</td>
</tr>
<tr>
<td>Specifies the time limit for the optimization process</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Specifies the optimality tolerance</td>
<td>OPTTOL=</td>
</tr>
<tr>
<td>Enables or disables printing summary</td>
<td>PRINTLEVEL=</td>
</tr>
<tr>
<td>Specifies units of CPU time or real time</td>
<td>TIMETYPE=</td>
</tr>
<tr>
<td><strong>Simplex Algorithm Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the type of initial basis</td>
<td>BASIS=</td>
</tr>
<tr>
<td>Specifies the type of pricing strategy</td>
<td>PRICETYPE=</td>
</tr>
<tr>
<td>Specifies the queue size for pricing</td>
<td>QUEUESIZE=</td>
</tr>
<tr>
<td>Enables or disables scaling of the problem</td>
<td>SCALE=</td>
</tr>
</tbody>
</table>
Table 6.1  (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies the initial seed for the random number generator</td>
<td>SEED=</td>
</tr>
</tbody>
</table>

Interior Point Algorithm Options

| Enables or disables interior crossover         | CROSSOVER=        |
| Specifies the stopping criterion based on duality gap | DUALITYGAP=       |

Parallel Options

| Enables the OPTLP procedure to run deterministically | DETERMINISTIC=    |
| Specifies number of threads for the parallel OPTLP procedure to use | NTHREADS=         |

PROC OPTLP Statement

PROC OPTLP <options> ;

You can specify the following options in the PROC OPTLP statement.

Data Table Options

DATA=CAS-libref.data-table

specifies the input data table that corresponds to the LP model. If this option is not specified, PROC OPTLP uses the most recently created data table. For more information about the input data table, see Chapter 9, “The MPS-Format Data Table.” CAS-libref.data-table is a two-level name, where CAS-libref refers to the caslib and session identifier, and data-table specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 19 in Chapter 4, “Shared Concepts.”

DUALIN=CAS-libref.data-table

DIN=CAS-libref.data-table

specifies the input data table that corresponds to the dual solution that is required for warm starting the primal and dual simplex algorithms. For more information, see the section “Data Input and Output” on page 119. CAS-libref.data-table is a two-level name, where CAS-libref refers to the caslib and session identifier, and data-table specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 19 in Chapter 4, “Shared Concepts.”

DUALOUT=CAS-libref.data-table

DOUT=CAS-libref.data-table

specifies the output data table for the dual solution. This data table contains the dual solution information. For more information, see the section “Data Input and Output” on page 119. CAS-libref.data-table is a two-level name, where CAS-libref refers to the caslib and session identifier, and data-table specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 19 in Chapter 4, “Shared Concepts.”
OBJSENSE=option
specifies whether the LP model is a minimization or a maximization problem. Specify OBJSENSE=MIN for a minimization problem and OBJSENSE=MAX for a maximization problem. Alternatively, you can specify the objective sense in the input data table; for more information, see the section “ROWS Section” on page 257. If for some reason the objective sense is specified differently in these two places, this option supersedes the objective sense specified in the input data table. If the objective sense is not specified anywhere, then PROC OPTLP interprets and solves the linear program as a minimization problem.

PRIMALIN=CAS-libref.data-table
PIN=CAS-libref.data-table
specifies the input data table corresponding to the primal solution that is required for warm starting the primal and dual simplex algorithms. See the section “Data Input and Output” on page 119 for details. CAS-libref.data-table is a two-level name, where CAS-libref refers to the caslib and session identifier, and data-table specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 19 in Chapter 4, “Shared Concepts.”

PRIMALOUT=CAS-libref.data-table
POUT=CAS-libref.data-table
specifies the output data table for the primal solution. This data table contains the primal solution information. See the section “Data Input and Output” on page 119 for details. CAS-libref.data-table is a two-level name, where CAS-libref refers to the caslib and session identifier, and data-table specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 19 in Chapter 4, “Shared Concepts.”

SAVE_ONLY_IF_OPTIMAL
specifies that the PRIMALOUT= and DUALOUT= data tables be saved only if the final solution obtained by the solver at termination is optimal. If the PRIMALOUT= and DUALOUT= options are specified, then by default (that is, omitting the SAVE_ONLY_IF_OPTIMAL option), PROC OPTLP always saves the solutions obtained at termination, regardless of the final status. If the SAVE_ONLY_IF_OPTIMAL option is not specified, the output data tables can contain an intermediate solution, if one is available.

Solver Options

ALGORITHM=option
SOLVER=option
SOL=option
specifies an LP algorithm. You can specify the following options:

PRIMAL (PS) uses the primal simplex algorithm.
DUAL (DS) uses the dual simplex algorithm.
NETWORK (NS) uses the network simplex algorithm.
INTERIORPOINT (IP) uses the interior point algorithm.
CONCURRENT (CON) uses several different algorithms in parallel.

The valid abbreviated value for each option is indicated in parentheses. By default, ALGORITHM=DUAL.
**Chapter 6: The OPTLP Procedure**

**ALGORITHM2=** *option*

**SOLVER2=** *option*

specifies an LP algorithm if ALGORITHM=NS. You can specify the following values:

- **PRIMAL (PS)** uses the primal simplex algorithm (after network simplex).
- **DUAL (DS)** uses the dual simplex algorithm (after network simplex).

The valid abbreviated value for each option is indicated in parentheses. By default, the OPTLP procedure decides which algorithm is best to use after calling the network simplex algorithm on the extracted network.

**IIS=FALSE | TRUE**

specifies whether PROC OPTLP attempts to identify a set of constraints and variables that form an irreducible infeasible set (IIS). The valid values of the IIS= option are described below.

- **FALSE** disables IIS detection.
- **TRUE** enables IIS detection.

If an IIS is found, information about infeasible constraints or variable bounds can be found in the DUALOUT= and PRIMALOUT= data tables. By default, IIS=FALSE. For more information, see the section “Irreducible Infeasible Set” on page 133.

**Presolve Options**

**DUALIZE=AUTOMATIC | OFF | ON**

controls the dualization of the problem. You can specify the following values:

- **AUTOMATIC** specifies that the presolver use a heuristic to decide whether to dualize the problem or not.
- **OFF** disables dualization. The optimization problem is solved in the form that you specify.
- **ON** specifies that the presolver formulate the dual of the linear optimization problem.

Dualization is usually helpful for problems that have many more constraints than variables. You can use this option with all simplex algorithms in PROC OPTLP, but it is most effective with the primal and dual simplex algorithms.

By default, DUALIZE=AUTOMATIC.

**PRESOLVER=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE**

**PRESOL=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE**

specifies the presolve option. You can specify the following values:

- **AUTOMATIC** applies the presolver by using the default settings.
- **NONE** disables the presolver.
- **BASIC** performs a basic presolve, such as removing empty rows, columns, and fixed variables.
MODERATE performs a basic presolve and applies other inexpensive presolve techniques.

AGGRESSIVE performs a moderate presolve and applies other aggressive (but computationally expensive) presolve techniques.

By default, PRESOLVER=AUTOMATIC, which is somewhere between the MODERATE and AGGRESSIVE settings. For more information, see the section “Presolve” on page 122.

Control Options

FEASTOL=$\epsilon$

specifies the feasibility tolerance $\epsilon \in [1E^{-9}, 1E^{-4}]$ for determining the feasibility of a variable value. The default value is $1E^{-6}$. Simplex algorithms use the absolute error and interior point algorithms use the relative error for the computation of feasibility tolerance.

LOGFREQ=$k$

PRINTFREQ=$k$

specifies that the printing of the solution progress to the iteration log is to occur after every $k$ iterations. The print frequency, $k$, is an integer between zero and the largest four-byte signed integer, which is $2^{31} - 1$.

The value $k = 0$ disables the printing of the progress of the solution.

If the LOGFREQ= option is not specified, then PROC OPTLP displays the iteration log with a dynamic frequency according to the problem size if the primal or dual simplex algorithm is used, with frequency 10,000 if the network simplex algorithm is used, or with frequency 1 if the interior point algorithm is used.

LOGLEVEL=NONE | BASIC | MODERATE | AGGRESSIVE

controls the amount of information displayed in the SAS log by the LP solver, from a short description of presolve information and summary to details at each iteration. The valid values for this option are described below.

NONE turns off all solver-related messages in SAS log.

BASIC displays a solver summary after stopping.

MODERATE prints a solver summary and an iteration log by using the interval dictated by the LOGFREQ= option.

AGGRESSIVE prints a detailed solver summary and an iteration log by using the interval dictated by the LOGFREQ= option.

By default, LOGLEVEL=MODERATE.

MAXITER=$k$

specifies the maximum number of iterations. The value $k$ can be any integer between one and the largest four-byte signed integer, which is $2^{31} - 1$. If you do not specify this option, the procedure does not stop based on the number of iterations performed. For network simplex, this iteration limit corresponds to the algorithm called after network simplex (either primal or dual simplex).
**MAXTIME=**

specifies an upper limit of \( t \) seconds of time for reading in the data and performing the optimization process. The value of the **TIMETYPE=** option determines the type of units used. If you do not specify this option, the procedure does not stop based on the amount of time elapsed. The value of \( t \) can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

**OPTTOL=**

specifies the optimality tolerance \( \varepsilon \in [1E^{-9}, 1E^{-4}] \) for declaring optimality. The default value is \( 1E^{-6} \). Simplex algorithms use the absolute error and interior point algorithms use the relative error for the computation of feasibility tolerance.

**PRINTLEVEL=**

specifies whether to print a summary of the problem and solution. If PRINTLEVEL=1, then the Output Delivery System (ODS) tables ProblemSummary, SolutionSummary, and PerformanceInfo are produced and printed. If PRINTLEVEL=2, then the same tables are produced and printed along with an additional table called ProblemStatistics. If PRINTLEVEL=0, then no ODS tables are produced or printed. By default, PRINTLEVEL=1.

For more information about the ODS tables created by PROC OPTLP, see the section “ODS Tables” on page 129.

**TIMETYPE=**

specifies whether CPU time or real time is used for the **MAXTIME=** option and the **OROPTLP** macro variable in a PROC OPTLP call. You can specify the following values:

- **CPU** specifies units of CPU time.
- **REAL** specifies units of real time.

By default, TIMETYPE=REAL.

**Simplex Algorithm Options**

**BASIS=**

specifies the option for generating an initial basis. You can specify the following values:

- **CRASH** generates an initial basis by using crash techniques (Maros 2003). The procedure creates a triangular basic matrix consisting of both decision variables and slack variables.
- **SLACK** generates an initial basis by using all slack variables.
- **WARMSTART** starts the primal and dual simplex algorithms with a user-specified initial basis. The **PRIMALIN=** and **DUALIN=** data tables are required to specify an initial basis.

The default option is determined automatically based on the problem structure. For network simplex, this option has no effect.
PRICETYPE=HYBRID | PARTIAL | FULL | DEVEX | STEEPESTEDGE

specifies the pricing strategy for the primal and dual simplex algorithms. You can specify the following values:

- **HYBRID**: uses a hybrid of the Devex and steepest-edge pricing strategies. This strategy is available for the primal simplex algorithm only.
- **PARTIAL**: uses Dantzig’s rule on a queue of decision variables. Optionally, you can specify QUEUESIZE=. This strategy is available for the primal simplex algorithm only.
- **FULL**: uses Dantzig’s rule on all decision variables.
- **DEVEX**: uses the Devex pricing strategy.
- **STEEPESTEDGE**: uses the steepest-edge pricing strategy.

The default option is determined automatically based on the problem structure. For the network simplex algorithm, this option applies only to the algorithm specified by the ALGORITHM2= option. For more information, see the section “Pricing Strategies for the Primal and Dual Simplex Algorithms” on page 123.

**QUEUESIZE=k**

specifies the queue size \( k \in [1, n] \), where \( n \) is the number of decision variables. This queue is used for finding an entering variable in the simplex iteration. The default value is chosen adaptively based on the number of decision variables. This option is used only when PRICETYPE=PARTIAL.

**SCALE=NONE | AUTOMATIC**

specifies a scaling option. You can specify the following values:

- **NONE**: disables scaling.
- **AUTOMATIC**: automatically applies scaling procedure if necessary.

By default, SCALE=AUTOMATIC.

**SEED=number**

specifies the initial seed for the random number generator. Because the seed affects the perturbation in the simplex algorithms, the result might be a different optimal solution and a different solver path, but the effect is usually negligible. The value of number can be any positive integer up to the largest four-byte signed integer, which is \( 2^{31} - 1 \). By default, SEED=100.

**Interior Point Algorithm Options**

**CROSSOVER=FALSE | TRUE**

specifies whether to convert the interior point solution to a basic simplex solution. If the interior point algorithm terminates with a solution, the crossover algorithm uses the interior point solution to create an initial basic solution. After performing primal fixing and dual fixing, the crossover algorithm calls a simplex algorithm to locate an optimal basic solution.

You can specify the following values:
**Chapter 6: The OPTLP Procedure**

**FALSE**

does not convert the interior point solution to a basic simplex solution.

**TRUE**

converts the interior point solution to a basic simplex solution.

By default, CROSSOVER=TRUE.

**DUALITYGAP=δ**

specifies the desired relative duality gap $δ \in [1E^{-9}, 1E^{-4}]$. This is the relative difference between the primal and dual objective function values and is the primary solution quality parameter. The default value is $1E^{-6}$. For more information, see the section “The Interior Point Algorithm” on page 124.

**Parallel Options**

**DETERMINISTIC=TRUE | FALSE**

specifies whether PROC OPTLP should run parallel in deterministic or nondeterministic mode. You can specify the following values:

**TRUE**

runs PROC OPTLP in deterministic parallel mode.

**FALSE**

runs PROC OPTLP in nondeterministic parallel mode.

By default, DETERMINISTIC=TRUE.

**NTHREADS=number**

specifies the maximum number of threads for PROC OPTLP to use for multithreaded processing. The value of number can be any integer between 1 and 256, inclusive. The default is the number of cores on the machine that executes the process or the number of cores permissible based on your installation (whichever is less). The number of simultaneously active CPUs is limited by your installation and license configuration.

**Decomposition Algorithm Statements**

The following statements are available for the decomposition algorithm in the OPTLP procedure:

**DECOMP <options> ;**

**DECOMPMASTER <options> ;**

**DECOMPSUBPROB <options> ;**

For more information about these statements, see Chapter 18, “The Decomposition Algorithm.”
Details: OPTLP Procedure

Data Input and Output

This subsection describes the PRIMALIN= and DUALIN= data tables required to warm start the primal and dual simplex algorithms, and the PRIMALOUT= and DUALOUT= output data tables.

Definitions of Variables in the PRIMALIN= Data Table

The PRIMALIN= data table has two required variables, which are defined as follows:

- **_VAR_**
  - Specifies the name of the decision variable.

- **_STATUS_**
  - Specifies the status of the decision variable. It can take one of the following values:
    - B basic variable
    - L nonbasic variable at its lower bound
    - U nonbasic variable at its upper bound
    - F free variable
    - A newly added variable in the modified LP model when using the BASIS=WARMSTART option

**NOTE:** The PRIMALIN= data table is created from the PRIMALOUT= data table that is obtained from a previous “normal” run of PROC OPTLP (one that uses only the DATA= data table as the input).

Definitions of Variables in the DUALIN= Data Table

The DUALIN= data table also has two required variables defined as follows:

- **_ROW_**
  - Specifies the name of the constraint.

- **_STATUS_**
  - Specifies the status of the slack variable for a given constraint. It can take one of the following values:
    - B basic variable
    - L nonbasic variable at its lower bound
    - U nonbasic variable at its upper bound
    - F free variable
    - A newly added variable in the modified LP model when using the BASIS=WARMSTART option

**NOTE:** The DUALIN= data table is created from the DUALOUT= data table that is obtained from a previous “normal” run of PROC OPTLP (one that uses only the DATA= data table as the input).
Definitions of Variables in the PRIMALOUT= Data Table

The PRIMALOUT= data table contains the primal solution to the LP model; each observation corresponds to a variable of the LP problem. If the SAVE_ONLY_IF_OPTIMAL option is not specified, the PRIMALOUT= data table can contain an intermediate solution, if one is available. See Example 6.1 for an example of the PRIMALOUT= data table. The variables in the data table have the following names and meanings.

_OBJ_ID_
    specifies the name of the objective function. This is particularly useful when there are multiple objective functions, in which case each objective function has a unique name.

    NOTE: PROC OPTLP does not support simultaneous optimization of multiple objective functions in this release.

_RHS_ID_
    specifies the name of the variable that contains the right-hand-side value of each constraint.

_VAR_
    specifies the name of the decision variable.

>Type_
    specifies the type of the decision variable. _TYPE_ can take one of the following values:

    N nonnegative
    D bounded (with both lower and upper bound)
    F free
    X fixed
    O other (with either lower or upper bound)

_OBJCOEF_
    specifies the coefficient of the decision variable in the objective function.

_LBOUND_
    specifies the lower bound on the decision variable.

_UBOUND_
    specifies the upper bound on the decision variable.

_VALUE_
    specifies the value of the decision variable.

_STATUS_
    specifies the status of the decision variable. _STATUS_ can take one of the following values:

    B basic variable
    L nonbasic variable at its lower bound
    U nonbasic variable at its upper bound
    F free variable
    A superbasic variable (a nonbasic variable that has a value strictly between its bounds)
I LP model infeasible (all decision variables have _STATUS_ equal to I)

For the interior point algorithm with IIS=FALSE, _STATUS_ is blank.

The following values can appear only if IIS=TRUE. See the section “Irreducible Infeasible Set” on page 133 for details.

I_L the lower bound of the variable is needed for the IIS
I_U the upper bound of the variable is needed for the IIS
I_F both bounds of the variable needed for the IIS (the variable is fixed or has conflicting bounds)

_**R_COST**_

specifies the reduced cost of the decision variable, which is the amount by which the objective function is increased per unit increase in the decision variable. The reduced cost associated with the $i^{th}$ variable is the $i^{th}$ entry of the following vector:

$$(c^T - c^T_B B^{-1} A)$$

where $B \in \mathbb{R}^{m \times m}$ denotes the basis (matrix composed of basic columns of the constraints matrix $A \in \mathbb{R}^{m \times n}$), $c \in \mathbb{R}^n$ is the vector of objective function coefficients, and $c_B \in \mathbb{R}^m$ is the vector of objective coefficients of the variables in the basis.

**Definitions of Variables in the DUALOUT= Data Table**

The DUALOUT= data table contains the dual solution to the LP model; each observation corresponds to a constraint of the LP problem. If the SAVE_ONLY_IF_OPTIMAL option is not specified, the PRIMALOUT= data table can contain an intermediate solution, if one is available. Information about the objective rows of the LP problems is not included. See Example 6.1 for an example of the DUALOUT= data table. The variables in the data table have the following names and meanings.

_**OBJ_ID**_

specifies the name of the objective function. This is particularly useful when there are multiple objective functions, in which case each objective function has a unique name.

**NOTE:** PROC OPTLP does not support simultaneous optimization of multiple objective functions in this release.

_**RHS_ID**_

specifies the name of the variable that contains the right-hand-side value of each constraint.

_**ROW**_

specifies the name of the constraint.

_**TYPE**_

specifies the type of the constraint. _**TYPE**_ can take one of the following values:

L “less than or equals” constraint
E equality constraint
G “greater than or equals” constraint
R ranged constraint (both “less than or equals” and “greater than or equals”)
__RHS__
specifies the value of the right-hand side of the constraint. It takes a missing value for a ranged constraint.

__L_RHS__
specifies the lower bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

__U_RHS__
specifies the upper bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

__VALUE__
specifies the value of the dual variable associated with the constraint.

__STATUS__
specifies the status of the slack variable for the constraint. __STATUS__ can take one of the following values:

- B: basic variable
- L: nonbasic variable at its lower bound
- U: nonbasic variable at its upper bound
- F: free variable
- A: superbasic variable (a nonbasic variable that has a value strictly between its bounds)
- I: LP model infeasible (all decision variables have __STATUS__ equal to I)

The following values can appear only if option IIS=TRUE. See the section “Irreducible Infeasible Set” on page 133 for details.

- I_L: the “GE” (≥) condition of the constraint is needed for the IIS
- I_U: the “LE” (≤) condition of the constraint is needed for the IIS
- I_F: both conditions of the constraint are needed for the IIS (the constraint is an equality or a range constraint with conflicting bounds)

__ACTIVITY__
specifies the left-hand-side value of a constraint. In other words, the value of __ACTIVITY__ for the i\textsuperscript{th} constraint would be equal to \(a_i^T x\), where \(a_i\) refers to the i\textsuperscript{th} row of the constraints matrix and \(x\) denotes the vector of current decision variable values.

---

**Presolve**

Presolve in PROC OPTLP uses a variety of techniques to reduce the problem size, improve numerical stability, and detect infeasibility or unboundedness (Andersen and Andersen 1995; Gondzio 1997). During presolve, redundant constraints and variables are identified and removed. Presolve can further reduce the problem size by substituting variables. Variable substitution is a very effective technique, but it might occasionally increase the number of nonzero entries in the constraint matrix.

In most cases, using presolve is very helpful in reducing solution times. You can enable presolve at different levels or disable it by specifying the PRESOLVER= option.
Pricing Strategies for the Primal and Dual Simplex Algorithms

Several pricing strategies for the primal and dual simplex algorithms are available. Pricing strategies determine which variable enters the basis at each simplex pivot. They can be controlled by specifying the PRICETYPE= option.

The primal simplex algorithm has the following five pricing strategies:

- **PARTIAL** uses Dantzig’s most violated reduced cost rule (Dantzig 1963). It scans a queue of decision variables and selects the variable with the most violated reduced cost as the entering variable. You can optionally specify the QUEUESIZE= option to control the length of this queue.
- **FULL** uses Dantzig’s most violated reduced cost rule. It compares the reduced costs of all decision variables and selects the variable with the most violated reduced cost as the entering variable.
- **DEVEX** implements the Devex pricing strategy developed by Harris (1973).
- **STEEPESTEDGE** uses the steepest-edge pricing strategy developed by Forrest and Goldfarb (1992).
- **HYBRID** uses a hybrid of the Devex and steepest-edge pricing strategies.

The dual simplex algorithm has only three pricing strategies available: FULL, DEVEX, and STEEPEST-EDGE.

Warm Start for the Primal and Dual Simplex Algorithms

You can warm start the primal and dual simplex algorithms by specifying the option BASIS=WARMSTART. Additionally you need to specify the PRIMALIN= and DUALIN= data tables. The primal and dual simplex algorithms start with the basis thus provided. If the given basis cannot form a valid basis, the algorithms use their crash techniques.

After an LP model is solved using the primal and dual simplex algorithms, the BASIS=WARMSTART option enables you to perform sensitivity analysis such as modifying the objective function, changing the right-hand sides of the constraints, adding or deleting constraints or decision variables, and combinations of these cases. A faster solution to such a modified LP model can be obtained by starting with the basis in the optimal solution to the original LP model. This can be done by using the BASIS=WARMSTART option, modifying the DATA= input data table, and specifying the PRIMALIN= and DUALIN= data tables. Example 6.4 and Example 6.5 illustrate how to reoptimize an LP problem with a modified objective function and a modified right-hand side by using this technique. Example 6.6 shows how to reoptimize an LP problem after adding a new constraint.

The network simplex algorithm ignores the option BASIS=WARMSTART.

**Caution:** Since the presolver uses the objective function and/or right-hand-side information, the basis provided by you might not be valid for the presolved model. It is therefore recommended that you turn the PRESOLVER= option off when using BASIS=WARMSTART.
The Network Simplex Algorithm

The network simplex algorithm in PROC OPTLP attempts to leverage the speed of the network simplex algorithm to more efficiently solve linear programs by using the following process:

1. It heuristically extracts the largest possible network substructure from the original problem.
2. It uses the network simplex algorithm to solve for an optimal solution to this substructure.
3. It uses this solution to construct an advanced basis to warm-start either the primal or dual simplex algorithm on the original linear programming problem.

The network simplex algorithm is a specialized version of the simplex algorithm that uses spanning-tree bases to more efficiently solve linear programming problems that have a pure network form. Such LPs can be modeled using a formulation over a directed graph, as a minimum-cost flow problem. Let $G = (N, A)$ be a directed graph, where $N$ denotes the nodes and $A$ denotes the arcs of the graph. The decision variable $x_{ij}$ denotes the amount of flow sent from node $i$ to node $j$. The cost per unit of flow on the arcs is designated by $c_{ij}$, and the amount of flow sent across each arc is bounded to be within $[l_{ij}, u_{ij}]$. The demand (or supply) at each node is designated as $b_i$, where $b_i > 0$ denotes a supply node and $b_i < 0$ denotes a demand node. The corresponding linear programming problem is as follows:

$$
\begin{align*}
\min & \quad \sum_{(i,j) \in A} c_{ij} x_{ij} \\
\text{subject to} & \quad \sum_{(i,j) \in A} x_{ij} - \sum_{(j,i) \in A} x_{ji} = b_i \quad \forall i \in N \\
& \quad x_{ij} \leq u_{ij} \quad \forall (i, j) \in A \\
& \quad x_{ij} \geq l_{ij} \quad \forall (i, j) \in A
\end{align*}
$$

The network simplex algorithm used in PROC OPTLP is the primal network simplex algorithm. This algorithm finds the optimal primal feasible solution and a dual solution that satisfies complementary slackness. Sometimes the directed graph $G$ is disconnected. In this case, the problem can be decomposed into its weakly connected components and each minimum-cost flow problem can be solved separately. After solving each component, the optimal basis for the network substructure is augmented with the non-network variables and constraints from the original problem. This advanced basis is then used as a starting point for the primal or dual simplex method. The solver automatically selects the algorithm to use after network simplex. However, you can override this selection with the ALGORITHM= option.

The network simplex algorithm can be more efficient than the other algorithms on problems with a large network substructure. You can view the size of the network structure in the log.

The Interior Point Algorithm

The interior point algorithm in PROC OPTLP implements an infeasible primal-dual predictor-corrector interior point algorithm. To illustrate the algorithm and the concepts of duality and dual infeasibility, consider the following LP formulation (the primal):

$$
\begin{align*}
\min & \quad c^T x \\
\text{subject to} & \quad Ax \geq b \\
& \quad x \geq 0
\end{align*}
$$
The corresponding dual formulation is as follows:

\[
\begin{align*}
\text{max} & \quad b^T y \\
\text{subject to} & \quad A^T y + w = c \\
& \quad y \geq 0 \\
& \quad w \geq 0
\end{align*}
\]

where \( y \in \mathbb{R}^m \) refers to the vector of dual variables and \( w \in \mathbb{R}^n \) refers to the vector of dual slack variables.

The dual formulation makes an important contribution to the certificate of optimality for the primal formulation. The primal and dual constraints combined with complementarity conditions define the first-order optimality conditions, also known as KKT (Karush-Kuhn-Tucker) conditions, which can be stated as follows:

\[
\begin{align*}
Ax - s &= b \quad \text{(primal feasibility)} \\
A^T y + w &= c \quad \text{(dual feasibility)} \\
WXe &= 0 \quad \text{(complementarity)} \\
SYe &= 0 \quad \text{(complementarity)} \\
x, y, w, s &\geq 0
\end{align*}
\]

where \( e \equiv (1, \ldots, 1)^T \) of appropriate dimension and \( s \in \mathbb{R}^m \) is the vector of primal slack variables.

**Note:** Slack variables (the \( s \) vector) are automatically introduced by the algorithm when necessary; it is therefore recommended that you not introduce any slack variables explicitly. This enables the algorithm to handle slack variables much more efficiently.

The letters \( X, Y, W, \) and \( S \) denote matrices with corresponding \( x, y, w, \) and \( s \) on the main diagonal and zero elsewhere, as in the following example:

\[
X = \begin{bmatrix}
x_1 & 0 & \cdots & 0 \\
0 & x_2 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & x_n
\end{bmatrix}
\]

If \((x^*, y^*, w^*, s^*)\) is a solution of the previously defined system of equations that represent the KKT conditions, then \( x^* \) is also an optimal solution to the original LP model.

At each iteration the interior point algorithm solves a large, sparse system of linear equations,

\[
\begin{bmatrix}
Y^{-1}S & A \\
A^T & -X^{-1}W
\end{bmatrix}
\begin{bmatrix}
\Delta y \\
\Delta x
\end{bmatrix} = \begin{bmatrix}
\Xi \\
\Theta
\end{bmatrix}
\]

where \( \Delta x \) and \( \Delta y \) denote the vector of search directions in the primal and dual spaces, respectively, and \( \Theta \) and \( \Xi \) constitute the vector of the right-hand sides.

The preceding system is known as the reduced KKT system. PROC OPTLP uses a preconditioned quasi-minimum residual algorithm to solve this system of equations efficiently.

An important feature of the interior point algorithm is that it takes full advantage of the sparsity in the constraint matrix, thereby enabling it to efficiently solve large-scale linear programs.

The interior point algorithm works simultaneously in the primal and dual spaces. It attains optimality when both primal and dual feasibility are achieved and when complementarity conditions hold. Therefore, it is of interest to observe the following four measures where \( \|v\|_2 \) is the Euclidean norm of the vector \( v \):
Chapter 6: The OPTLP Procedure

- relative primal infeasibility measure $\alpha$:
  \[
  \alpha = \frac{\|Ax - b - s\|_2}{\|b\|_2 + 1}
  \]

- relative dual infeasibility measure $\beta$:
  \[
  \beta = \frac{\|c - ATy - w\|_2}{\|c\|_2 + 1}
  \]

- relative duality gap $\delta$:
  \[
  \delta = \frac{|c^T x - b^T y|}{|c^T x| + 1}
  \]

- absolute complementarity $\gamma$:
  \[
  \gamma = \sum_{i=1}^{n} x_i w_i + \sum_{i=1}^{m} y_i s_i
  \]

These measures are displayed in the iteration log.

---

**Iteration Log for the Primal and Dual Simplex Algorithms**

The primal and dual simplex algorithms implement a two-phase simplex algorithm. Phase I finds a feasible solution, which phase II improves to an optimal solution.

When LOGFREQ=1, the following information is printed in the iteration log:

- **Algorithm** indicates which simplex method is running by printing the letter P (primal) or D (dual).
- **Phase** indicates whether the algorithm is in phase I or phase II of the simplex method.
- **Iteration** indicates the iteration number.
- **Objective Value** indicates the current amount of infeasibility in phase I and the primal objective value of the current solution in phase II.
- **Time** indicates the time elapsed (in seconds).
- **Entering Variable** indicates the entering pivot variable. A slack variable that enters the basis is indicated by the corresponding row name followed by “(S)”. If the entering nonbasic variable has distinct and finite lower and upper bounds, then a “bound swap” can take place in the primal simplex method.
- **Leaving Variable** indicates the leaving pivot variable. A slack variable that leaves the basis is indicated by the corresponding row name followed by “(S)”. The leaving variable is the same as the entering variable if a bound swap has taken place.

When you omit the LOGFREQ= option or specify a value greater than 1, only the algorithm, phase, iteration, objective value, and time information is printed in the iteration log.
The behavior of objective values in the iteration log depends on both the current phase and the chosen algorithm. In phase I, both simplex methods have artificial objective values that decrease to 0 when a feasible solution is found. For the dual simplex method, phase II maintains a dual feasible solution, so a minimization problem has increasing objective values in the iteration log. For the primal simplex method, phase II maintains a primal feasible solution, so a minimization problem has decreasing objective values in the iteration log.

During the solution process, some elements of the LP model might be perturbed to improve performance. In this case the objective values that are printed correspond to the perturbed problem. After reaching optimality for the perturbed problem, PROC OPTLP solves the original problem by switching from the primal simplex method to the dual simplex method (or from the dual to the primal simplex method). Because the problem might be perturbed again, this process can result in several changes between the two algorithms.

### Iteration Log for the Network Simplex Algorithm

After finding the embedded network and formulating the appropriate relaxation, the network simplex algorithm uses a primal network simplex algorithm. In the case of a connected network, with one (weakly connected) component, the log shows the progress of the simplex algorithm. The following information is displayed in the iteration log:

- **Iteration** indicates the iteration number.
- **PrimalObj** indicates the primal objective value of the current solution.
- **Primal Infeas** indicates the maximum primal infeasibility of the current solution.
- **Time** indicates the time spent on the current component by network simplex.

The frequency of the simplex iteration log is controlled by the `LOGFREQ=` option. The default value of the `LOGFREQ=` option is 10,000.

If the network relaxation is disconnected, the information in the iteration log shows progress at the component level. The following information is displayed in the iteration log:

- **Component** indicates the component number being processed.
- **Nodes** indicates the number of nodes in this component.
- **Arcs** indicates the number of arcs in this component.
- **Iterations** indicates the number of simplex iterations needed to solve this component.
- **Time** indicates the time spent so far in network simplex.

The frequency of the component iteration log is controlled by the `LOGFREQ=` option. In this case, the default value of the `LOGFREQ=` option is determined by the size of the network.

The `LOGLEVEL=` option adjusts the amount of detail shown. By default, `LOGLEVEL=` is set to MODERATE and reports as described previously. If set to NONE, no information is shown. If set to BASIC, the only information shown is a summary of the network relaxation and the time spent solving the relaxation. If set to AGGRESSIVE, in the case of one component, the log displays as described previously; in the case of multiple components, for each component, a separate simplex iteration log is displayed.
Chapter 6: The OPTLP Procedure

Iteration Log for the Interior Point Algorithm

The interior point algorithm implements an infeasible primal-dual predictor-corrector interior point algorithm. The following information is displayed in the iteration log:

- **Iter** indicates the iteration number.
- **Complement** indicates the (absolute) complementarity.
- **Duality Gap** indicates the (relative) duality gap.
- **Primal Infeas** indicates the (relative) primal infeasibility measure.
- **Bound Infeas** indicates the (relative) bound infeasibility measure.
- **Dual Infeas** indicates the (relative) dual infeasibility measure.
- **Time** indicates the time elapsed (in seconds).

If the sequence of solutions converges to an optimal solution of the problem, you should see all columns in the iteration log converge to zero or very close to zero. If they do not, it can be the result of insufficient iterations being performed to reach optimality. In this case, you might need to increase the value specified in the `MAXITER=` or `MAXTIME=` options. If the complementarity or the duality gap do not converge, the problem might be infeasible or unbounded. If the infeasibility columns do not converge, the problem might be infeasible.

Iteration Log for the Crossover Algorithm

The crossover algorithm takes an optimal solution from the interior point algorithm and transforms it into an optimal basic solution. The iterations of the crossover algorithm are similar to simplex iterations; this similarity is reflected in the format of the iteration logs.

When `LOGFREQ`=1, the following information is printed in the iteration log:

- **Phase** indicates whether the primal crossover (PC) or dual crossover (DC) technique is used.
- **Iteration** indicates the iteration number.
- **Objective Value** indicates the total amount by which the superbasic variables are off their bound. This value decreases to 0 as the crossover algorithm progresses.
- **Time** indicates the time elapsed (in seconds).
- **Entering Variable** indicates the entering pivot variable. A slack variable that enters the basis is indicated by the corresponding row name followed by “(S).”
- **Leaving Variable** indicates the leaving pivot variable. A slack variable that leaves the basis is indicated by the corresponding row name followed by “(S).”

When you omit the `LOGFREQ=` option or specify a value greater than 1, only the phase, iteration, objective value, and time information is printed in the iteration log.

After all the superbasic variables have been eliminated, the crossover algorithm continues with regular primal or dual simplex iterations.
Concurrent LP

The ALGORITHM=CON option starts several different linear optimization algorithms in parallel in a single-machine mode. The OPTLP procedure automatically determines which algorithms to run and how many threads to assign to each algorithm. If sufficient resources are available, the procedure runs all four standard algorithms. When the first algorithm ends, the procedure returns the results from that algorithm and terminates any other algorithms that are still running. If you specify a value of DETERMINISTIC, the algorithm for which the results are returned is not necessarily the one that finished first. The OPTLP procedure deterministically selects the algorithm for which the results are returned. Regardless of which mode (deterministic or nondeterministic) is in effect, terminating algorithms that are still running might take a significant amount of time.

During concurrent optimization, the procedure displays the iteration log for the dual simplex algorithm. For more information about this iteration log, see the section “Iteration Log for the Primal and Dual Simplex Algorithms” on page 126. Upon termination, the procedure displays the iteration log for the algorithm that finishes first, unless the dual simplex algorithm finishes first. If you specify LOGLEVEL=AGGRESSIVE, the OPTLP procedure displays the iteration logs for all algorithms that are run concurrently.

If you specify PRINTLEVEL=2 and ALGORITHM=CON, the OPTLP procedure produces an ODS table called ConcurrentSummary. This table contains a summary of the solution statuses of all algorithms that are run concurrently.

Parallel Processing

The interior point, concurrent LP, and decomposition algorithms can be run in single-machine mode; in this mode, the computation is executed by multiple threads on a single computer.

ODS Tables

PROC OPTLP creates two Output Delivery System (ODS) tables by default. The first table, ProblemSummary, is a summary of the input LP problem. The second table, SolutionSummary, is a brief summary of the solution status. You can use ODS table names to select tables and create output data sets. For more information about ODS, see SAS Output Delivery System: Procedures Guide.

If you specify a value of 2 for the PRINTLEVEL= option, then the ProblemStatistics table and the Timing table are produced. The ProblemStatistics table contains information about the problem data. For more information, see the section “Problem Statistics” on page 132. The Timing table contains detailed information about the solution time. If you specify PRINTLEVEL=2 and ALGORITHM=CON, the ConcurrentSummary table is produced. This table contains solution status information for all algorithms that are run concurrently. For more information, see the section “Concurrent LP” on page 129.

Table 6.2 lists all the ODS tables that can be produced by the OPTLP procedure, along with the statement and option specifications required to produce each table.
Table 6.2 ODS Tables Produced by PROC OPTLP

<table>
<thead>
<tr>
<th>ODS Table Name</th>
<th>Description</th>
<th>Statement</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProblemSummary</td>
<td>Summary of the input LP problem</td>
<td>PROC OPTLP</td>
<td>PRINTLEVEL=1 (default)</td>
</tr>
<tr>
<td>SolutionSummary</td>
<td>Summary of the solution status</td>
<td>PROC OPTLP</td>
<td>PRINTLEVEL=1 (default)</td>
</tr>
<tr>
<td>ProblemStatistics</td>
<td>Description of input problem data</td>
<td>PROC OPTLP</td>
<td>PRINTLEVEL=2</td>
</tr>
<tr>
<td>ConcurrentSummary</td>
<td>Summary of the solution status for all algorithms run concurrently</td>
<td>PROC OPTLP</td>
<td>PRINTLEVEL=2, ALGORITHM=CON</td>
</tr>
<tr>
<td>Timing</td>
<td>Detailed solution timing</td>
<td>PROC OPTLP</td>
<td>PRINTLEVEL=2</td>
</tr>
</tbody>
</table>

A typical output of PROC OPTLP is shown in Figure 6.2.

Figure 6.2 Typical OPTLP Output

The OPTLP Procedure

Problem Summary

<table>
<thead>
<tr>
<th>Problem Name</th>
<th>ADLITTLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
<td>Minimization</td>
</tr>
<tr>
<td>Objective Function</td>
<td>Z....</td>
</tr>
<tr>
<td>RHS</td>
<td>ZZZZ0001</td>
</tr>
</tbody>
</table>

| Number of Variables | 97 |
| Bounded Above       | 0  |
| Bounded Below       | 97 |
| Bounded Above and Below | 0  |
| Free                | 0  |
| Fixed               | 0  |

| Number of Constraints | 56  |
| LE (<=)               | 40  |
| EQ (=)                | 15  |
| GE (>=)               | 1   |
| Range                 | 0   |

Constraint Coefficients 383
You can create output data sets from these tables by using the ODS OUTPUT statement. This can be useful, for example, when you want to create a report to summarize multiple PROC OPTLP runs. The output data sets corresponding to the preceding output are shown in Figure 6.3, where you can also find (at the row following the heading of each data set in display) the variable names that are used in the table definition (template) of each table.

#### Figure 6.3 ODS Output Data Sets

**Problem Summary**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name1</th>
<th>Label1</th>
<th>cValue1</th>
<th>nValue1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>problemName</td>
<td>Problem Name</td>
<td>ADLITTLE</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>objectiveSense</td>
<td>Objective Sense</td>
<td>Minimization</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>objectiveName</td>
<td>Objective Function</td>
<td>.Z....</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>rhs</td>
<td>RHS</td>
<td>ZZZZ0001</td>
<td>.</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>6</td>
<td>numVars</td>
<td>Number of Variables</td>
<td>97</td>
<td>97.000000</td>
</tr>
<tr>
<td>7</td>
<td>numBoundedAbove</td>
<td>Bounded Above</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>numBoundedBelow</td>
<td>Bounded Below</td>
<td>97</td>
<td>97.000000</td>
</tr>
<tr>
<td>9</td>
<td>numBounded</td>
<td>Bounded Above and Below</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>numFree</td>
<td>Free</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>numFixed</td>
<td>Fixed</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>13</td>
<td>numRows</td>
<td>Number of Constraints</td>
<td>56</td>
<td>56.000000</td>
</tr>
<tr>
<td>14</td>
<td>numLE</td>
<td>LE (&lt;=)</td>
<td>40</td>
<td>40.000000</td>
</tr>
<tr>
<td>15</td>
<td>numEQ</td>
<td>EQ (=)</td>
<td>15</td>
<td>15.000000</td>
</tr>
<tr>
<td>16</td>
<td>numGE</td>
<td>GE (&gt;=)</td>
<td>1</td>
<td>1.000000</td>
</tr>
<tr>
<td>17</td>
<td>numRange</td>
<td>Range</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>19</td>
<td>numElems</td>
<td>Constraint Coefficients</td>
<td>383</td>
<td>383.000000</td>
</tr>
</tbody>
</table>
Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of $10^9$) can cause difficulty when the remaining entries are single-digit numbers. The `PRINTLEVEL=2` option in the OPTLP procedure causes the ODS table `ProblemStatistics` to be generated. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Figure 6.4 demonstrates the contents of the ODS table `ProblemStatistics`.

**Problem Statistics**

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of $10^9$) can cause difficulty when the remaining entries are single-digit numbers. The `PRINTLEVEL=2` option in the OPTLP procedure causes the ODS table `ProblemStatistics` to be generated. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Figure 6.4 demonstrates the contents of the ODS table `ProblemStatistics`.

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name1</th>
<th>Label1</th>
<th>cValue1</th>
<th>nValue1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>solver</td>
<td>Solver</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>algorithm</td>
<td>Algorithm</td>
<td>Dual Simplex</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>objectiveName</td>
<td>Objective Function</td>
<td>Z....</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>solStatus</td>
<td>Solution Status</td>
<td>Optimal</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>objective</td>
<td>Objective Value</td>
<td>225494.96316</td>
<td>225495</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>primalInfeasibility</td>
<td>Primal Infeasibility</td>
<td>3.410605E-13</td>
<td>3.410605E-13</td>
</tr>
<tr>
<td>8</td>
<td>dualInfeasibility</td>
<td>Dual Infeasibility</td>
<td>4.547474E-13</td>
<td>4.547474E-13</td>
</tr>
<tr>
<td>9</td>
<td>boundInfeasibility</td>
<td>Bound Infeasibility</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>iterations</td>
<td>Iterations</td>
<td>73</td>
<td>73.000000</td>
</tr>
<tr>
<td>12</td>
<td>presolveTime</td>
<td>Presolve Time</td>
<td>0.00</td>
<td>0.000578</td>
</tr>
<tr>
<td>13</td>
<td>solutionTime</td>
<td>Solution Time</td>
<td>0.00</td>
<td>0.002172</td>
</tr>
</tbody>
</table>
Irreducible Infeasible Set

For a linear programming problem, an irreducible infeasible set (IIS) is an infeasible subset of constraints and variable bounds that will become feasible if any single constraint or variable bound is removed. It is possible to have more than one IIS in an infeasible LP. Identifying an IIS can help isolate the structural infeasibility in an LP.

The presolver in the OPTLP procedure can detect infeasibility, but it identifies only the variable bound or constraint that triggers the infeasibility.
The IIS=True option directs the OPTLP procedure to search for an IIS in a specified LP. The OPTLP procedure does not apply the presolver to the problem during the IIS search. If PROC OPTLP detects an IIS, it first outputs the IIS to the data sets that are specified by the PRIMALOUT= and DUALOUT= options, and then it stops. The number of iterations that are reported in the macro variable and the ODS table is the total number of simplex iterations. This total includes the initial LP solve and all subsequent iterations during the constraint deletion phase.

The IIS= option can add special values to the _STATUS_ variables in the output data sets. (For more information, see the section “Data Input and Output” on page 119.) For constraints, a status of “I_L”, “I_U”, or “I_F” indicates that the “GE” (≥), “LE” (≤), or “EQ” (=) constraint, respectively, is part of the IIS. For range constraints, a status of “I_L” or “I_U” indicates that the lower or upper bound of the constraint, respectively, is needed for the IIS, and “I_F” indicates that the bounds in the constraint are conflicting. For variables, a status of “I_L”, “I_U”, or “I_F” indicates that the lower, upper, or both bounds of the variable, respectively, are needed for the IIS. From this information, you can identify both the names of the constraints (variables) in the IIS and the corresponding bound where infeasibility occurs.

Making any one of the constraints or variable bounds in the IIS nonbinding removes the infeasibility from the IIS. In some cases, changing a right-hand side or bound by a finite amount removes the infeasibility. However, the only way to guarantee removal of the infeasibility is to set the appropriate right-hand side or bound to $\infty$ or $-\infty$. Because it is possible for an LP to have multiple irreducible infeasible sets, simply removing the infeasibility from one set might not make the entire problem feasible. To make the entire problem feasible, you can specify IIS=True and rerun PROC OPTLP after removing the infeasibility from an IIS. Repeating this process until the LP solver no longer detects an IIS results in a feasible problem. This approach to infeasibility repair can produce different end problems depending on which right-hand sides and bounds you choose to relax.

Changing different constraints and bounds can require considerably different changes to the MPS-format SAS data set. For example, if you use the default lower bound of 0 for a variable but you want to relax the lower bound to $-\infty$, you might need to add an MI row to the BOUNDS section of the data set. For more information about changing variable and constraint bounds, see Chapter 9, “The MPS-Format Data Table.”

The IIS= option in PROC OPTLP uses two different methods to identify an IIS:

1. Based on the result of the initial solve, the sensitivity filter removes several constraints and variable bounds immediately while still maintaining infeasibility. This phase is quick and dramatically reduces the size of the IIS.

2. Next, the deletion filter removes each remaining constraint and variable bound one by one to check which of them are needed to obtain an infeasible system. This second phase is more time consuming, but it ensures that the IIS set that PROC OPTLP returns is indeed irreducible. The progress of the deletion filter is reported at regular intervals. Occasionally, the sensitivity filter might be called again during the deletion filter to improve performance.

See Example 6.7 for an example that demonstrates the use of the IIS= option in locating and removing infeasibilities. You can find more details about IIS algorithms in Chinnneck (2008).
The OPTLP procedure defines a macro variable named _OROPTLP_. This variable contains a character string that indicates the status of the OPTLP procedure upon termination. The various terms of the variable are interpreted as follows.

**STATUS**
indicates the solver status at termination. It can take one of the following values:

- **OK** The procedure terminated normally.
- **SYNTAX_ERROR** Incorrect syntax was used.
- **DATA_ERROR** The input data were inconsistent.
- **OUT_OF_MEMORY** Insufficient memory was allocated to the procedure.
- **IO_ERROR** A problem occurred in reading or writing data.
- **ERROR** The status cannot be classified into any of the preceding categories.

**ALGORITHM**
indicates the algorithm that produces the solution data in the macro variable. This term appears only when STATUS=OK. It can take one of the following values:

- **PS** The primal simplex algorithm produced the solution data.
- **DS** The dual simplex algorithm produced the solution data.
- **NS** The network simplex algorithm produced the solution data.
- **IP** The interior point algorithm produced the solution data.
- **DECOMP** The decomposition algorithm produced the solution data.

When you run algorithms concurrently (ALGORITHM=CON), this term indicates which algorithm is the first to terminate.

**SOLUTION_STATUS**
indicates the solution status at termination. It can take one of the following values:

- **OPTIMAL** The solution is optimal.
- **CONDITIONAL_OPTIMAL** The solution is optimal, but some infeasibilities (primal, dual or bound) exceed tolerances due to scaling or preprocessing.
- **FEASIBLE** The problem is feasible.
- **INFEASIBLE** The problem is infeasible.
- **UNBOUNDED** The problem is unbounded.
- **INFEASIBLE_OR_UNBOUNDED** The problem is infeasible or unbounded.
- **ITERATION_LIMIT_REACHED** The maximum allowable number of iterations was reached.
- **TIME_LIMIT_REACHED** The solver reached its execution time limit.
- **ABORTED** The solver was interrupted externally.
- **FAILED** The solver failed to converge, possibly due to numerical issues.
OBJECTIVE
indicates the objective value obtained by the solver at termination.

PRIMAL_INFEASIBILITY
indicates, for the primal simplex and dual simplex algorithms, the maximum (absolute) violation of the primal constraints by the primal solution. For the interior point algorithm, this term indicates the relative violation of the primal constraints by the primal solution.

DUAL_INFEASIBILITY
indicates, for the primal simplex and dual simplex algorithms, the maximum (absolute) violation of the dual constraints by the dual solution. For the interior point algorithm, this term indicates the relative violation of the dual constraints by the dual solution.

BOUND_INFEASIBILITY
indicates, for the primal simplex and dual simplex algorithms, the maximum (absolute) violation of the lower or upper bounds (or both) by the primal solution. For the interior point algorithm, this term indicates the relative violation of the lower or upper bounds (or both) by the primal solution.

DUALITY_GAP
indicates the (relative) duality gap. This term appears only if the interior point algorithm is used.

COMPLEMENTARITY
indicates the (absolute) complementarity. This term appears only if the interior point algorithm is used.

ITERATIONS
indicates the number of iterations taken to solve the problem. When the network simplex algorithm is used, this term indicates the number of network simplex iterations taken to solve the network relaxation. When crossover is enabled, this term indicates the number of interior point iterations taken to solve the problem.

ITERATIONS2
indicates the number of simplex iterations performed by the secondary algorithm. In network simplex, the secondary algorithm is selected automatically, unless a value has been specified for the ALGORITHM2= option. When crossover is enabled, the secondary algorithm is selected automatically. This term appears only if the network simplex algorithm is used or if crossover is enabled.

PRESOLVE_TIME
indicates the time (in seconds) used in preprocessing.

SOLUTION_TIME
indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

When SOLUTION_STATUS has a value of OPTIMAL, CONDITIONAL_OPTIMAL, ITERATION_LIMIT_REACHED, or TIME_LIMIT_REACHED, all terms of the _OROPTLP_ macro variable are present; for other values of SOLUTION_STATUS, some terms do not appear.
Example 6.1: Oil Refinery Problem

Consider an oil refinery scenario. A step in refining crude oil into finished oil products involves a distillation process that splits crude into various streams. Suppose there are three types of crude available: Arabian light (a_l), Arabian heavy (a_h), and Brega (br). These crudes are distilled into light naphtha (na_l), intermediate naphtha (na_i), and heating oil (h_o). These in turn are blended into two types of jet fuel. Jet fuel j_1 is made up of 30% intermediate naphtha and 70% heating oil, and jet fuel j_2 is made up of 20% light naphtha and 80% heating oil. What amounts of the three crudes maximize the profit from producing jet fuel (j_1, j_2)? This problem can be formulated as the following linear program:

max \quad -175a_l - 165a_h - 205br + 350j_1 + 350j_2

subject to

\begin{align*}
(napha_l) & \quad 0.035 a_l + 0.03 a_h + 0.045 br = na_l \\
(napha_i) & \quad 0.1 a_l + 0.075 a_h + 0.135 br = na_i \\
(htg_oil) & \quad 0.39 a_l + 0.3 a_h + 0.43 br = h_o \\
(blend1) & \quad 0.3 j_1 \quad \leq na_i \\
(blend2) & \quad 0.2 j_2 \quad \leq na_l \\
(blend3) & \quad 0.7 j_1 + 0.8 j_2 \quad \leq h_o \\
\quad a_l & \quad \leq 110 \\
\quad a_h & \quad \leq 165 \\
\quad br & \quad \leq 80 \\
\end{align*}

and

\begin{align*}
\ni a_l, a_h, br, na_l, na_i, h_o, j_1, j_2 \geq 0
\end{align*}

The constraints “blend1” and “blend2” ensure that j_1 and j_2 are made with the specified amounts of na_i and na_l, respectively. The constraint “blend3” is actually the reduced form of the following constraints:

\begin{align*}
\quad h_o1 & \quad \geq 0.7 j_1 \\
\quad h_o2 & \quad \geq 0.8 j_2 \\
\quad h_o1 + h_o2 & \quad \leq h_o \\
\end{align*}

where h_o1 and h_o2 are dummy variables.
Chapter 6: The OPTLP Procedure

You can use the following SAS code to create the input data table `ex1`:

```sas
data mycas.ex1;
  input _id_ field1 $ field2 $ field3 $ field4 field5 $ field6;
  datalines;
  1 NAME . EX1 . . .
  2 ROWS . . . . .
  3 N profit . . . .
  4 E napha_l . . . .
  5 E napha_i . . . .
  6 E htg_oil . . . .
  7 L blend1 . . . .
  8 L blend2 . . . .
  9 L blend3 . . . .
 10 COLUMNS . . . .
 11 . a_l profit -175 napha_l .035
 12 . a_l napha_i .100 htg_oil .390
 13 . a_h profit -165 napha_l .030
 14 . a_h napha_i .075 htg_oil .300
 15 . br profit -205 napha_l .045
 16 . br napha_i .135 htg_oil .430
 17 . na_l napha_l -1 blend2 -1
 18 . na_i napha_i -1 blend1 -1
 19 . h_o htg_oil -1 blend3 -1
 20 . j_1 profit 350 blend1 .3
 21 . j_1 blend3 .7 .
 22 . j_2 profit 350 blend2 .2
 23 . j_2 blend3 .8 .
 24 BOUNDS . . . .
 25 UP . a_l 110 .
 26 UP . a_h 165 .
 27 UP . br 80 .
 28 ENDTDATA . . .
```

You can use the following call to PROC OPTLP to solve the LP problem:

```sas
proc optlp data=mycas.ex1
  objsense = max
  algorithm = primal
  primalout = mycas.ex1pout
  dualout = mycas.ex1dout
  logfreq = 1;
run;
%put &_OROPTLP_;
```

Note that the OBJSENSE=MAX option is used to indicate that the objective function is to be maximized.

The primal and dual solutions are displayed in Output 6.1.1.
Output 6.1.1 Example 1: Primal and Dual Solution Output

**Primal Solution**

<table>
<thead>
<tr>
<th>Objective Function ID</th>
<th>RHS ID</th>
<th>Variable Name</th>
<th>Variable Type</th>
<th>Objective Coefficient</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
<th>Variable Value</th>
<th>Variable Status</th>
<th>Reduced Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 profit</td>
<td>a_l</td>
<td>D</td>
<td>-175</td>
<td>0</td>
<td>110</td>
<td>110.000</td>
<td>U</td>
<td>-10.2083</td>
<td></td>
</tr>
<tr>
<td>2 profit</td>
<td>a_h</td>
<td>D</td>
<td>-165</td>
<td>0</td>
<td>165</td>
<td>0.000</td>
<td>L</td>
<td>-22.8125</td>
<td></td>
</tr>
<tr>
<td>3 profit</td>
<td>b_r</td>
<td>D</td>
<td>-205</td>
<td>0</td>
<td>80</td>
<td>80.000</td>
<td>U</td>
<td>2.8125</td>
<td></td>
</tr>
<tr>
<td>4 profit</td>
<td>na_l</td>
<td>N</td>
<td>0</td>
<td>0</td>
<td>1.7977E308</td>
<td>7.450</td>
<td>B</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>5 profit</td>
<td>na_i</td>
<td>N</td>
<td>0</td>
<td>0</td>
<td>1.7977E308</td>
<td>21.800</td>
<td>B</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>6 profit</td>
<td>h_o</td>
<td>N</td>
<td>0</td>
<td>0</td>
<td>1.7977E308</td>
<td>77.300</td>
<td>B</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>7 profit</td>
<td>j_1</td>
<td>N</td>
<td>350</td>
<td>0</td>
<td>1.7977E308</td>
<td>72.667</td>
<td>B</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>8 profit</td>
<td>j_2</td>
<td>N</td>
<td>350</td>
<td>0</td>
<td>1.7977E308</td>
<td>33.042</td>
<td>B</td>
<td>0.0000</td>
<td></td>
</tr>
</tbody>
</table>

**Dual Solution**

<table>
<thead>
<tr>
<th>Objective Function ID</th>
<th>RHS ID</th>
<th>Constraint Name</th>
<th>Constraint Type</th>
<th>Constraint RHS</th>
<th>Constraint Lower Bound</th>
<th>Constraint Upper Bound</th>
<th>Dual Solution</th>
<th>Constraint Status</th>
<th>Constraint Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 profit</td>
<td>napha_l</td>
<td>E</td>
<td>0</td>
<td>.</td>
<td>0.000 L</td>
<td></td>
<td></td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>2 profit</td>
<td>napha_i</td>
<td>E</td>
<td>0</td>
<td>.</td>
<td>-145.833 U</td>
<td></td>
<td></td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>3 profit</td>
<td>hlg_oil</td>
<td>E</td>
<td>0</td>
<td>.</td>
<td>-437.500 U</td>
<td></td>
<td></td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>4 profit</td>
<td>blend1</td>
<td>L</td>
<td>0</td>
<td>.</td>
<td>145.833 L</td>
<td></td>
<td></td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>5 profit</td>
<td>blend2</td>
<td>L</td>
<td>0</td>
<td>.</td>
<td>0.000 B</td>
<td></td>
<td></td>
<td>-0.84167</td>
<td></td>
</tr>
<tr>
<td>6 profit</td>
<td>blend3</td>
<td>L</td>
<td>0</td>
<td>.</td>
<td>437.500 L</td>
<td></td>
<td></td>
<td>-0.84167</td>
<td></td>
</tr>
</tbody>
</table>

The progress of the solution is printed to the log as follows.
Output 6.1.2 Log: Solution Progress

NOTE: The problem EX1 has 8 variables (0 free, 0 fixed).
NOTE: The problem has 6 constraints (3 LE, 3 EQ, 0 GE, 0 range).
NOTE: The problem has 19 constraint coefficients.
WARNING: The objective sense has been changed to maximization.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 3 variables and 3 constraints.
NOTE: The LP presolver removed 6 constraint coefficients.
NOTE: The presolved problem has 5 variables, 3 constraints, and 13 constraint coefficients.
NOTE: The LP solver is called.
NOTE: The Primal Simplex algorithm is used.

<table>
<thead>
<tr>
<th>Objective</th>
<th>Entering</th>
<th>Leaving</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phase Iteration</td>
<td>Value</td>
<td>Time</td>
</tr>
<tr>
<td>P 2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P 2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>P 2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>P 2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>P 2</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>P 2</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>D 2</td>
<td>7</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Optimal.
NOTE: Objective = 1347.9166667.
NOTE: The Primal Simplex solve time is 0.00 seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.684024 seconds.
NOTE: The data set MYCAS.EX1POUT has 8 observations and 10 variables.
NOTE: The data set MYCAS.EX1DOUT has 6 observations and 10 variables.

Note that the %put statement immediately after the OPTLP procedure prints value of the macro variable _OROPTLP_ to the log as follows.

Output 6.1.3 Log: Value of the Macro Variable _OROPTLP_

STATUS=OK ALGORITHM=PS SOLUTION_STATUS=OPTIMAL OBJECTIVE=1347.9166667
PRIMAL_INFEASIBILITY=0 DUAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0 ITERATIONS=7
PRESOLVE_TIME=0.00 SOLUTION_TIME=0.00

The value briefly summarizes the status of the OPTLP procedure upon termination.
Example 6.2: Using the Interior Point Algorithm

You can also solve the oil refinery problem described in Example 6.1 by using the interior point algorithm. You can create the input data table from an external MPS-format flat file by using the SAS macro %MPS2SASD or SAS DATA step code, both of which are described in “Getting Started: OPTLP Procedure” on page 108. You can use the following SAS code to solve the problem:

```sas
proc optlp data=mycas.ex1
   objsense = max
   algorithm = ip
   primalout = mycas.ex1ipout
   dualout = mycas.ex1idout
   logfreq = 1;
run;
```

The optimal solution is displayed in Output 6.2.1.

### Output 6.2.1 Interior Point Algorithm: Primal Solution Output

**Primal Solution**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Objective Function ID</th>
<th>RHS ID</th>
<th>Variable Name</th>
<th>Type</th>
<th>Objective Coefficient</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
<th>Variable Value</th>
<th>Variable Status</th>
<th>Reduced Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>profit</td>
<td>a_l</td>
<td>D</td>
<td>-175</td>
<td>0</td>
<td>110</td>
<td>110.000</td>
<td>U</td>
<td>10.2083</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>profit</td>
<td>a_h</td>
<td>D</td>
<td>-165</td>
<td>0</td>
<td>165</td>
<td>0.000</td>
<td>L</td>
<td>-22.8125</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>profit</td>
<td>br</td>
<td>D</td>
<td>-205</td>
<td>0</td>
<td>80</td>
<td>80.000</td>
<td>U</td>
<td>2.8125</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>profit</td>
<td>na_l</td>
<td>N</td>
<td>0</td>
<td>0</td>
<td>1.7977E308</td>
<td>7.450</td>
<td>B</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>profit</td>
<td>na_i</td>
<td>N</td>
<td>0</td>
<td>0</td>
<td>1.7977E308</td>
<td>21.800</td>
<td>B</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>profit</td>
<td>h_o</td>
<td>N</td>
<td>0</td>
<td>0</td>
<td>1.7977E308</td>
<td>77.300</td>
<td>B</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>profit</td>
<td>j_1</td>
<td>N</td>
<td>350</td>
<td>0</td>
<td>1.7977E308</td>
<td>72.667</td>
<td>B</td>
<td>0.0000</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>profit</td>
<td>j_2</td>
<td>N</td>
<td>350</td>
<td>0</td>
<td>1.7977E308</td>
<td>33.042</td>
<td>B</td>
<td>0.0000</td>
<td></td>
</tr>
</tbody>
</table>

The iteration log is displayed in Output 6.2.2.
Example 6.3: The Diet Problem

Consider the problem of diet optimization. There are six different foods: bread, milk, cheese, potato, fish, and yogurt. The cost and nutrition values per unit are displayed in Table 6.3.
Table 6.3 Cost and Nutrition Values

<table>
<thead>
<tr>
<th></th>
<th>Bread</th>
<th>Milk</th>
<th>Cheese</th>
<th>Potato</th>
<th>Fish</th>
<th>Yogurt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost</td>
<td>2.0</td>
<td>3.5</td>
<td>8.0</td>
<td>1.5</td>
<td>11.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Protein, g</td>
<td>4.0</td>
<td>8.0</td>
<td>7.0</td>
<td>1.3</td>
<td>8.0</td>
<td>9.2</td>
</tr>
<tr>
<td>Fat, g</td>
<td>1.0</td>
<td>5.0</td>
<td>9.0</td>
<td>0.1</td>
<td>7.0</td>
<td>1.0</td>
</tr>
<tr>
<td>Carbohydrates, g</td>
<td>15.0</td>
<td>11.7</td>
<td>0.4</td>
<td>22.6</td>
<td>0.0</td>
<td>17.0</td>
</tr>
<tr>
<td>Calories</td>
<td>90</td>
<td>120</td>
<td>106</td>
<td>97</td>
<td>130</td>
<td>180</td>
</tr>
</tbody>
</table>

The objective is to find a minimum-cost diet that contains at least 300 calories, not more than 10 grams of protein, not less than 10 grams of carbohydrates, and not less than 8 grams of fat. In addition, the diet should contain at least 0.5 unit of fish and no more than 1 unit of milk.

You can use the following SAS code to create the MPS-format input data table:

```sas
data mycas.ex3;
   input _id_ field1 $ field2 $ field3 $ field4 field5 $ field6;
datalines;
1 NAME . EX3 . . . .
2 ROWS . . . . .
3 N diet . . . . .
4 G calories . . . . .
5 L protein . . . . .
6 G fat . . . . .
7 G carbs . . . . .
8 COLUMNS . . . . .
9 . br diet 2 calories 90
10 . br protein 4 fat 1
11 . br carbs 15 . .
12 . mi diet 3.5 calories 120
13 . mi protein 8 fat 5
14 . mi carbs 11.7 . .
15 . ch diet 8 calories 106
16 . ch protein 7 fat 9
17 . ch carbs .4 . .
18 . po diet 1.5 calories 97
19 . po protein 1.3 fat .1
20 . po carbs 22.6 . .
21 . fi diet 11 calories 130
22 . fi protein 8 fat 7
23 . fi carbs 0 . .
24 . yo diet 1 calories 180
25 . yo protein 9.2 fat 1
26 . yo carbs 17 . .
27 RHS . . . . .
28 . . calories 300 protein 10
29 . . fat 8 carbs 10
30 BOUNDS . . . . .
31 UP . mi 1 . .
32 LO . fi .5 . .
33 ENDTA . . . . .
;
```

You can solve the diet problem by using PROC OPTLP as follows:

```plaintext
proc optlp data=mycas.ex3
   presolver = none
   algorithm = ps
   primalout = mycas.ex3pout
   dualout = mycas.ex3dout
   logfreq = 1;
run;
```

The solution summary and the optimal primal solution are displayed in Output 6.3.1.

**Output 6.3.1** Diet Problem: Solution Summary and Optimal Primal Solution

### Solution Summary

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name1</th>
<th>Label1</th>
<th>cValue1</th>
<th>nValue1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>solver</td>
<td>Solver</td>
<td>LP</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>algorithm</td>
<td>Algorithm</td>
<td>Primal Simplex</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>objectiveName</td>
<td>Objective Function</td>
<td>diet</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>solStatus</td>
<td>Solution Status</td>
<td>Optimal</td>
<td>.</td>
</tr>
<tr>
<td>5</td>
<td>objective</td>
<td>Objective Value</td>
<td>12.081337881</td>
<td>12.081338</td>
</tr>
<tr>
<td>6</td>
<td>primalInfeasibility</td>
<td>Primal Infeasibility</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>dualInfeasibility</td>
<td>Dual Infeasibility</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>boundInfeasibility</td>
<td>Bound Infeasibility</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>11 iterations</td>
<td>Iterations</td>
<td>6</td>
<td>6.000000</td>
</tr>
<tr>
<td>12</td>
<td>presolveTime</td>
<td>Presolve Time</td>
<td>0.00</td>
<td>0.000061035</td>
</tr>
<tr>
<td>13</td>
<td>solutionTime</td>
<td>Solution Time</td>
<td>0.00</td>
<td>0.000737</td>
</tr>
</tbody>
</table>

### Primal Solution

<table>
<thead>
<tr>
<th>Obs</th>
<th>Objective Function</th>
<th>RHS ID</th>
<th>Variable Name</th>
<th>Type</th>
<th>Objective Coefficient</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
<th>Variable Value</th>
<th>Variable Status</th>
<th>Reduced Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>diet</td>
<td>br</td>
<td>N</td>
<td></td>
<td>2.0</td>
<td>0.0</td>
<td>1.7977E308</td>
<td>0.000000</td>
<td>L</td>
<td>1.19066</td>
</tr>
<tr>
<td>2</td>
<td>diet</td>
<td>mi</td>
<td>D</td>
<td></td>
<td>3.5</td>
<td>0.0</td>
<td>1.05360</td>
<td>0.000000</td>
<td>B</td>
<td>0.000000</td>
</tr>
<tr>
<td>3</td>
<td>diet</td>
<td>ch</td>
<td>N</td>
<td></td>
<td>8.0</td>
<td>0.0</td>
<td>1.7977E308</td>
<td>0.44950</td>
<td>B</td>
<td>0.000000</td>
</tr>
<tr>
<td>4</td>
<td>diet</td>
<td>po</td>
<td>N</td>
<td></td>
<td>1.5</td>
<td>0.0</td>
<td>1.7977E308</td>
<td>1.86517</td>
<td>B</td>
<td>0.000000</td>
</tr>
<tr>
<td>5</td>
<td>diet</td>
<td>fi</td>
<td>O</td>
<td></td>
<td>11.0</td>
<td>0.5</td>
<td>1.7977E308</td>
<td>0.50000</td>
<td>L</td>
<td>5.15641</td>
</tr>
<tr>
<td>6</td>
<td>diet</td>
<td>yo</td>
<td>N</td>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.7977E308</td>
<td>0.00000</td>
<td>L</td>
<td>1.10849</td>
</tr>
</tbody>
</table>

The cost of the optimal diet is 12.08 units.
Example 6.4: Reoptimizing after Modifying the Objective Function

Using the diet problem described in Example 6.3, this example illustrates how to reoptimize an LP problem after modifying the objective function.

Assume that the optimal solution of the diet problem is found and the optimal solutions are stored in the data sets `ex3out` and `ex3dout`.

Suppose the cost of cheese increases from 8 to 10 per unit and the cost of fish decreases from 11 to 7 per serving unit. The COLUMNS section in the input data table `ex3` is updated (and the data table is saved as `ex4`) as follows:

```
COLUMNS
   . . . . .
   ... ch diet 10 calories 106
   ...
   ...
RHS
   ...
```

You can use the following DATA step to create the data table `ex4`:

```data mycas.ex4;
   input _id_ field1 $ field2 $ field3 $ field4 field5 $ field6;
   datalines;
1  NAME . EX3 . . .
2  ROWS . . . . .
3  N diet . . .
4  G calories . . .
5  L protein . . .
6  G fat . . .
7  G carbs . . .
8  COLUMNS . . . .
9  . br diet 2 calories 90
10 . br protein 4 fat 1
11 . br carbs 15 .
12 . mi diet 3.5 calories 120
13 . mi protein 8 fat 5
14 . mi carbs 11.7 .
15 . ch diet 10 calories 106
16 . ch protein 7 fat 9
17 . ch carbs .4 .
18 . po diet 1.5 calories 97
19 . po protein 1.3 fat .1
20 . po carbs 22.6 .
21 . fi diet 7 calories 130
22 . fi protein 8 fat 7
```

You can use the BASIS=WARMSTART option (and the ex3pout and ex3dout data tables from Example 6.3) in the following call to PROC OPTLP to solve the modified problem:

```plaintext
proc optlp data=mycas.ex4
    presolver = none
    basis = warmstart
    primalin = mycas.ex3pout
    dualin = mycas.ex3dout
    algorithm = primal
    primalout = mycas.ex4pout
    dualout = mycas.ex4dout
    logfreq = 1;
run;
```

The following iteration log indicates that it takes the primal simplex algorithm no extra iterations to solve the modified problem by using BASIS=WARMSTART, since the optimal solution to the LP problem in Example 6.3 remains optimal after the objective function is changed.

**Output 6.4.1 Iteration Log**

Note that the primal simplex algorithm is preferred because the primal solution to the original LP is still feasible for the modified problem in this case.
Example 6.5: Reoptimizing after Modifying the Right-Hand Side

You can also modify the right-hand side of your problem and use the BASIS=WARMSTART option to obtain an optimal solution more quickly. Since the dual solution to the original LP is still feasible for the modified problem in this case, the dual simplex algorithm is preferred. This case is illustrated by using the same diet problem as in Example 6.3. Assume that you now need a diet that supplies at least 150 calories. The RHS section in the input data table ex3 is updated (and the data set is saved as ex5) as follows:

```
... RHS ...
  . . . . . .
  . . calories 150 protein 10
  . . fat 8 carbs 10
  . . . . . BOUNDS ...
...`
```

You can use the following DATA step to create the data table ex5:

```
data mycas.ex5;
  input _id_ field1 $ field2 $ field3 $ field4 field5 $ field6;
  datalines;
1  NAME . EX3 . . .
2  ROWS . . . . .
3  N diet . . . .
4  G calories . . . .
5  L protein . . . .
6  G fat . . . .
7  G carbs . . . .
8  COLUMNS . . . . .
9  . br diet 2 calories 90
10 . br protein 4 fat 1
11 . br carbs 15 . .
12 . mi diet 3.5 calories 120
13 . mi protein 8 fat 5
14 . mi carbs 11.7 . .
15 . ch diet 8 calories 106
16 . ch protein 7 fat 9
17 . ch carbs .4 . .
18 . po diet 1.5 calories 97
19 . po protein 1.3 fat .1
20 . po carbs 22.6 . .
21 . fi diet 11 calories 130
22 . fi protein 8 fat 7
23 . fi carbs 0 . .
24 . yo diet 1 calories 180
25 . yo protein 9.2 fat 1
26 . yo carbs 17 . .
27 RHS . . . . .
28 . . calories 150 protein 10
29 . . fat 8 carbs 10
30 BOUNDS . . . . .
31 UP . . . . .
32 LO . . . . .
33 ENDATA . . . . .
;```

You can use the BASIS=WARMSTART option in the following call to PROC OPTLP to solve the modified problem:

```plaintext
proc optlp data=mycas.ex5
    presolver = none
    basis = warmstart
    primalin = mycas.ex3pout
    dualin = mycas.ex3dout
    algorithm = dual
    primalout = mycas.ex5pout
    dualout = mycas.ex5dout
    logfreq = 1;
run;
```

Note that the dual simplex algorithm is preferred because the dual solution to the last solved LP is still feasible for the modified problem in this case.

The following iteration log indicates that it takes the dual simplex algorithm just one more phase II iteration to solve the modified problem by using BASIS=WARMSTART.

Using BASIS=WARMSTART

```
Output 6.5.1 Iteration Log

NOTE: The problem EX3 has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 constraint coefficients.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Iteration</th>
<th>Value</th>
<th>Time</th>
<th>Entering Variable</th>
<th>Leaving Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>2</td>
<td>8.813205E+00</td>
<td>0</td>
<td>calories (S)</td>
<td>carbs (S)</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>9.174413E+00</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Optimal.
NOTE: Objective = 9.1744131985.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.902546 seconds.
NOTE: The data set MYCAS.EX5POUT has 6 observations and 10 variables.
NOTE: The data set MYCAS.EX5DOUT has 4 observations and 10 variables.
```

Compare this with the following call to PROC OPTLP:

```plaintext
proc optlp data=mycas.ex5
    presolver = none
    algorithm = dual
    logfreq = 1;
run;
```
This call to PROC OPTLP solves the modified problem “from scratch” (without using the BASIS=WARMSTART option) and produces the following iteration log.

**Output 6.5.2  Iteration Log**

<table>
<thead>
<tr>
<th>Phase</th>
<th>Iteration</th>
<th>Objective Value</th>
<th>Entering Variable</th>
<th>Leaving Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>D 2</td>
<td>1</td>
<td>5.500000E+00</td>
<td>mi</td>
<td>fat (S)</td>
</tr>
<tr>
<td>D 2</td>
<td>2</td>
<td>8.650000E+00</td>
<td>ch</td>
<td>protein (S)</td>
</tr>
<tr>
<td>D 2</td>
<td>3</td>
<td>8.925676E+00</td>
<td>po</td>
<td>carbs (S)</td>
</tr>
<tr>
<td>D 2</td>
<td>4</td>
<td>9.174413E+00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Optimal.
NOTE: Objective = 9.1744131985.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.742009 seconds.

It is clear that using the BASIS=WARMSTART option saves computation time. For larger or more complex examples, the benefits of using this option are more pronounced.

---

**Example 6.6: Reoptimizing after Adding a New Constraint**

Assume that after solving the diet problem in Example 6.3 you need to add a new constraint on sodium intake of no more than 550 mg/day for adults. The updated nutrition data are given in Table 6.4.

**Table 6.4  Updated Cost and Nutrition Values**

<table>
<thead>
<tr>
<th>Bread</th>
<th>Milk</th>
<th>Cheese</th>
<th>Potato</th>
<th>Fish</th>
<th>Yogurt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost</td>
<td>2.0</td>
<td>3.5</td>
<td>8.0</td>
<td>1.5</td>
<td>11.0</td>
</tr>
<tr>
<td>Protein, g</td>
<td>4.0</td>
<td>8.0</td>
<td>7.0</td>
<td>1.3</td>
<td>8.0</td>
</tr>
<tr>
<td>Fat, g</td>
<td>1.0</td>
<td>5.0</td>
<td>9.0</td>
<td>0.1</td>
<td>7.0</td>
</tr>
<tr>
<td>Carbohydrates, g</td>
<td>15.0</td>
<td>11.7</td>
<td>0.4</td>
<td>22.6</td>
<td>0.0</td>
</tr>
<tr>
<td>Calories, Cal</td>
<td>90</td>
<td>120</td>
<td>106</td>
<td>97</td>
<td>130</td>
</tr>
<tr>
<td>sodium, mg</td>
<td>148</td>
<td>122</td>
<td>337</td>
<td>186</td>
<td>56</td>
</tr>
</tbody>
</table>

The input data table `ex3` is updated (and the data table is saved as `ex6`) as follows:

```bash
/* added a new constraint to the diet problem */
data mycas.ex6;
  input _id_ field1 $ field2 $ field3 $ field4 field5 $ field6;
datalines;
1 NAME . . . . .
```
For the modified problem you can warm start the primal and dual simplex algorithms to get a solution faster. The dual simplex algorithm is preferred because a dual feasible solution can be readily constructed from the optimal solution to the diet optimization problem.

Since there is a new constraint in the modified problem, you can use the following SAS code to create a new DUALIN= data table `ex6din` with this information:

```sas
data mycas.ex6newcon;
   _ROW_='sodium '; _STATUS_='A';
   output;
run;

/* create a new DUALIN= data set to include the new constraint */
data mycas.ex6din;
   set mycas.ex3dout mycas.ex6newcon;
run;
```
Example 6.6: Reoptimizing after Adding a New Constraint

Note that this step is optional. In this example, you can still use the data table ex3dout as the DUALIN= data set to solve the modified LP problem by using the BASIS=WARMSTART option. PROC OPTLP validates the PRIMALIN= and DUALIN= data sets against the input model. Any new variable (or constraint) in the model is added to the PRIMALIN= (or DUALIN=) data set, and its status is assigned to be ‘A’. The primal and dual simplex algorithms decide its corresponding status internally. Any variable in the PRIMALIN= and DUALIN= data sets but not in the input model is removed.

The _ROW_ and _STATUS_ columns of the DUALIN= data table ex6din are shown in Output 6.6.1.

**Output 6.6.1** DUALIN= Data Set with a Newly Added Constraint

<table>
<thead>
<tr>
<th>Obs</th>
<th><em>ROW</em></th>
<th><em>STATUS</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>calories</td>
<td>U</td>
</tr>
<tr>
<td>2</td>
<td>protein</td>
<td>L</td>
</tr>
<tr>
<td>3</td>
<td>fat</td>
<td>U</td>
</tr>
<tr>
<td>4</td>
<td>carbs</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>sodium</td>
<td>A</td>
</tr>
</tbody>
</table>

The dual simplex algorithm is called to solve the modified diet optimization problem more quickly with the following SAS code:

```sas
proc optlp data=mycas.ex6
   objsense=min
   presolver=none
   algorithm=ds
   primalout=mycas.ex6pout
dualout=mycas.ex6dout
scale=none
logfreq=1
basis=warmstart
primalin=mycas.ex3pout
dualin=mycas.ex6din;
run;
```

The optimal primal and dual solutions of the modified problem are displayed in Output 6.6.2.

**Output 6.6.2** Primal and Dual Solution Output

**Primal Solution**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Objective Function ID</th>
<th>RHS ID</th>
<th>Variable Name</th>
<th>Variable Type</th>
<th>Objective Coefficient</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
<th>Variable Value</th>
<th>Variable Status</th>
<th>Reduced Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>diet</td>
<td>br</td>
<td>N</td>
<td></td>
<td>2.0</td>
<td>0.0</td>
<td>1.7977E308</td>
<td>0.00000</td>
<td>L</td>
<td>1.19066</td>
</tr>
<tr>
<td>2</td>
<td>diet</td>
<td>mi</td>
<td>D</td>
<td></td>
<td>3.5</td>
<td>0.0</td>
<td>1.00536</td>
<td>0.00000</td>
<td>B</td>
<td>0.00000</td>
</tr>
<tr>
<td>3</td>
<td>diet</td>
<td>ch</td>
<td>N</td>
<td></td>
<td>8.0</td>
<td>0.0</td>
<td>1.7977E308</td>
<td>0.44950</td>
<td>B</td>
<td>0.00000</td>
</tr>
<tr>
<td>4</td>
<td>diet</td>
<td>po</td>
<td>N</td>
<td></td>
<td>1.5</td>
<td>0.0</td>
<td>1.7977E308</td>
<td>1.86517</td>
<td>B</td>
<td>0.00000</td>
</tr>
<tr>
<td>5</td>
<td>diet</td>
<td>fi</td>
<td>O</td>
<td></td>
<td>11.0</td>
<td>0.5</td>
<td>1.7977E308</td>
<td>0.50000</td>
<td>L</td>
<td>5.15641</td>
</tr>
<tr>
<td>6</td>
<td>diet</td>
<td>yo</td>
<td>N</td>
<td></td>
<td>1.0</td>
<td>0.0</td>
<td>1.7977E308</td>
<td>0.00000</td>
<td>L</td>
<td>1.0849</td>
</tr>
</tbody>
</table>
The iteration log shown in Output 6.6.3 indicates that it takes the dual simplex algorithm no more iterations to solve the modified problem by using the BASIS=WARMSTART option, since the optimal solution to the original problem remains optimal after one more constraint is added.

Output 6.6.3 Iteration Log

NOTE: The problem EX6 has 6 variables (0 free, 0 fixed).
NOTE: The problem has 5 constraints (2 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 29 constraint coefficients.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Iteration</th>
<th>Value</th>
<th>Time</th>
<th>Entering Variable</th>
<th>Leaving Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>2</td>
<td>1.208134E+01</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Optimal.
NOTE: Objective = 12.081337881.
NOTE: The Dual Simplex solve time is 0.00 seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.952535 seconds.
NOTE: The data set MYCAS.EX6POUT has 6 observations and 10 variables.
NOTE: The data set MYCAS.EX6DOUT has 5 observations and 10 variables.

Both this example and Example 6.4 illustrate the situation in which the optimal solution does not change after some perturbation of the parameters of the LP problem. The simplex algorithm starts from an optimal solution and quickly verifies the optimality. Usually the optimal solution of the slightly perturbed problem can be obtained after performing relatively small number of iterations if starting with the optimal solution of the original problem. In such cases you can expect a dramatic reduction of computation time, for instance, if you want to solve a large LP problem and a slightly perturbed version of this problem by using the BASIS=WARMSTART option rather than solving both problems from scratch.
Example 6.7: Finding an Irreducible Infeasible Set

This example demonstrates the use of the IIS= option to locate an irreducible infeasible set. Suppose you want to solve a linear program that has the following simple formulation:

\[
\begin{align*}
\text{min} & \quad x_1 + x_2 + x_3 \\
\text{subject to} & \quad x_1 + x_2 \geq 10 \\
& \quad x_1 + x_3 \leq 4 \\
& \quad 4 \leq x_2 + x_3 \leq 5 \\
& \quad x_1, x_2 \geq 0 \\
& \quad 0 \leq x_3 \leq 3
\end{align*}
\]

The corresponding MPS-format data table is as follows:

```/* infeasible */
data mycas.exiis;
   input _id_ field1 $ field2 $ field3 $ field4 field5 $ field6;
datalines;
1 NAME . . . . 
2 ROWS . . . . 
3 N cost . . . . 
4 G con1 . . . . 
5 L con2 . . . . 
6 G con3 . . . . 
7 COLUMNS . . . . 
8 . x1 cost 1 con1 1 
9 . x1 con2 1 . . 
10 . x2 cost 1 con1 1 
11 . x2 con3 1 . . 
12 . x3 cost 1 con2 1 
13 . x3 con3 1 . . 
14 RHS . . . . 
15 . rhs con1 10 con2 4 
16 . rhs con3 4 . . 
17 RANGES . . . . 
18 . r1 con3 1 . . 
19 BOUNDS . . . . 
20 UP b1 x3 3 . . 
21 ENDDATA . . . . ;```

It is easy to verify that the following three constraints (or rows) and one variable (or column) bound form an IIS for this problem:

\[
\begin{align*}
x_1 + x_2 & \geq 10 \\
x_1 + x_3 & \leq 4 \\
x_2 + x_3 & \leq 5 \\
x_3 & \geq 0
\end{align*}
\]
You can use the IIS=TRUE option to detect this IIS by using the following statements:

```latex
proc optlp data=mycas.exiis
   iis=true
   primalout=mycas.iis_vars
   dualout=mycas.iis_cons
   logfreq=1;
run;
```

The OPTLP procedure outputs the detected IIS to the data sets specified by the PRIMALOUT= and DUALOUT= options, then stops. The notes shown in Output 6.7.1 are printed to the log.

Output 6.7.1 The IIS= Option: Log

```
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 3 constraints (1 LE, 0 EQ, 1 GE, 1 range).
NOTE: The problem has 6 constraint coefficients.
NOTE: The LP solver is called.
NOTE: The IIS= option is enabled.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Iteration</th>
<th>Objective Value</th>
<th>Time</th>
<th>Entering Variable</th>
<th>Leaving Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>P 1</td>
<td>1</td>
<td>6.0000000E+00</td>
<td>0</td>
<td>con3 (S)</td>
<td>con3 (S)</td>
</tr>
<tr>
<td>P 1</td>
<td>2</td>
<td>5.0000000E+00</td>
<td>0</td>
<td>x1</td>
<td>con2 (S)</td>
</tr>
<tr>
<td>P 1</td>
<td>3</td>
<td>1.0000000E+00</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Applying the IIS sensitivity filter.
NOTE: The sensitivity filter removed 1 constraints and 3 variable bounds.
NOTE: Applying the IIS deletion filter.
NOTE: Processing constraints.

<table>
<thead>
<tr>
<th>Processed</th>
<th>Removed</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Processing variable bounds.

<table>
<thead>
<tr>
<th>Processed</th>
<th>Removed</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The deletion filter removed 0 constraints and 0 variable bounds.
NOTE: The IIS= option found this problem to be infeasible.
NOTE: The IIS= option found an irreducible infeasible set with 1 variables and 3 constraints.
NOTE: The IIS solve time is 0.00 seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.448224 seconds.
NOTE: The data set MYCAS.IIS_VARS has 3 observations and 10 variables.
NOTE: The data set MYCAS.IIS_CONS has 3 observations and 10 variables.
```

The data sets iis_cons and iis_vars are shown in Output 6.7.2.
Example 6.7: Finding an Irreducible Infeasible Set

Output 6.7.2 Identify Rows and Columns in the IIS

Constraints in the IIS

<table>
<thead>
<tr>
<th>Objective Function</th>
<th>RHS ID</th>
<th>Constraint Name</th>
<th>Constraint Type</th>
<th>Constraint RHS</th>
<th>Constraint Lower Bound</th>
<th>Constraint Upper Bound</th>
<th>Dual Solution Status</th>
<th>Constraint Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 cost</td>
<td>rhs</td>
<td>con1</td>
<td>G</td>
<td>10</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>I_L</td>
</tr>
<tr>
<td>2 cost</td>
<td>rhs</td>
<td>con2</td>
<td>L</td>
<td>4</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>I_U</td>
</tr>
<tr>
<td>3 cost</td>
<td>rhs</td>
<td>con3</td>
<td>R</td>
<td>5</td>
<td>4</td>
<td>.</td>
<td>.</td>
<td>I_U</td>
</tr>
</tbody>
</table>

Variables in the IIS

The constraint \( x_2 + x_3 \leq 5 \), which is an element of the IIS, is created by the RANGES section. The original constraint is con3, a \( \geq \) constraint with an RHS value of 4. If you choose to remove the constraint \( x_2 + x_3 \leq 5 \), you can accomplish this by removing con3 from the RANGES section in the MPS-format data table exiis. Since con3 is the only observation in the section, the identifier observation can also be removed. The modified LP problem is specified in the following SAS statements:

```sas
/* dropping con3, feasible */
data mycas.exiisf;
  input _id_ field1 $ field2 $ field3 $ field4 field5 $ field6;
datalines;
1 NAME . . . . .
2 ROWS . . . . .
3 N cost . . . .
4 G con1 . . . .
5 L con2 . . . .
6 G con3 . . . .
7 COLUMNS . . . .
8 . x1 cost 1 con1 1
9 . x1 con2 1 .
10 . x2 cost 1 con1 1
11 . x2 con3 1 .
12 . x3 cost 1 con2 1
13 . x3 con3 1 .
14 RHS . . . . .
15 . rhs con1 10 con2 4
16 . rhs con3 4 .
17 BOUNDS . . . . .
18 UP b1 x3 3 .
19 ENDTAS . . . . . ;
```
Since one element of the IIS has been removed, the modified LP problem should no longer contain the infeasible set. Due to the size of this problem, there should be no additional irreducible infeasible sets. You can confirm this by submitting the following SAS statements:

```sas
proc optlp data=mycas.exiisf
   iis=true;
run;
```

The notes shown in Output 6.7.3 are printed to the log.

**Output 6.7.3** The IIS= Option: Log

<table>
<thead>
<tr>
<th>Objective</th>
<th>Phase</th>
<th>Iteration</th>
<th>Value</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P 1</td>
<td>1</td>
<td>1.400000E+01</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>P 1</td>
<td>2</td>
<td>0.000000E+00</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The IIS= option found this problem to be feasible.
NOTE: The IIS solve time is 0.00 seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.379887 seconds.
NOTE: The data set WORK.EXSS has 8 observations and 4 variables.

The solution summary is displayed in Output 6.7.4.

**Output 6.7.4** Infeasibility Removed

**Solution Summary**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name1</th>
<th>Label1</th>
<th>cValue1</th>
<th>nValue1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>solver</td>
<td>Solver</td>
<td>LP</td>
<td>.</td>
</tr>
<tr>
<td>2</td>
<td>algorithm</td>
<td>Algorithm</td>
<td>IIS</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>objectiveName</td>
<td>Objective Function</td>
<td>cost</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>solStatus</td>
<td>Solution Status</td>
<td>Feasible</td>
<td>.</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>iterations</td>
<td>Iterations</td>
<td>2</td>
<td>2.000000</td>
</tr>
<tr>
<td>7</td>
<td>presolveTime</td>
<td>Presolve Time</td>
<td>0.00</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>solutionTime</td>
<td>Solution Time</td>
<td>0.00</td>
<td>0.000378</td>
</tr>
</tbody>
</table>

**Example 6.8: Using the Network Simplex Algorithm**

This example demonstrates how to use the network simplex algorithm to find the minimum-cost flow in a directed graph. Consider the directed graph in Figure 6.5, which appears in Ahuja, Magnanti, and Orlin (1993).
Example 6.8: Using the Network Simplex Algorithm

Figure 6.5 Minimum-Cost Network Flow Problem: Data

You can use the following SAS statements to create the input data table ex8:

```sas
data mycas.ex8;
  input _id_ field1 $8. field2 $13. @28 field3 $13. field4 @56 field5 $13. field6;
datalines;
1 NAME . . . . .
2 ROWS . . . . .
3 N obj . . . . .
4 E balance['1'] . . . .
5 E balance['2'] . . . .
6 E balance['3'] . . . .
7 E balance['4'] . . . .
8 E balance['5'] . . . .
9 E balance['6'] . . . .
10 E balance['7'] . . . .
11 E balance['8'] . . . .
12 COLUMNS . . . . .
13 . x['1','4'] obj 2 balance['1'] 1
14 . x['1','4'] balance['4'] -1 . .
15 . x['2','1'] obj 1 balance['1'] -1
16 . x['2','1'] balance['2'] 1 . .
17 . x['2','3'] balance['2'] 1 balance['3'] -1
18 . x['2','6'] obj 6 balance['2'] 1
19 . x['2','6'] balance['6'] -1 . .
20 . x['3','4'] obj 1 balance['3'] 1
21 . x['3','4'] balance['4'] -1 . .
22 . x['3','5'] obj 4 balance['3'] 1
23 . x['3','5'] balance['5'] -1 . .
24 . x['4','7'] obj 5 balance['4'] 1
25 . x['4','7'] balance['7'] -1 . .
26 . x['5','6'] obj 2 balance['5'] 1
27 . x['5','6'] balance['6'] -1 . .
28 . x['5','7'] obj 7 balance['5'] 1
29 . x['5','7'] balance['7'] -1 . .
30 . x['6','8'] obj 8 balance['6'] 1
31 . x['6','8'] balance['8'] -1 . .
```

You can use the following SAS statements to create the input data table ex8:
You can use the following call to PROC OPTLP to find the minimum-cost flow:

```plaintext
proc optlp
   presolver = none
   printlevel = 2
   logfreq = 1
   data = mycas.ex8
   primalout = mycas.ex8out
   algorithm = ns;
run;
```

The optimal solution is displayed in Output 6.8.1.

**Output 6.8.1** Network Simplex Algorithm: Primal Solution Output

### Primal Solution

<table>
<thead>
<tr>
<th>Obs</th>
<th>Objective Function ID</th>
<th>RHS ID Name</th>
<th>Variable Type</th>
<th>Objective Coefficient</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
<th>Variable Value</th>
<th>Variable Status</th>
<th>Reduced Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>obj</td>
<td>.RHS. x['1', '4'] D</td>
<td>D</td>
<td>2</td>
<td>0</td>
<td>15</td>
<td>10</td>
<td>B</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>obj</td>
<td>.RHS. x['2', '1'] D</td>
<td>D</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>L</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>obj</td>
<td>.RHS. x['2', '3'] D</td>
<td>D</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>obj</td>
<td>.RHS. x['2', '6'] D</td>
<td>D</td>
<td>6</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>obj</td>
<td>.RHS. x['3', '4'] D</td>
<td>D</td>
<td>1</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>obj</td>
<td>.RHS. x['3', '5'] D</td>
<td>D</td>
<td>4</td>
<td>0</td>
<td>10</td>
<td>5</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>obj</td>
<td>.RHS. x['4', '7'] D</td>
<td>D</td>
<td>5</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>U</td>
<td>-5</td>
</tr>
<tr>
<td>8</td>
<td>obj</td>
<td>.RHS. x['5', '6'] D</td>
<td>D</td>
<td>2</td>
<td>0</td>
<td>20</td>
<td>0</td>
<td>L</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>obj</td>
<td>.RHS. x['5', '7'] D</td>
<td>D</td>
<td>7</td>
<td>0</td>
<td>15</td>
<td>5</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>obj</td>
<td>.RHS. x['6', '8'] D</td>
<td>D</td>
<td>8</td>
<td>0</td>
<td>10</td>
<td>10</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>obj</td>
<td>.RHS. x['7', '8'] D</td>
<td>D</td>
<td>9</td>
<td>0</td>
<td>15</td>
<td>0</td>
<td>L</td>
<td>6</td>
</tr>
</tbody>
</table>
Example 6.8: Using the Network Simplex Algorithm

The optimal solution is represented graphically in Figure 6.6.

Figure 6.6 Minimum-Cost Network Flow Problem: Optimal Solution

The iteration log is displayed in Output 6.8.2.

Output 6.8.2 Log: Solution Progress

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Objective</th>
<th>Infeasibility</th>
<th>Infeasibility</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.000000E+00</td>
<td>2.000000E+01</td>
<td>8.900000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>0.000000E+00</td>
<td>2.000000E+01</td>
<td>8.900000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td>5.000000E+00</td>
<td>1.500000E+01</td>
<td>8.400000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>5.000000E+00</td>
<td>1.500000E+01</td>
<td>8.300000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>7.500000E+01</td>
<td>1.500000E+01</td>
<td>8.300000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>7.500000E+01</td>
<td>1.500000E+01</td>
<td>7.900000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>1.300000E+02</td>
<td>1.000000E+01</td>
<td>7.600000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>2.700000E+02</td>
<td>0.000000E+00</td>
<td>0.000000E+00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: The total Network Simplex solve time is 0.00 seconds.
NOTE: Optimal.
NOTE: Objective = 270.
NOTE: The Cloud Analytic Services server processed the request in 0.435991 seconds.
NOTE: The data set MYCAS.EX8OUT has 11 observations and 10 variables.
References


Chapter 7  
The OPTMILP Procedure

Overview: OPTMILP Procedure

The OPTMILP procedure solves general mixed integer linear programs (MILPs).

A standard mixed integer linear program has the formulation

\[
\begin{align*}
\min & \quad c^T x \\
\text{subject to} & \quad Ax \{\geq, =, \leq\} b \\
& \quad 1 \leq x \leq u \\
& \quad x_i \in \mathbb{Z} \quad \forall i \in S
\end{align*}
\]  

(MILP)
where

\[
\begin{align*}
\mathbf{x} &\in \mathbb{R}^n  & \text{is the vector of structural variables} \\
\mathbf{A} &\in \mathbb{R}^{m\times n}  & \text{is the matrix of technological coefficients} \\
\mathbf{c} &\in \mathbb{R}^n  & \text{is the vector of objective function coefficients} \\
\mathbf{b} &\in \mathbb{R}^m  & \text{is the vector of constraints’ right-hand sides (RHS)} \\
\mathbf{l} &\in \mathbb{R}^n  & \text{is the vector of lower bounds on variables} \\
\mathbf{u} &\in \mathbb{R}^n  & \text{is the vector of upper bounds on variables} \\
\mathcal{S} & & \text{is a nonempty subset of the set \{1 \ldots n\} of indices}
\end{align*}
\]

The OPTMILP procedure implements a linear-programming-based branch-and-cut algorithm. This divide-and-conquer approach attempts to solve the original problem by solving linear programming relaxations of a sequence of smaller subproblems. The OPTMILP procedure also implements advanced techniques such as presolving, generating cutting planes, and applying primal heuristics to improve the efficiency of the overall algorithm.

The OPTMILP procedure requires a mixed integer linear program to be specified using a data table that adheres to the mathematical programming system (MPS) format, a widely accepted format in the optimization community. Chapter 9 discusses the MPS format in detail. It is also possible to input an incumbent solution; see the section “Warm Start” on page 179 for details.

The OPTMILP procedure provides various control options and solution strategies. In particular, you can enable, disable, or set levels for the advanced techniques previously mentioned.

The OPTMILP procedure outputs an optimal solution or the best feasible solution found, if any, in data tables. This enables you to generate solution reports and perform additional analyses by using SAS software.

---

**Getting Started: OPTMILP Procedure**

The following example illustrates the use of the OPTMILP procedure to solve mixed integer linear programs. For more examples, see the section “Examples: OPTMILP Procedure” on page 194. Suppose you want to solve the following problem:

\[
\begin{align*}
\text{min} \quad & 2x_1 - 3x_2 - 4x_3 \\
\text{s.t.} \quad & -2x_2 - 3x_3 \geq -5 \quad (R1) \\
& x_1 + x_2 + 2x_3 \leq 4 \quad (R2) \\
& x_1 + 2x_2 + 3x_3 \leq 7 \quad (R3) \\
& x_1, x_2, x_3 \geq 0 \\
& x_1, x_2, x_3 \in \mathbb{Z}
\end{align*}
\]

The corresponding MPS-format data set is created as follows:

```sas
data ex_mip;
  input field1 $ field2 $ field3 $ field4 field5 $ field6 $;
  datalines;
NAME . EX_MIP . . . .
ROWS . . . . . .
N COST . . . .
```

---
To make the created data set available in the CAS server, you need to upload it. At the same time, you also need to add an _id_ column to ensure that the MPS reader can read the file in the correct order even if it is distributed across several machines and hence loses the original ordering. For more information about the MPS format, see Chapter 9, “The MPS-Format Data Table.”

```sas
data mycas.ex_mip;
  set ex_mip;
  _id_ = _n_; 
run;
```

Alternatively, you can create the MPS data table directly in CAS and include the _id_ column in the datalines input.

You can also create this data set from an MPS format flat file (ex_mip.mps) by using the following SAS macro:

```sas
%mps2sasd(mpsfile = "ex_mip.mps", outdata = mycas.ex_mip);
```

This problem can be solved by using the following statement to call the OPTMILP procedure:

```sas
proc optmilp data = mycas.ex_mip
  objsense = min
  primalout = mycas.primal_out
  dualout = mycas.dual_out
  presolver = automatic
  heuristics = automatic;
run;
```

The DATA= option names the MPS format data table that contains the problem data. The OBJSENSE= option specifies whether to maximize or minimize the objective function. The PRIMALOUT= option names the data table to contain the optimal solution or the best feasible solution found by the solver. The DUALOUT= option names the data table to contain the constraint activities. The PRESOLVER= and HEURISTICS= options specify the levels for presolving and applying heuristics, respectively. In this example, each option is set to its default value AUTOMATIC, meaning that the solver automatically determines the appropriate levels for presolve and heuristics.
The optimal integer solution and its corresponding constraint activities, stored in the data tables primal_out and dual_out, respectively, are displayed in Figure 7.1 and Figure 7.2.

**Figure 7.1** Optimal Solution

<table>
<thead>
<tr>
<th>Objective Function ID</th>
<th>RHS Name</th>
<th>Variable</th>
<th>Type</th>
<th>Objective Coefficient</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
<th>Variable Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 COST</td>
<td>RHS X1</td>
<td>B</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2 COST</td>
<td>RHS X2</td>
<td>B</td>
<td>-3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>3 COST</td>
<td>RHS X3</td>
<td>B</td>
<td>-4</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 7.2** Constraint Activities

<table>
<thead>
<tr>
<th>Objective Function ID</th>
<th>RHS Name</th>
<th>Constraint</th>
<th>Constraint Type</th>
<th>Constraint RHS</th>
<th>Constraint Lower Bound</th>
<th>Constraint Upper Bound</th>
<th>Constraint Activity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 COST</td>
<td>RHS R1</td>
<td>G</td>
<td>-5</td>
<td></td>
<td></td>
<td></td>
<td>-5</td>
</tr>
<tr>
<td>2 COST</td>
<td>RHS R2</td>
<td>L</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>3 COST</td>
<td>RHS R3</td>
<td>L</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>

The solution summary stored in the macro variable _OROPTMILP_ can be viewed by issuing the following statement:

```
%put &_OROPTMILP_;
```

This produces the output shown in Figure 7.3.

**Figure 7.3** Macro Output

```
STATUS=OK ALGORITHM=BAC SOLUTION_STATUS=OPTIMAL OBJECTIVE=-7 RELATIVE_GAP=0
ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0
INTEGER_INFEASIBILITY=0 BEST_BOUND=-7 NODES=0 ITERATIONS=0 PRESOLVE_TIME=0.00
SOLUTION_TIME=0.00
```

See the section “Data Input and Output” on page 177 for details about the type and status codes displayed for variables and constraints.
Syntax: OPTMILP Procedure

The following statements are available in the OPTMILP procedure:

```
PROC OPTMILP <options> ;
DECOMP <options> ;
DECOMPMaster <options> ;
DECOMPMasterIP <options> ;
DECOMPSUBPROB <options> ;
```

Functional Summary

Table 7.1 summarizes the options available for the OPTMILP procedure, classified by function.

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Table Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the input data table</td>
<td>DATA=</td>
</tr>
<tr>
<td>Specifies the constraint activities output data table</td>
<td>DUALOUT=</td>
</tr>
<tr>
<td>Specifies whether the MILP model is a maximization or minimization problem</td>
<td>OBJSENSE=</td>
</tr>
<tr>
<td>Specifies the primal solution input data table (warm start)</td>
<td>PRIMALIN=</td>
</tr>
<tr>
<td>Specifies the primal solution output data table</td>
<td>PRIMALOUT=</td>
</tr>
<tr>
<td><strong>Presolve Option</strong></td>
<td>PRESOLVER=</td>
</tr>
<tr>
<td>Specifies the type of presolve</td>
<td></td>
</tr>
<tr>
<td><strong>Control Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the stopping criterion based on absolute objective gap</td>
<td>ABSOBJGAP=</td>
</tr>
<tr>
<td>Specifies the cutoff value for node removal</td>
<td>CUTOFF=</td>
</tr>
<tr>
<td>Emphasizes feasibility or optimality</td>
<td>EMPHASIS=</td>
</tr>
<tr>
<td>Specifies the maximum violation on variables and constraints</td>
<td>FEASTOL=</td>
</tr>
<tr>
<td>Specifies the maximum allowed difference between an integer variable’s value and an integer</td>
<td>INTTOL=</td>
</tr>
<tr>
<td>Specifies the frequency of printing the node log</td>
<td>LOGFREQ=</td>
</tr>
<tr>
<td>Specifies the detail of solution progress printed in log</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the maximum number of nodes to be processed</td>
<td>MAXNODES=</td>
</tr>
<tr>
<td>Specifies the maximum number of solutions to be found</td>
<td>MAXSOLS=</td>
</tr>
<tr>
<td>Specifies the time limit for the optimization process</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Specifies the tolerance used in determining the optimality of nodes in the branch-and-bound tree</td>
<td>OPTTOL=</td>
</tr>
<tr>
<td>Toggles ODS output</td>
<td>PRINTLEVEL=</td>
</tr>
<tr>
<td>Specifies the probing level</td>
<td>PROBE=</td>
</tr>
<tr>
<td>Specifies the stopping criterion based on relative objective gap</td>
<td>RELOBJGAP=</td>
</tr>
<tr>
<td>Enables the use of scaling on the problem matrix</td>
<td>SCALE=</td>
</tr>
<tr>
<td>Specifies the initial seed for the random number generator</td>
<td>SEED=</td>
</tr>
<tr>
<td>Specifies the stopping criterion based on target objective value</td>
<td>TARGET=</td>
</tr>
<tr>
<td>Specifies whether time units are CPU time or real time</td>
<td>TIMETYPE=</td>
</tr>
</tbody>
</table>
Table 7.1  (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Heuristics Option</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the primal heuristics level</td>
<td>HEURISTICS=</td>
</tr>
<tr>
<td><strong>Search Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the level of conflict search</td>
<td>CONFLICTSEARCH=</td>
</tr>
<tr>
<td>Specifies the node selection strategy</td>
<td>NODESEL=</td>
</tr>
<tr>
<td>Enables use of variable priorities</td>
<td>PRIORITY=</td>
</tr>
<tr>
<td>Specifies the restarting strategy</td>
<td>RESTARTS=</td>
</tr>
<tr>
<td>Specifies the number of simplex iterations performed on each variable in the strong branching variable selection strategy</td>
<td>STRONGITER=</td>
</tr>
<tr>
<td>Specifies the number of candidates for the strong branching variable selection strategy</td>
<td>STRONGLEN=</td>
</tr>
<tr>
<td>Specifies the level of symmetry detection</td>
<td>SYMMETRY=</td>
</tr>
<tr>
<td>Specifies the rule for selecting the branching variable</td>
<td>VARSEL=</td>
</tr>
<tr>
<td><strong>Cut Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the cut level for all cuts</td>
<td>ALLCUTS=</td>
</tr>
<tr>
<td>Specifies the clique cut level</td>
<td>CUTCLIQUE=</td>
</tr>
<tr>
<td>Specifies the flow cover cut level</td>
<td>CUTFLOWCOVER=</td>
</tr>
<tr>
<td>Specifies the flow path cut level</td>
<td>CUTFLOWPATH=</td>
</tr>
<tr>
<td>Specifies the Gomory cut level</td>
<td>CUTGOMORY=</td>
</tr>
<tr>
<td>Specifies the generalized upper bound (GUB) cover cut level</td>
<td>CUTGUB=</td>
</tr>
<tr>
<td>Specifies the implied bounds cut level</td>
<td>CUTIMPLIED=</td>
</tr>
<tr>
<td>Specifies the knapsack cover cut level</td>
<td>CUTFKAPACK=</td>
</tr>
<tr>
<td>Specifies the lift-and-project cut level</td>
<td>CUTLAP=</td>
</tr>
<tr>
<td>Specifies the mixed lifted 0-1 cut level</td>
<td>CUTMILIFTED=</td>
</tr>
<tr>
<td>Specifies the mixed integer rounding (MIR) cut level</td>
<td>CUTMIR=</td>
</tr>
<tr>
<td>Specifies the multicommodity network flow cut level</td>
<td>CUTFMULTICOMMODITY=</td>
</tr>
<tr>
<td>Specifies the row multiplier factor for cuts</td>
<td>CUTSFACOR=</td>
</tr>
<tr>
<td>Specifies the overall cut aggressiveness</td>
<td>CUTFSTRATEGY=</td>
</tr>
<tr>
<td>Specifies the zero-half cut level</td>
<td>CUTZEROHALF=</td>
</tr>
<tr>
<td><strong>Parallel Options</strong></td>
<td></td>
</tr>
<tr>
<td>Enables PROC OPTMILP to run in concurrent mode</td>
<td>CONCURRENT=</td>
</tr>
<tr>
<td>Enables PROC OPTMILP to run deterministically</td>
<td>DETERMINISTIC=</td>
</tr>
<tr>
<td>Enables PROC OPTMILP to run in distributed mode</td>
<td>DISTRIBUTED=</td>
</tr>
<tr>
<td>Specifies the number of threads for the parallel OPTMILP procedure to use</td>
<td>NTHREADS=</td>
</tr>
</tbody>
</table>

**PROC OPTMILP Statement**

PROC OPTMILP <options> ;

You can specify the following options in the PROC OPTMILP statement.
Data Table Options

DATA=CAS-libref.data-table
specifies the input data table that corresponds to the MILP model. If this option is not specified, PROC OPTMILP uses the most recently created data table. See Chapter 9, “The MPS-Format Data Table,” for more details about the input data table. CAS-libref.data-table is a two-level name, where CAS-libref refers to the caslib and session identifier, and data-table specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 19 in Chapter 4, “Shared Concepts.”

DUALOUT=CAS-libref.data-table
DOUT=CAS-libref.data-table
specifies the output data table to contain the constraint activities. CAS-libref.data-table is a two-level name, where CAS-libref refers to the caslib and session identifier, and data-table specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 19 in Chapter 4, “Shared Concepts.”

OBJSENSE=MIN | MAX
specifies whether the MILP model is a minimization or a maximization problem. You can use OBJSENSE=MIN for a minimization problem and OBJSENSE=MAX for a maximization problem. Alternatively, you can specify the objective sense in the input data table. This option supersedes the objective sense specified in the input data table. If the objective sense is not specified anywhere, then PROC OPTMILP interprets and solves the MILP as a minimization problem.

PRIMALIN=CAS-libref.data-table
enables you to input a warm start solution in a data table. PROC OPTMILP validates both the data table and the solution stored in the data table. If the data table is not valid, then the PRIMALIN= data table is ignored. If the solution stored in a valid PRIMALIN= data table is a feasible integer solution, then it provides an incumbent solution and a bound for the branch-and-bound algorithm. If the solution stored in a valid PRIMALIN= data table is infeasible, contains missing values, or contains fractional values for integer variables, PROC OPTMILP tries to repair the solution with a number of specialized repair heuristics. See the section “Warm Start” on page 179 for details. CAS-libref.data-table is a two-level name, where CAS-libref refers to the caslib and session identifier, and data-table specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 19 in Chapter 4, “Shared Concepts.”

PRIMALOUT=CAS-libref.data-table
POUT=CAS-libref.data-table
specifies the output data table for the primal solution. This data table contains the primal solution information. See the section “Data Input and Output” on page 177 for details. CAS-libref.data-table is a two-level name, where CAS-libref refers to the caslib and session identifier, and data-table specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 19 in Chapter 4, “Shared Concepts.”
Presolve Option

\texttt{PRESOLVER=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE}

specifies a presolve level. You can specify the following values:

- **AUTOMATIC** applies the default level of presolve processing.
- **NONE** disables the presolver.
- **BASIC** performs minimal presolve processing.
- **MODERATE** applies a higher level of presolve processing.
- **AGGRESSIVE** applies the highest level of presolve processing.

By default, \texttt{PRESOLVER=AUTOMATIC}.

Control Options

\texttt{ABSOBJGAP=number}

\texttt{ABSOLUTEOBJECTIVEGAP=number}

specifies a stopping criterion. When the absolute difference between the best integer objective and the best bound on the objective function value becomes smaller than the value of \texttt{number}, the procedure stops. The value of \texttt{number} can be any nonnegative number; the default value is 1E–6.

\texttt{CUTOFF=number}

cuts off any nodes in a minimization (maximization) problem that have an objective value at or above (below) \texttt{number}. The value of \texttt{number} can be any number; the default value is the largest (smallest) number that can be represented by a double.

\texttt{EMPHASIS=BALANCE | OPTIMAL | FEASIBLE}

specifies the type of search emphasis. You can specify the following values:

- **BALANCE** performs a balanced search.
- **OPTIMAL** emphasizes optimality over feasibility.
- **FEASIBLE** emphasizes feasibility over optimality.

By default, \texttt{EMPHASIS=BALANCE}.

\texttt{FEASTOL=number}

specifies the tolerance that PROC OPTMILP uses to check the feasibility of a solution. This tolerance applies both to the maximum violation of bounds on variables and to the difference between the right-hand sides and left-hand sides of constraints. The value of \texttt{number} can be any value between 1E–4 and 1E–9, inclusive. However, the value of \texttt{number} cannot be larger than the integer feasibility tolerance. If the value of \texttt{number} is larger than the value of the \texttt{INTTOL=} option, then PROC OPTMILP sets \texttt{FEASTOL=} to the value of \texttt{INTTOL=}.

The default value is 1E–6.

If PROC OPTMILP fails to find a feasible solution within this tolerance but does find a solution that has some violation, then the procedure stops with a solution status of OPTIMAL_COND (see the section “Macro Variable _OROPTMILP_” on page 191).
INTTOL=number

INTEGERTOLERANCE=number

specifies the amount by which an integer variable value can differ from an integer and still be considered integer feasible. The value of number can be any number between 1E–9 and 0.5, inclusive. PROC OPTMILP attempts to find an optimal solution whose integer infeasibility is less than number. The default value is 1E–5.

If the best solution that PROC OPTMILP finds has an integer infeasibility larger than the value of number, then PROC OPTMILP stops with a solution status of OPTIMAL_COND (see the section “Macro Variable _OROPTMILP_” on page 191).

LOGFREQ=k

PRINTFREQ=k

prints information in the node log every k seconds, where k is any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If k=0, then the node log is disabled. If k is positive, then the root node processing information is printed and, if possible, an entry is made every k seconds. An entry is also made each time a better integer solution is found.

By default, LOGFREQ=5.

LOGLEVEL=NONE | BASIC | MODERATE | AGGRESSIVE

controls the amount of information displayed in the SAS log by the solver. You can specify the following values:

NONE turns off all solver-related messages in the SAS log.

BASIC displays a solver summary after stopping.

MODERATE prints a solver summary and a node log at the interval specified in the LOGFREQ= option.

AGGRESSIVE prints a detailed solver summary and a node log at the interval specified in the LOGFREQ= option.

By default, LOGLEVEL=MODERATE.

MAXNODES=number

specifies the maximum number of branch-and-bound nodes to be processed, where number can be any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value is $2^{31} - 1$. If you run PROC OPTMILP in concurrent mode, then the procedure stops as soon as number is reached on any machine. If you run PROC OPTMILP in distributed mode, then the procedure periodically checks and stops as soon as the total number of nodes that are processed by all grid nodes exceeds number.

MAXSOLS=number

specifies a stopping criterion, where number can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If number of solutions have been found, then the procedure stops. The default value of number is $2^{31} - 1$. 
MAXTIME=t
specifies an upper limit of t seconds of time for reading in the data and performing the optimization process. The value of the TIMETYPE= option determines the type of units used. If you do not specify MAXTIME= option, the procedure does not stop based on the amount of time elapsed. If concurrent or distributed mode of PROC OPTMILP is enabled, then the procedure stops as soon as t is reached on any machine. The value of t can be any positive number; the default value is the largest number that can be represented by a double.

OPTTOL=number
specifies the tolerance that is used to determine the optimality of nodes in the branch-and-bound tree. The value of number can be any value between (and including) 1E–4 and 1E–9. The default value is 1E–6.

PRINTLEVEL=0 | 1 | 2
specifies whether to print a summary of the problem and solution. You can specify the following values:

0 does not produce or print any Output Delivery System (ODS) tables.
1 produces and prints the following ODS tables: ProblemSummary and SolutionSummary.
2 produces and prints the following ODS tables: ProblemSummary, SolutionSummary, ProblemStatistics, and Timing table.

By default, PRINTLEVEL=1. For more information about the ODS tables created by PROC OPTMILP, see the section “ODS Tables” on page 186.

PROBE=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the probing strategy. You can specify the following values:

AUTOMATIC uses the probing strategy that is determined by PROC OPTMILP.
NONE disables probing.
MODERATE uses probing moderately.
AGGRESSIVE uses probing aggressively.

By default, PROBE=AUTOMATIC. For more information, see the section “Presolve and Probing” on page 182.

RELOBJGAP=number
specifies a stopping criterion based on the best integer objective (BestInteger) and the best bound on the objective function value (BestBound). The relative objective gap is equal to

$$\frac{|\text{BestInteger} - \text{BestBound}|}{(1E-10 + |\text{BestBound}|)}$$

When this value becomes smaller than the specified gap size number, the procedure stops. The value of number can be any nonnegative number; the default value is 1E–4.
SCALE=AUTOMATIC | NONE
indicates whether to scale the problem matrix. You can specify the following values:

AUTOMATIC scales the matrix as determined by PROC OPTMILP.
NONE disables scaling.

By default, SCALE=AUTOMATIC.

SEED=number
specifies the initial seed of the random number generator. This option affects the perturbation in the simplex solvers; thus it might result in a different optimal solution and a different solver path. This option usually has a significant, but unpredictable, effect on the solution time. The value of number can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value of the seed is 100.

TARGET=number
specifies a stopping criterion for a minimization or maximization problem. If the best integer objective is better than or equal to number, the procedure stops. The value of number can be any number; the default value is the largest (in magnitude) negative number (for a minimization problem) or the largest (in magnitude) positive number (for a maximization problem) that can be represented by a double.

TIMETYPE=CPU | REAL
specifies whether CPU time or real time is used for the MAXTIME= option and the _OROPTMILP_ macro variable in a PROC OPTMILP call. You can specify the following values:

CPU specifies that units are in CPU time.
REAL specifies that units are in real time.

By default, TIMETYPE=REAL.

Heuristics Option

HEURISTICS=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE
controls the level of primal heuristics applied by PROC OPTMILP. This level determines how frequently primal heuristics are applied during the branch-and-bound tree search. It also affects the maximum number of iterations allowed in iterative heuristics. Some computationally expensive heuristics might be disabled by the solver at less aggressive levels. You can specify the following values:

AUTOMATIC applies the default level of heuristics, similar to MODERATE.
NONE disables all primal heuristics. This value does not disable the heuristics that repair an infeasible input solution that is specified in a PRIMALIN= data table.
BASIC applies basic primal heuristics at low frequency.
MODERATE applies most primal heuristics at moderate frequency.
AGGRESSIVE applies all primal heuristics at high frequency.

By default, HEURISTICS=AUTOMATIC. For more information about primal heuristics, see the section “Primal Heuristics” on page 184.
Search Options

CONFLICTSEARCH=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the level of conflict search performed by PROC OPTMILP. Conflict search is used to find clauses resulting from infeasible subproblems that arise in the search tree. You can specify the following values:

- AUTOMATIC performs conflict search based on a strategy that is determined by PROC OPTMILP.
- NONE disables conflict search.
- MODERATE performs a moderate conflict search.
- AGGRESSIVE performs an aggressive conflict search.

By default, CONFLICTSEARCH=AUTOMATIC.

NODESEL=AUTOMATIC | BESTBOUND | BESTESTIMATE | DEPTH
specifies the node selection strategy. You can specify the following values:

- AUTOMATIC uses automatic node selection.
- BESTBOUND chooses the node with the best relaxed objective (best-bound-first strategy).
- BESTESTIMATE chooses the node with the best estimate of the integer objective value (best-estimate-first strategy).
- DEPTH chooses the most recently created node (depth-first strategy).

By default, NODESEL=AUTOMATIC. For more information about node selection, see the section “Node Selection” on page 181.

PRIORITY=TRUE | FALSE
indicates whether to use specified branching priorities for integer variables. You can specify the following values:

- TRUE uses priorities when they exist.
- FALSE ignores variable priorities.

By default, PRIORITY=TRUE. For more information, see the section “Branching Priorities” on page 182.

RESTARTS=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE
specifies the strategy for restarting the processing of the root node. You can specify the following values:

- AUTOMATIC uses a restarting strategy determined by PROC OPTMILP.
- NONE disables restarting.
- BASIC uses a basic restarting strategy.
- MODERATE uses a moderate restarting strategy.
- AGGRESSIVE uses an aggressive restarting strategy.

By default, RESTARTS=AUTOMATIC.
**STRONGITER=** *number | AUTOMATIC*

specifies the number of simplex iterations performed for each variable in the candidate list when using the strong branching variable selection strategy. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC, PROC OPTMILP uses the default value; this value is calculated automatically.

**STRONGLEN=** *number | AUTOMATIC*

specifies the number of candidates used when performing the strong branching variable selection strategy. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC, PROC OPTMILP uses the default value; this value is calculated automatically.

**SYMMETRY=** *AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE*

specifies the level of symmetry detection. Symmetry detection identifies groups of equivalent decision variables and uses this information to solve the problem more efficiently. You can specify the following values:

- AUTOMATIC performs symmetry detection based on a strategy that is determined by PROC OPTMILP.
- NONE disables symmetry detection.
- BASIC performs a basic symmetry detection.
- MODERATE performs a moderate symmetry detection.
- AGGRESSIVE performs an aggressive symmetry detection.

By default, SYMMETRY=AUTOMATIC. For more information about symmetry detection, see (Ostrowski 2008).

**VARSEL=** *AUTOMATIC | MAXINFEAS | MININFEAS | PSEUDO | STRONG*

specifies the rule for selecting the branching variable. You can specify the following values:

- AUTOMATIC uses automatic branching variable selection.
- MAXINFEAS chooses the variable with maximum infeasibility.
- MININFEAS chooses the variable with minimum infeasibility.
- PSEUDO chooses a branching variable based on pseudocost.
- STRONG uses a strong branching variable selection strategy.

By default, VARSEL=AUTOMATIC. For details about variable selection, see the section “Variable Selection” on page 181.

**Cut Options**

Table 7.2 describes the *string* values for the cut options in PROC OPTMILP.
You can specify the `CUTSTRATEGY=` option to set the overall aggressiveness of the cut generation in PROC OPTMILP. Alternatively, you can use the `ALLCUTS=` option to set all cut types to the same level. You can override the `ALLCUTS=` value by using the options that correspond to particular cut types. For example, if you want PROC OPTMILP to generate only Gomory cuts, specify `ALLCUTS=NONE` and `CUTGOMORY=AUTOMATIC`. If you want to generate all cuts aggressively but generate no lift-and-project cuts, set `ALLCUTS=AGGRESSIVE` and `CUTLAP=NONE`.

**Table 7.2**  Values for Individual Cut Options

<table>
<thead>
<tr>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOMATIC</td>
<td>Generates cutting planes based on a strategy</td>
</tr>
<tr>
<td></td>
<td>determined by PROC OPTMILP</td>
</tr>
<tr>
<td>NONE</td>
<td>Disables generation of cutting planes</td>
</tr>
<tr>
<td>MODERATE</td>
<td>Uses a moderate cut strategy</td>
</tr>
<tr>
<td>AGGRESSIVE</td>
<td>Uses an aggressive cut strategy</td>
</tr>
</tbody>
</table>

**ALLCUTS=AUTOMATIC | NONE | MODERATE | AGGRESSIVE**

provides a shorthand way of setting all the cuts-related options in one setting. In other words, `ALLCUTS=string` is equivalent to setting each of the individual cuts parameters to the same value `string`. Thus, `ALLCUTS=AUTOMATIC` has the effect of setting `CUTCLIQUE=AUTOMATIC`, `CUTFLOWCOVER=AUTOMATIC`, . . . , and `CUTZEROHALF=AUTOMATIC`. Table 7.2 lists the values that can be assigned to `string`. In addition, you can override levels for individual cuts with the `CUTCLIQUE=`, `CUTFLOWCOVER=`, `CUTFLOWPATH=`, `CUTGOMORY=`, `CUTGUB=`, `CUTIMPLIED=`, `CUTKNAPSACK=`, `CUTLAP=`, `CUTMILIFTED=`, `CUTMIR=`, `CUTMULTICOMMODITY=`, and `CUTZEROHALF=` options. If the `ALLCUTS=` option is not specified, all the cuts-related options are either set to their individually specified values (if the corresponding option is specified) or to their default values (if that option is not specified).

**CUTCLIQUE=AUTOMATIC | NONE | MODERATE | AGGRESSIVE**

specifies the level of clique cuts generated by PROC OPTMILP. Table 7.2 describes the possible values. This option overrides the `ALLCUTS=` option. By default, `CUTCLIQUE=AUTOMATIC`.

**CUTFLOWCOVER=AUTOMATIC | NONE | MODERATE | AGGRESSIVE**

specifies the level of flow cover cuts generated by PROC OPTMILP. Table 7.2 describes the possible values. This option overrides the `ALLCUTS=` option. By default, `CUTFLOWCOVER=AUTOMATIC`.

**CUTFLOWPATH=AUTOMATIC | NONE | MODERATE | AGGRESSIVE**

specifies the level of flow path cuts generated by PROC OPTMILP. Table 7.2 describes the possible values. This option overrides the `ALLCUTS=` option. By default, `CUTFLOWPATH=AUTOMATIC`.

**CUTGOMORY=AUTOMATIC | NONE | MODERATE | AGGRESSIVE**

specifies the level of Gomory cuts generated by PROC OPTMILP. Table 7.2 describes the possible values. This option overrides the `ALLCUTS=` option. By default, `CUTGOMORY=AUTOMATIC`.

**CUTGUB=AUTOMATIC | NONE | MODERATE | AGGRESSIVE**

specifies the level of generalized upper bound (GUB) cover cuts generated by PROC OPTMILP. Table 7.2 describes the possible values. This option overrides the `ALLCUTS=` option. By default, `CUTGUB=AUTOMATIC`. 
**PROC OPTMILP Statement**

- **CUTIMPLIED=**
  - AUTOMATIC | NONE | MODERATE | AGGRESSIVE
  - Specifies the level of implied bound cuts generated by PROC OPTMILP. Table 7.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTIMPLIED=AUTOMATIC.

- **CUTKNAPSACK=**
  - AUTOMATIC | NONE | MODERATE | AGGRESSIVE
  - Specifies the level of knapsack cover cuts generated by PROC OPTMILP. Table 7.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTKNAPSACK=AUTOMATIC.

- **CUTLAP=**
  - AUTOMATIC | NONE | MODERATE | AGGRESSIVE
  - Specifies the level of lift-and-project (LAP) cuts generated by PROC OPTMILP. Table 7.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTLAP=NONE.

- **CUTMILIFTED=**
  - AUTOMATIC | NONE | MODERATE | AGGRESSIVE
  - Specifies the level of mixed lifted 0-1 cuts that are generated by PROC OPTMILP. Table 7.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTMILIFTED=AUTOMATIC.

- **CUTMIR=**
  - AUTOMATIC | NONE | MODERATE | AGGRESSIVE
  - Specifies the level of mixed integer rounding (MIR) cuts generated by PROC OPTMILP. Table 7.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTMIR=AUTOMATIC.

- **CUTMULTICOMMODITY=**
  - AUTOMATIC | NONE | MODERATE | AGGRESSIVE
  - Specifies the level of multicommodity network flow cuts generated by PROC OPTMILP. Table 7.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTMULTICOMMODITY=AUTOMATIC.

- **CUTSFACOR=**
  - number
  - Specifies a row multiplier factor for cuts. The number of cuts that are added is limited to number times the original number of rows. The value of number can be any nonnegative number less than or equal to 100; the default value is automatically calculated by PROC OPTMILP.

- **CUTSTRATEGY=**
  - AUTOMATIC | NONE | MODERATE | AGGRESSIVE
  - Specifies the overall aggressiveness of the cut generation in the procedure. By default, CUTSTRATEGY=AUTOMATIC. Setting a nondefault value adjusts a number of cut parameters such that the cut generation is none, moderate, or aggressive compared to the default value.

- **CUTZEROHALF=**
  - AUTOMATIC | NONE | MODERATE | AGGRESSIVE
  - Specifies the level of zero-half cuts that are generated by PROC OPTMILP. Table 7.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTZEROHALF=AUTOMATIC.

**Parallel Options**

- **CONCURRENT=**
  - TRUE | FALSE
  - Specifies whether to enable concurrent mode for PROC OPTMILP. You can specify the following values:
TRUE runs PROC OPTMILP in concurrent mode. The solver automatically runs the same problem on each grid node with different PROC OPTMILP options.

FALSE runs default PROC OPTMILP.

By default, CONCURRENT=FALSE.

DETERMINISTIC=TRUE | FALSE specifies whether to enable PROC OPTMILP to run deterministically when distributed or concurrent mode is enabled. The DETERMINISTIC= option is valid only when the DISTRIBUTED= or CONCURRENT= option is specified. You can specify the following values:

TRUE runs PROC OPTMILP in deterministic distributed or concurrent mode. The procedure always returns the same solution for subsequent runs on the same grid.

FALSE runs PROC OPTMILP in nondeterministic distributed or concurrent mode.

By default, DETERMINISTIC=FALSE when you specify the CONCURRENT= option, and DETERMINISTIC=TRUE when you specify the DISTRIBUTED= option.

DISTRIBUTED=TRUE | FALSE specifies whether to run PROC OPTMILP in distributed mode. For more information about this option as it relates to the standard PROC OPTMILP, see the section “Parallel Processing” on page 184. For more information about this option as it relates to the decomposition algorithm, see Chapter 18, “The Decomposition Algorithm.” You can specify the following values:

TRUE runs PROC OPTMILP in distributed mode.

FALSE turns off distributed mode of PROC OPTMILP.

By default, DISTRIBUTED=FALSE.

NTHREADS=number specifies the maximum number of threads to use for multithreaded processing. The branch-and-cut algorithm can take advantage of multicore machines and can potentially run faster when number is greater than 1. The value of number can be any integer between 1 and 256, inclusive. The default is the number of cores on the machine that executes the process or the number of cores permissible based on your installation (whichever is less). The number of simultaneously active CPUs is limited by your installation and license configuration.

Decomposition Algorithm Statements

The following statements are available for the decomposition algorithm in the OPTMILP procedure:

DECOMP <options> ;
DECOMPMASTER <options> ;
DECOMPMASTERIP <options> ;
DECOMPSUBPROB <options> ;

For more information about these statements, see Chapter 18, “The Decomposition Algorithm.”
Details: OPTMILP Procedure

Data Input and Output

This subsection describes the PRIMALIN= data table required to warm start PROC OPTMILP, in addition to the PRIMALOUT= and DUALOUT= data tables.

Definitions of Variables in the PRIMALIN= Data Table

The PRIMALIN= data table has two required variables defined as follows:

_VAR_
  specifies the variable (column) names of the problem. The values should match the column names in the DATA= data table for the current problem.

_VALUE_
  specifies the solution value for each variable in the problem.

NOTE: If PROC OPTMILP produces a feasible solution, the primal output data table from that run can be used as the PRIMALIN= data table for a subsequent run, provided that the variable names are the same. If this input solution is not feasible for the subsequent run, the solver automatically tries to repair it. See the section “Warm Start” on page 179 for more details.

Definitions of Variables in the PRIMALOUT= Data Table

PROC OPTMILP stores the current best integer feasible solution of the problem in the data table specified by the PRIMALOUT= option. The variables in this data table are defined as follows:

_OBJ_ID_
  specifies the identifier of the objective function.

_RHS_ID_
  specifies the identifier of the right-hand side.

_VAR_
  specifies the variable (column) names.

_TYPE_
  specifies the variable type. _TYPE_ can take one of the following values:

  C  continuous variable
  I  general integer variable
  B  binary variable (0 or 1)
Chapter 7: The OPTMILP Procedure

_OBJCOEF_
specifies the coefficient of the variable in the objective function.

_LBOUND_
specifies the lower bound on the variable.

_UBOUND_
specifies the upper bound on the variable.

_VALUE_
specifies the value of the variable in the current solution.

Definitions of Variables in the DUALOUT= Data Table

The DUALOUT= data table contains the constraint activities that correspond to the primal solution in the PRIMALOUT= data table. Information about additional objective rows of the MILP problem is not included. The variables in this data table are defined as follows:

_OBJ_ID_
specifies the identifier of the objective function from the input data table.

_RHS_ID_
specifies the identifier of the right-hand side from the input data table.

_ROW_
specifies the constraint (row) name.

_TYPE_
specifies the constraint type. _TYPE_ can take one of the following values:

L  “less than or equal” constraint
E  equality constraint
G  “greater than or equal” constraint
R  ranged constraint (both “less than or equal” and “greater than or equal”)

_RHS_
specifies the value of the right-hand side of the constraint. It takes a missing value for a ranged constraint.

_L_RHS_
specifies the lower bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

_U_RHS_
specifies the upper bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

_ACTIVITY_
specifies the activity of a constraint for a given primal solution. In other words, the value of _ACTIVITY_ for the \( i \)th constraint is equal to \( a_i^T x \), where \( a_i \) refers to the \( i \)th row of the constraint matrix and \( x \) denotes the vector of the current primal solution.
Warm Start

PROC OPTMILP enables you to input a warm start solution by using the PRIMALIN= option. PROC OPTMILP checks that the decision variables named in _VAR_ are the same as those in the MPS-format data table. If they are not the same, PROC OPTMILP issues a warning and ignores the input solution. PROC OPTMILP also checks whether the solution is infeasible, contains missing values, or contains fractional values for integer variables. If this is the case, PROC OPTMILP attempts to repair the solution with a number of specialized repair heuristics. The success of the attempt largely depends both on the specific model and on the proximity between the input solution and an integer feasible solution. An infeasible input solution can be considered a hint for PROC OPTMILP that might or might not help to solve the problem.

An integer feasible or repaired input solution provides an incumbent solution in addition to an upper (min) or lower (max) bound for the branch-and-bound algorithm. PROC OPTMILP uses the input solution to reduce the search space and to guide the search process. When it is difficult to find a good integer feasible solution for a problem, warm start can reduce solution time significantly.

Branch-and-Bound Algorithm

The branch-and-bound algorithm, first proposed by Land and Doig (1960), is an effective approach to solving mixed integer linear programs. The following discussion outlines the approach and explains how PROC OPTMILP enhances the basic algorithm by using several advanced techniques.

The branch-and-bound algorithm solves a mixed integer linear program by dividing the search space and generating a sequence of subproblems. The search space of a mixed integer linear program can be represented by a tree. Each node in the tree is identified with a subproblem derived from previous subproblems on the path that leads to the root of the tree. The subproblem (MILP0) associated with the root is identical to the original problem, which is called (MILP), given in the section “Overview: OPTMILP Procedure” on page 161.

The linear programming relaxation (LP0) of (MILP0) can be written as

\[
\begin{align*}
\min & \quad c^T x \\
\text{subject to} & \quad Ax \{\geq, =, \leq\} b \\
& \quad 1 \leq x \leq u
\end{align*}
\]

The branch-and-bound algorithm generates subproblems along the nodes of the tree by using the following scheme. Consider \(\hat{x}^0\), the optimal solution to (LP0), which is usually obtained by using the dual simplex algorithm. If \(\hat{x}^0_i\) is an integer for all \(i \in S\), then \(\hat{x}^0\) is an optimal solution to (MILP). Suppose that for some \(i \in S\), \(\hat{x}^0_i\) is nonintegral. In that case the algorithm defines two new subproblems (MILP1) and (MILP2), descendants of the parent subproblem (MILP0). The subproblem (MILP1) is identical to (MILP0) except for the additional constraint

\[ x_i \leq \lfloor \hat{x}_i^0 \rfloor \]

and the subproblem (MILP2) is identical to (MILP0) except for the additional constraint

\[ x_i \geq \lceil \hat{x}_i^0 \rceil \]
The notation \([y]\) represents the largest integer that is less than or equal to \(y\), and the notation \([y]\) represents the smallest integer that is greater than or equal to \(y\). The two preceding constraints can be handled by modifying the bounds of the variable \(x_i\) rather than by explicitly adding the constraints to the constraint matrix. The two new subproblems do not have \(\tilde{x}^0\) as a feasible solution, but the integer solution to (MILP) must satisfy one of the preceding constraints. The two subproblems thus defined are called active nodes in the branch-and-bound tree, and the variable \(x_i\) is called the branching variable.

In the next step the branch-and-bound algorithm chooses one of the active nodes and attempts to solve the linear programming relaxation of that subproblem. The relaxation might be infeasible, in which case the subproblem is dropped (fathomed). If the subproblem can be solved and the solution is integer feasible (that is, \(x_i\) is an integer for all \(i \in S\)), then its objective value provides an upper bound for the objective value in the minimization problem (MILP); if the solution is not integer feasible, then it defines two new subproblems. Branching continues in this manner until there are no active nodes. At this point the best integer solution found is an optimal solution for (MILP). If no integer solution has been found, then (MILP) is integer infeasible. You can specify other criteria to stop the branch-and-bound algorithm before it processes all the active nodes; see the section “Controlling the Branch-and-Bound Algorithm” on page 180 for details.

Upper bounds from integer feasible solutions can be used to fathom or cut off active nodes. Since the objective value of an optimal solution cannot be greater than an upper bound, active nodes with lower bounds higher than an existing upper bound can be safely deleted. In particular, if \(z\) is the objective value of the current best integer solution, then any active subproblems whose relaxed objective value is greater than or equal to \(z\) can be discarded.

It is important to realize that mixed integer linear programs are nondeterministic polynomial-time hard (NP-hard). Roughly speaking, this means that the effort required to solve a mixed integer linear program grows exponentially with the size of the problem. For example, a problem with 10 binary variables can generate in the worst case \(2^{10} = 1,024\) nodes in the branch-and-bound tree. A problem with 20 binary variables can generate in the worst case \(2^{20} = 1,048,576\) nodes in the branch-and-bound tree. Although it is unlikely that the branch-and-bound algorithm has to generate every single possible node, the need to explore even a small fraction of the potential number of nodes for a large problem can be resource-intensive.

A number of techniques can speed up the search progress of the branch-and-bound algorithm. Heuristics are used to find feasible solutions, which can improve the upper bounds on solutions of mixed integer linear programs. Cutting planes can reduce the search space and thus improve the lower bounds on solutions of mixed integer linear programs. When using cutting planes, the branch-and-bound algorithm is also called the branch-and-cut algorithm. Preprocessing can reduce problem size and improve problem solvability. PROC OPTMILP employs various heuristics, cutting planes, preprocessing, and other techniques, which you can control through corresponding options.

**Controlling the Branch-and-Bound Algorithm**

There are numerous strategies that can be used to control the branch-and-bound search (see Linderoth and Savelsbergh 1998, Achterberg, Koch, and Martin 2005). PROC OPTMILP implements the most widely used strategies and provides several options that enable you to direct the choice of the next active node and of the branching variable. In the discussion that follows, let \((LP^k)\) be the linear programming relaxation of subproblem \((MILP^k)\). Also, let

\[ f_i(k) = \tilde{x}_i^k - [\tilde{x}_i^k] \]

where \(\tilde{x}^k\) is the optimal solution to the relaxation problem \((LP^k)\) solved at node \(k\).
Node Selection

The **NODESEL=** option specifies the strategy used to select the next active node. The valid keywords for this option are AUTOMATIC, BESTBOUND, BESTESTIMATE, and DEPTH. The following list describes the strategy associated with each keyword:

- **AUTOMATIC** allows PROC OPTMILP to choose the best node selection strategy based on problem characteristics and search progress. This is the default setting.

- **BESTBOUND** chooses the node with the smallest (or largest, in the case of a maximization problem) relaxed objective value. The best-bound strategy tends to reduce the number of nodes to be processed and can improve lower bounds quickly. However, if there is no good upper bound, the number of active nodes can be large. This can result in the solver running out of memory.

- **BESTESTIMATE** chooses the node with the smallest (or largest, in the case of a maximization problem) objective value of the estimated integer solution. Besides improving lower bounds, the best-estimate strategy also attempts to process nodes that can yield good feasible solutions.

- **DEPTH** chooses the node that is deepest in the search tree. Depth-first search is effective in locating feasible solutions, since such solutions are usually deep in the search tree. Compared to the costs of the best-bound and best-estimate strategies, the cost of solving LP relaxations is less in the depth-first strategy. The number of active nodes is generally small, but it is possible that the depth-first search will remain in a portion of the search tree with no good integer solutions. This occurrence is computationally expensive.

Variable Selection

The **VARSEL=** option specifies the strategy used to select the next branching variable. The valid keywords for this option are AUTOMATIC, MAXINFEAS, MININFEAS, PSEUDO, and STRONG. The following list describes the action taken in each case when $\tilde{x}^k$, a relaxed optimal solution of $(\text{MILP}^k)$, is used to define two active subproblems. In the following list, “INTTOL” refers to the value assigned using the **INTTOL=** option. For details about the **INTTOL=** option, see the section “Control Options” on page 168.

- **AUTOMATIC** enables PROC OPTMILP to choose the best variable selection strategy based on problem characteristics and search progress. This is the default setting.

- **MAXINFEAS** chooses as the branching variable the variable $x_i$ such that $i$ maximizes

$$\{\min\{f_i(k), 1 - f_i(k)\} \mid i \in S\text{ and } \text{INTTOL} \leq f_i(k) \leq 1 - \text{INTTOL}\}$$

- **MININFEAS** chooses as the branching variable the variable $x_i$ such that $i$ minimizes

$$\{\min\{f_i(k), 1 - f_i(k)\} \mid i \in S\text{ and } \text{INTTOL} \leq f_i(k) \leq 1 - \text{INTTOL}\}$$
PSEUDO chooses as the branching variable the variable $x_i$ such that $i$ maximizes the weighted up and down pseudocosts. Pseudocost branching attempts to branch on significant variables first, quickly improving lower bounds. Pseudocost branching estimates significance based on historical information; however, this approach might not be accurate for future search.

STRONG chooses as the branching variable the variable $x_i$ such that $i$ maximizes the estimated improvement in the objective value. Strong branching first generates a list of candidates, then branches on each candidate and records the improvement in the objective value. The candidate with the largest improvement is chosen as the branching variable. Strong branching can be effective for combinatorial problems, but it is usually computationally expensive.

**Branching Priorities**

In some cases, it is possible to speed up the branch-and-bound algorithm by branching on variables in a specific order. You can accomplish this in PROC OPTMILP by attaching branching priorities to the integer variables in your model.

You can set branching priorities for use by PROC OPTMILP in two ways. You can specify the branching priorities directly in the input MPS-format data table; see the section “BRANCH Section (Optional)” on page 262 for details. If you are constructing a model in PROC OPTMODEL, you can set branching priorities for integer variables by using the .priority suffix. More information about this suffix is available in the section “Integer Variable Suffixes” on page 384 in Chapter 10. For an example in which branching priorities are used, see Example 14.3.

**Presolve and Probing**

PROC OPTMILP includes a variety of presolve techniques to reduce problem size, improve numerical stability, and detect infeasibility or unboundedness (Andersen and Andersen 1995; Gondzio 1997). During presolve, redundant constraints and variables are identified and removed. Presolve can further reduce the problem size by substituting variables. Variable substitution is a very effective technique, but it might occasionally increase the number of nonzero entries in the constraint matrix. Presolve might also modify the constraint coefficients to tighten the formulation of the problem.

In most cases, using presolve is very helpful in reducing solution times. You can enable presolve at different levels by specifying the PRESOLVER= option.

Probing is a technique that tentatively sets each binary variable to 0 or 1, then explores the logical consequences (Savelsbergh 1994). Probing can expedite the solution of a difficult problem by fixing variables and improving the model. However, probing is often computationally expensive and can significantly increase the solution time in some cases. You can enable probing at different levels by specifying the PROBE= option.

**Cutting Planes**

The feasible region of every linear program forms a polyhedron. Every polyhedron in $n$-space can be written as a finite number of half-spaces (equivalently, inequalities). In the notation used in this chapter, this polyhedron is defined by the set $Q = \{x \in \mathbb{R}^n | Ax \leq b, l \leq x \leq u\}$. After you add the restriction that
some variables must be integral, the set of feasible solutions, $\mathcal{F} = \{ x \in \mathcal{Q} \mid x_i \in \mathbb{Z} \ \forall i \in \mathcal{S} \}$, no longer forms a polyhedron.

The **convex hull** of a set $X$ is the minimal convex set that contains $X$. In solving a mixed integer linear program, in order to take advantage of LP-based algorithms you want to find the convex hull, $\text{conv}(\mathcal{F})$, of $\mathcal{F}$. If you can find $\text{conv}(\mathcal{F})$ and describe it compactly, then you can solve a mixed integer linear program with a linear programming solver. This is generally very difficult, so you must be satisfied with finding an approximation. Typically, the better the approximation, the more efficiently the LP-based branch-and-bound algorithm can perform.

As described in the section “Branch-and-Bound Algorithm” on page 179, the branch-and-bound algorithm begins by solving the linear programming relaxation over the polyhedron $\mathcal{Q}$. Clearly, $\mathcal{Q}$ contains the convex hull of the feasible region of the original integer program; that is, $\text{conv}(\mathcal{F}) \subseteq \mathcal{Q}$.

**Cutting plane** techniques are used to tighten the linear relaxation to better approximate $\text{conv}(\mathcal{F})$. Assume you are given a solution $\bar{x}$ to some intermediate linear relaxation during the branch-and-bound algorithm. A cut, or valid inequality ($\pi x \leq \pi^0$), is some half-space with the following characteristics:

- The half-space contains $\text{conv}(\mathcal{F})$; that is, every integer feasible solution is feasible for the cut ($\pi x \leq \pi^0$, $\forall x \in \mathcal{F}$).
- The half-space does not contain the current solution $\bar{x}$; that is, $\bar{x}$ is not feasible for the cut ($\pi \bar{x} > \pi^0$).

Cutting planes were first made popular by Dantzig, Fulkerson, and Johnson (1954) in their work on the traveling salesman problem. The two major classifications of cutting planes are **generic cuts** and **structured cuts**. Generic cuts are based solely on algebraic arguments and can be applied to any relaxation of any integer program. Structured cuts are specific to certain structures that can be found in some relaxations of the mixed integer linear program. These structures are automatically discovered during the cut initialization phase of PROC OPTMILP. Table 7.3 lists the various types of cutting planes that are built into PROC OPTMILP. Included in each type are algorithms for numerous variations based on different relaxations and lifting techniques. For a survey of cutting plane techniques for mixed integer programming, see Marchand et al. (1999). For a survey of lifting techniques, see Atamturk (2004).

<table>
<thead>
<tr>
<th>Generic Cutting Planes</th>
<th>Structured Cutting Planes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gomory mixed integer</td>
<td>Cliques</td>
</tr>
<tr>
<td>Lift-and-project</td>
<td>Flow cover</td>
</tr>
<tr>
<td>Mixed integer rounding</td>
<td>Flow path</td>
</tr>
<tr>
<td>Mixed lifted 0-1</td>
<td>Generalized upper bound cover</td>
</tr>
<tr>
<td>Zero-half</td>
<td>Implied bound</td>
</tr>
<tr>
<td></td>
<td>Knapsack cover</td>
</tr>
<tr>
<td></td>
<td>Multicommodity network flow</td>
</tr>
</tbody>
</table>

You can set levels for individual cuts by using the **CUTCLIQUE=**, **CUTFLOWCOVER=**, **CUTFLOWPATH=**, **CUTGOMORY=**, **CUTGUB=**, **CUTIMPLIED=**, **CUTKNAPSACK=**, **CUTLAP=**, **CUTMIR=**, **CUTMULTICOMMODITY=**, and **CUTZEROHALF=** options. The valid levels for these options are given in Table 7.2.

The cut level determines the internal strategy used by PROC OPTMILP for generating the cutting planes.
The strategy consists of several factors, including how frequently the cut search is called, the number of cuts allowed, and the aggressiveness of the search algorithms.

Sophisticated cutting planes, such as those included in PROC OPTMILP, can take a great deal of CPU time. Usually, the additional tightening of the relaxation helps speed up the overall process because it provides better bounds for the branch-and-bound tree and helps guide the LP solver toward integer solutions. In rare cases, shutting off cutting planes completely might lead to faster overall run times.

The default settings of PROC OPTMILP have been tuned to work well for most instances. However, problem-specific expertise might suggest adjusting one or more of the strategies. These options give you that flexibility.

### Primal Heuristics

Primal heuristics, an important component of PROC OPTMILP, are applied during the branch-and-bound algorithm. They are used to find integer feasible solutions early in the search tree, thereby improving the upper bound for a minimization problem. Primal heuristics play a role that is complementary to cutting planes in reducing the gap between the upper and lower bounds, thus reducing the size of the branch-and-bound tree.

Applying primal heuristics in the branch-and-bound algorithm assists in the following areas:

- finding a good upper bound early in the tree search (this can lead to earlier fathoming, resulting in fewer subproblems to be processed)
- locating a reasonably good feasible solution when that is sufficient (sometimes a good feasible solution is the best the solver can produce within certain time or resource limits)
- providing upper bounds for some bound-tightening techniques

The OPTMILP procedure implements several heuristic methodologies. Some algorithms, such as rounding and iterative rounding (diving) heuristics, attempt to construct an integer feasible solution by using fractional solutions to the continuous relaxation at each node of the branch-and-cut tree. Other algorithms start with an incumbent solution and attempt to find a better solution within a neighborhood of the current best solution.

The HEURISTICS= option enables you to control the level of primal heuristics that are applied by PROC OPTMILP. This level determines how frequently primal heuristics are applied during the tree search. Some expensive heuristics might be disabled by the solver at less aggressive levels. Setting the HEURISTICS= option to a lower level also reduces the maximum number of iterations that are allowed in iterative heuristics.

### Parallel Processing

You can run the branch-and-cut algorithm in single-machine mode, distributed mode, or concurrent mode. In single-machine mode, the computation is executed on a single computer by multiple threads. The number of threads is specified by the NTHREADS= option. In distributed mode, the branch-and-cut algorithm is executed on grid nodes. In concurrent mode, the computation is executed on grid nodes by solving the same problem across grid nodes under different PROC OPTMILP settings. If distributed mode is enabled, then by default PROC OPTMILP is executed deterministically. You disable deterministic distributed mode by setting
the DETERMINISTIC= option to FALSE. If concurrent mode is enabled, then by default PROC OPTMILP is executed nondeterministically.

You can run the decomposition algorithm in either single-machine or distributed mode. In distributed mode, the computation is executed on multiple computing nodes in a distributed computing environment.

---

**Node Log**

The following information about the status of the branch-and-bound algorithm is printed in the node log:

<table>
<thead>
<tr>
<th>Node</th>
<th>indicates the sequence number of the current node in the search tree.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>indicates the current number of active nodes in the branch-and-bound tree.</td>
</tr>
<tr>
<td>Sols</td>
<td>indicates the number of feasible solutions found so far.</td>
</tr>
<tr>
<td>BestInteger</td>
<td>indicates the best upper bound (assuming minimization) found so far.</td>
</tr>
<tr>
<td>BestBound</td>
<td>indicates the best lower bound (assuming minimization) found so far.</td>
</tr>
<tr>
<td>Gap</td>
<td>indicates the relative gap between BestInteger and BestBound, displayed as a percentage. If the relative gap is larger than 1,000, then the absolute gap is displayed. If no active nodes remain, the value of Gap is 0.</td>
</tr>
<tr>
<td>Time</td>
<td>indicates the elapsed real or CPU time.</td>
</tr>
</tbody>
</table>

If the CONCURRENT= option is set to TRUE, then the following information about the status of the branch-and-bound algorithm is printed in the node log:

<table>
<thead>
<tr>
<th>Node</th>
<th>indicates the total number of nodes processed by all concurrent solves.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Active</td>
<td>indicates the minimum number of active nodes among all concurrent solves.</td>
</tr>
<tr>
<td>Sols</td>
<td>indicates the number of feasible solutions found by all concurrent solves.</td>
</tr>
<tr>
<td>BestInteger</td>
<td>indicates the best upper bound (assuming minimization) found by all concurrent solves.</td>
</tr>
<tr>
<td>BestBound</td>
<td>indicates the best lower bound (assuming minimization) found by all concurrent solves.</td>
</tr>
<tr>
<td>Gap</td>
<td>indicates the relative gap between BestInteger and BestBound, displayed as a percentage. If the relative gap is larger than 1,000, then the absolute gap is displayed. If no active nodes remain, the value of Gap is 0.</td>
</tr>
<tr>
<td>Time</td>
<td>indicates the elapsed real or CPU time.</td>
</tr>
</tbody>
</table>

If CONCURRENT=TRUE, the node log displays the progress of the overall grid. If DETERMINISTIC=TRUE, PROC OPTMILP tries to return a reproducible solution. Therefore, there might be discrepancies between the node log and the final solution that is reported by the deterministic concurrent OPTMILP procedure.

The LOGFREQ= and LOGLEVEL= options can be used to control the amount of information printed in the node log. By default, the root node processing information is printed and, if possible, an entry is made every five seconds. A new entry is also included each time a better integer solution is found. The LOGFREQ= option enables you to change the interval between entries in the node log. Figure 7.4 shows a sample node log.
Figure 7.4 Sample Node Log

NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0 fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 45 variables and 59 constraints.
NOTE: The MILP presolver removed 144 constraint coefficients.
NOTE: The presolved problem has 88 variables, 64 constraints, and 203 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Solns</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>869515</td>
<td>.</td>
<td>0</td>
</tr>
<tr>
<td>343</td>
<td>306</td>
<td>1</td>
<td>881651</td>
<td>874001</td>
<td>0.88%</td>
<td>0</td>
</tr>
<tr>
<td>1033</td>
<td>339</td>
<td>3</td>
<td>881438</td>
<td>874303</td>
<td>0.82%</td>
<td>0</td>
</tr>
<tr>
<td>1215</td>
<td>471</td>
<td>4</td>
<td>881438</td>
<td>874458</td>
<td>0.80%</td>
<td>0</td>
</tr>
<tr>
<td>1497</td>
<td>128</td>
<td>5</td>
<td>878651</td>
<td>874753</td>
<td>0.45%</td>
<td>0</td>
</tr>
<tr>
<td>2143</td>
<td>365</td>
<td>6</td>
<td>878430</td>
<td>874753</td>
<td>0.42%</td>
<td>0</td>
</tr>
<tr>
<td>2164</td>
<td>365</td>
<td>7</td>
<td>878430</td>
<td>874753</td>
<td>0.42%</td>
<td>0</td>
</tr>
<tr>
<td>2224</td>
<td>378</td>
<td>8</td>
<td>878430</td>
<td>874753</td>
<td>0.42%</td>
<td>0</td>
</tr>
<tr>
<td>5534</td>
<td>0</td>
<td>8</td>
<td>878430</td>
<td>878430</td>
<td>0.00%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Optimal.
NOTE: Objective = 878430.316.
NOTE: The Cloud Analytic Services server processed the request in 1.843911 seconds.

ODS Tables

PROC OPTMILP creates two Output Delivery System (ODS) tables by default. The first table, ProblemSummary, is a summary of the input MILP problem. The second table, SolutionSummary, is a brief summary of the solution status. You can use ODS table names to select tables and create output data tables. For more information about ODS, see SAS Output Delivery System: User’s Guide.

If you specify a value of 2 for the PRINTLEVEL= option, then the ProblemStatistics table and a Timing table are produced. The ProblemStatistics table contains information about the problem data. See the section “Problem Statistics” on page 190 for more information. The Timing table contains detailed information about the solution time.

Table 7.4 lists all the ODS tables that can be produced by the OPTMILP procedure, along with the statement and option specifications required to produce each table.
Table 7.4  ODS Tables Produced by PROC OPTMILP

<table>
<thead>
<tr>
<th>ODS Table Name</th>
<th>Description</th>
<th>Statement</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProblemSummary</td>
<td>Summary of the input MILP problem</td>
<td>PROC OPTMILP</td>
<td>PRINTLEVEL=1 (default)</td>
</tr>
<tr>
<td>SolutionSummary</td>
<td>Summary of the solution status</td>
<td>PROC OPTMILP</td>
<td>PRINTLEVEL=1 (default)</td>
</tr>
<tr>
<td>ProblemStatistics</td>
<td>Description of input problem data</td>
<td>PROC OPTMILP</td>
<td>PRINTLEVEL=2</td>
</tr>
<tr>
<td>Timing</td>
<td>Detailed solution timing</td>
<td>PROC OPTMILP</td>
<td>PRINTLEVEL=2</td>
</tr>
</tbody>
</table>

A typical ProblemSummary table is shown in Figure 7.5.

Figure 7.5  Example PROC OPTMILP Output: Problem Summary

The OPTMILP Procedure

| Problem Summary |
|-----------------|----------------|
| Problem Name    | EX_MIP         |
| Objective Sense | Minimization   |
| Objective Function | COST       |
| RHS             | RHS            |

Number of Variables   3
Bounded Above       0
Bounded Below       0
Bounded Above and Below 3
Free               0
Fixed              0
Binary            3
Integer            0

Number of Constraints 3
LE (<=)          2
EQ (=)           0
GE (>=)          1
Range            0

Constraint Coefficients 8
A typical SolutionSummary table is shown in Figure 7.6.

**Figure 7.6** Example PROC OPTMILP Output: Solution Summary

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Relative Gap</td>
</tr>
<tr>
<td>Absolute Gap</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
</tr>
<tr>
<td>Integer Infeasibility</td>
</tr>
<tr>
<td>Best Bound</td>
</tr>
<tr>
<td>Nodes</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

You can create output data tables from these tables by using the ODS OUTPUT statement. The output data tables from the preceding example are displayed in Figure 7.7 and Figure 7.8, where you can also find variable names for the tables used in the ODS template of the OPTMILP procedure.
**Figure 7.7** ODS Output Data Table: Problem Summary

**Problem Summary**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name1</th>
<th>Label1</th>
<th>cValue1</th>
<th>nValue1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>problemName</td>
<td>Problem Name</td>
<td>EX_MIP</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>objectiveSense</td>
<td>Objective Sense</td>
<td>Minimization</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>objectiveName</td>
<td>Objective Function</td>
<td>COST</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>rhs</td>
<td>RHS</td>
<td>RHS</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>numVars</td>
<td>Number of Variables</td>
<td>3</td>
<td>3.000000</td>
</tr>
<tr>
<td>7</td>
<td>numBoundedAbove</td>
<td>Bounded Above</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>numBoundedBelow</td>
<td>Bounded Below</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>numBounded</td>
<td>Bounded Above and Below</td>
<td>3</td>
<td>3.000000</td>
</tr>
<tr>
<td>10</td>
<td>numFree</td>
<td>Free</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>numFixed</td>
<td>Fixed</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>numBinary</td>
<td>Binary</td>
<td>3</td>
<td>3.000000</td>
</tr>
<tr>
<td>13</td>
<td>numInteger</td>
<td>Integer</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>numRows</td>
<td>Number of Constraints</td>
<td>3</td>
<td>3.000000</td>
</tr>
<tr>
<td>16</td>
<td>numLE</td>
<td>LE (&lt;=)</td>
<td>2</td>
<td>2.000000</td>
</tr>
<tr>
<td>17</td>
<td>numEQ</td>
<td>EQ (=)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td>numGE</td>
<td>GE (&gt;=)</td>
<td>1</td>
<td>1.000000</td>
</tr>
<tr>
<td>19</td>
<td>numRange</td>
<td>Range</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>numElements</td>
<td>Constraint Coefficients</td>
<td>8</td>
<td>8.000000</td>
</tr>
</tbody>
</table>
Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of $10^9$) can cause difficulty when the remaining entries are single-digit numbers. The `PRINTLEVEL=2` option in the `OPTMILP` procedure causes the ODS table `ProblemStatistics` to be generated. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Figure 7.9 demonstrates the contents of the ODS table `ProblemStatistics`.
The variable names in the ODS table ProblemStatistics are Label1, cValue1, and nValue1.

Macro Variable _OROPTMILP_

The OPTMILP procedure defines a macro variable named _OROPTMILP_. This variable contains a character string that indicates the status of the OPTMILP procedure upon termination. The various terms of the variable are interpreted as follows.

**STATUS**

indicates the solver status at termination. It can take one of the following values:

- **OK**: The procedure terminated normally.
- **SYNTAX_ERROR**: Incorrect syntax was used.
- **DATA_ERROR**: The input data was inconsistent.
- **OUT_OF_MEMORY**: Insufficient memory was allocated to the procedure.
- **IO_ERROR**: A problem occurred in reading or writing data.
- **ERROR**: The status cannot be classified into any of the preceding categories.
ALGORITHM
indicates the algorithm that produced the solution data in the macro variable. This term only appears when STATUS=OK. It can take one of the following values:

BAC The branch-and-cut algorithm produced the solution data.
DECOMP The decomposition algorithm produced the solution data.

SOLUTION_STATUS
indicates the solution status at termination. It can take one of the following values:

OPTIMAL The solution is optimal.
OPTIMAL_AGAP The solution is optimal within the absolute gap specified by the ABSOBJGAP= option.
OPTIMAL_RGAP The solution is optimal within the relative gap specified by the RELOBJGAP= option.
OPTIMAL_COND The solution is optimal, but some infeasibilities (primal, bound, or integer) exceed tolerances due to scaling or choice of a small INTTOL= value.
TARGET The solution is not worse than the target specified by the TARGET= option.
INFEASIBLE The problem is infeasible.
UNBOUNDED The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED The problem is infeasible or unbounded.
SOLUTION_LIM The solver reached the maximum number of solutions specified by the MAXSOLS= option.
NODE_LIM_SOL The solver reached the maximum number of nodes specified by the MAXNODES= option and found a solution.
NODE_LIM_NOSOL The solver reached the maximum number of nodes specified by the MAXNODES= option and did not find a solution.
TIME_LIM_SOL The solver reached the execution time limit specified by the MAXTIME= option and found a solution.
TIME_LIM_NOSOL The solver reached the execution time limit specified by the MAXTIME= option and did not find a solution.
ABORT_SOL The solver was stopped by the user but still found a solution.
ABORT_NOSOL The solver was stopped by the user and did not find a solution.
OUTMEM_SOL The solver ran out of memory but still found a solution.
OUTMEM_NOSOL The solver ran out of memory and either did not find a solution or failed to output the solution due to insufficient memory.
FAIL_SOL The solver stopped due to errors but still found a solution.
FAIL_NOSOL The solver stopped due to errors and did not find a solution.
**OBJECTIVE**
indicates the objective value obtained by the solver at termination.

**RELATIVE_GAP**
indicates the relative gap between the best integer objective (BestInteger) and the best bound on the objective function value (BestBound) upon termination of the OPTMILP procedure. The relative gap is equal to

\[
\frac{|\text{BestInteger} - \text{BestBound}|}{10 + |\text{BestBound}|}
\]

**ABSOLUTE_GAP**
indicates the absolute gap between the best integer objective (BestInteger) and the best bound on the objective function value (BestBound) upon termination of the OPTMILP procedure. The absolute gap is equal to |BestInteger − BestBound|.

**PRIMAL_INFEASIBILITY**
indicates the maximum (absolute) violation of the primal constraints by the solution.

**BOUND_INFEASIBILITY**
indicates the maximum (absolute) violation by the solution of the lower or upper bounds (or both).

**INTEGER_INFEASIBILITY**
indicates the maximum (absolute) violation of the integrality of integer variables returned by the OPTMILP procedure.

**BEST_BOUND**
indicates the best bound on the objective function value at termination. A missing value indicates that the OPTMILP procedure was not able to obtain such a bound.

**NODES**
indicates the number of nodes enumerated by the OPTMILP procedure when using the branch-and-bound algorithm.

**ITERATIONS**
indicates the number of simplex iterations taken to solve the problem.

**PRESOLVE_TIME**
indicates the time (in seconds) used in preprocessing.

**SOLUTION_TIME**
indicates the time (in seconds) taken to solve the problem, including preprocessing time.

**NOTE:** The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.
Examples: OPTMILP Procedure

This section contains examples that illustrate the options and syntax of PROC OPTMILP. Example 7.1 demonstrates a model contained in an MPS-format SAS data set and finds an optimal solution by using PROC OPTMILP. Example 7.2 illustrates the use of standard MPS files in PROC OPTMILP. Example 7.3 demonstrates how to warm start PROC OPTMILP. More detailed examples of mixed integer linear programs, along with example SAS code, are given in Chapter 14.

Example 7.1: Simple Integer Linear Program

This example illustrates a model in an MPS-format data table. This data table is passed to PROC OPTMILP, and a solution is found.

Consider a scenario where you have a container with a set of limiting attributes (volume $V$ and weight $W$) and a set $I$ of items that you want to pack. Each item type $i$ has a certain value $p_i$, a volume $v_i$, and a weight $w_i$. You must choose at most four items of each type so that the total value is maximized and all the chosen items fit into the container. Let $x_i$ be the number of items of type $i$ to be included in the container. This model can be formulated as the following integer linear program:

$$\begin{align*}
\text{max} & \quad \sum_{i \in I} p_i x_i \\
\text{s.t.} & \quad \sum_{i \in I} v_i x_i \leq V & \text{(volume_con)} \\
& \quad \sum_{i \in I} w_i x_i \leq W & \text{(weight_con)} \\
& \quad x_i \leq 4 & \forall i \in I \\
& \quad x_i \in \mathbb{Z}^+ & \forall i \in I
\end{align*}$$

Constraint (volume_con) enforces the volume capacity limit, while constraint (weight_con) enforces the weight capacity limit. An instance of this problem can be saved in an MPS-format data table by using the following code:
data mycas.ex1data;
  input _id_ field1 $ field2 $ field3 $ field4 field5 $ field6;
datalines;
1   NAME .     .  .       .   .  ex1data
2   ROWS .     .  .       .   .  .
3   MAX z  .    .  .       .   .  .
4   L    volume_con . .       .   .  .
5   L    weight_con . .       .   .  .
6   COLUMNS .    .  .       .   .  .
7 .   .  .     .  .       .   .  .
8 .   .  x[1] z  1       volume_con 10
10 .  x[2] z  2       volume_con 300
12 .  x[3] z  3       volume_con 250
14 .  x[4] z  4       volume_con 610
15 .  x[4] weight_con 100 .  .
16 .  x[5] z  5       volume_con 500
18 .  x[6] z  6       volume_con 120
20 .  x[7] z  7       volume_con 45
22 .  x[8] z  8       volume_con 100
24 .  x[9] z  9       volume_con 200
26 .  x[10] z 10      volume_con 61
27 . x[10] weight_con 110 .  .
28 .  .  .     .  .       .   .  .
29 .  .  .     .  .       .   .  .
30 .  .  .     .  .       .   .  .
31 .  .  .     .  .       .   .  .
32 .  BOUNDS .    .  .       .   .  .
33 .  .  .     .  .       .   .  .
34 .  .  .     .  .       .   .  .
35 .  .  .     .  .       .   .  .
36 .  .  .     .  .       .   .  .
37 .  .  .     .  .       .   .  .
38 .  .  .     .  .       .   .  .
39 .  .  .     .  .       .   .  .
40 .  .  .     .  .       .   .  .
41 .  .  .     .  .       .   .  .
42 .  .  .     .  .       .   .  .
43 .  .  .     .  .       .   .  .

In the COLUMNS section of this data table, the name of the objective is \( z \), and the objective coefficients \( p_i \) appear in field4. The coefficients \( v_i \) of \( \text{(volume\_con)} \) appear in field6. The coefficients \( w_i \) of \( \text{(weight\_con)} \) appear in field4. In the RHS section, the bounds \( V \) and \( W \) appear in field4. The _id_ column ensures that the data can be read in the right order even if the table is stored on separate machines in CAS.
This problem can be solved by using the following statements to call the OPTMILP procedure:

```plaintext
proc optmilp data=mycas.ex1data primalout=mycas.ex1soln;
run;
```

The progress of the solver is shown in Output 7.1.1.

**Output 7.1.1** Simple Integer Linear Program PROC OPTMILP Log

NOTE: The problem exdata has 10 variables (0 binary, 10 integer, 0 free, 0 fixed).
NOTE: The problem has 2 constraints (2 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 20 constraint coefficients.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 2 variables and 0 constraints.
NOTE: The MILP presolver removed 4 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 8 variables, 2 constraints, and 16 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>85.00000000</td>
<td>158.0000000</td>
<td>46.20%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>85.00000000</td>
<td>88.0955497</td>
<td>3.51%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>85.00000000</td>
<td>87.4545455</td>
<td>2.81%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>85.00000000</td>
<td>87.4545455</td>
<td>2.81%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The MILP presolver is applied again.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>87.00000000</td>
<td>87.4545455</td>
<td>0.52%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
<td>87.00000000</td>
<td>87.0000000</td>
<td>0.00%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Optimal.
NOTE: Objective = 87.
NOTE: The Cloud Analytic Services server processed the request in 1.287507 seconds.
NOTE: The data set MYCAS.EX1SOLN has 10 observations and 8 variables.
The data table ex1soln is shown in Output 7.1.2.

**Output 7.1.2** Simple Integer Linear Program Solution

**Example 1 Solution Data**

<table>
<thead>
<tr>
<th>Objective Function ID</th>
<th>RHS ID</th>
<th>Variable Name</th>
<th>Variable Type</th>
<th>Objective Coefficient</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
<th>Variable Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td>.RHS.</td>
<td>x[1]</td>
<td>l</td>
<td>1</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>.RHS.</td>
<td>x[2]</td>
<td>l</td>
<td>2</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>.RHS.</td>
<td>x[3]</td>
<td>l</td>
<td>3</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>.RHS.</td>
<td>x[4]</td>
<td>l</td>
<td>4</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>.RHS.</td>
<td>x[5]</td>
<td>l</td>
<td>5</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>.RHS.</td>
<td>x[6]</td>
<td>l</td>
<td>6</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>.RHS.</td>
<td>x[7]</td>
<td>l</td>
<td>7</td>
<td>0</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>z</td>
<td>.RHS.</td>
<td>x[8]</td>
<td>l</td>
<td>8</td>
<td>0</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>z</td>
<td>.RHS.</td>
<td>x[9]</td>
<td>l</td>
<td>9</td>
<td>0</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>z</td>
<td>.RHS.</td>
<td>x[10]</td>
<td>l</td>
<td>10</td>
<td>0</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

The optimal solution is $x_6 = 3, x_7 = 1, x_8 = 4$, and $x_{10} = 3$, with a total value of 87. From this solution, you can compute the total volume used, which is 988 (≤ $V = 1000$); the total weight used is 499 (≤ $W = 500$). The problem summary and solution summary are shown in Output 7.1.3.

**Output 7.1.3** Simple Integer Linear Program Summary

**The OPTMILP Procedure**

**Problem Summary**

<table>
<thead>
<tr>
<th>Problem Name</th>
<th>ex1data</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
<td>Maximization</td>
</tr>
<tr>
<td>Objective Function</td>
<td>z</td>
</tr>
<tr>
<td>RHS</td>
<td>.RHS.</td>
</tr>
</tbody>
</table>

Number of Variables 10
Bounded Above 0
Bounded Below 0
Bounded Above and Below 10
Free 0
Fixed 0
Binary 0
Integer 10

Number of Constraints 2
LE (<=) 2
EQ (=) 0
GE (>=) 0
Range 0

Constraint Coefficients 20
Example 7.2: MIPLIB Benchmark Instance

The following example illustrates the conversion of a standard MPS-format file into an MPS-format data table. The problem is re-solved several times, each time by using a different control option. For such a small example, it is helpful to disable cuts and heuristics in order to see the computational savings gained by using other options. For larger or more complex examples, the benefits of using the various control options are more pronounced.

The standard set of MILP benchmark cases are the various versions of MIPLIB (Bixby et al. 1998, Achterberg, Koch, and Martin 2003) and can be found at http://miplib.zib.de/. The following statement uses the %MPS2SASD macro to convert an example from MIPLIB to a data table:

```%mps2sasd(mpsfile="bell3a.mps", outdata=mycas.mpsdata);```

The problem can then be solved using PROC OPTMILP on the data table uploaded to CAS:

```proc optmilp data=mycas.mpsdata allcuts=none heuristics=none; run;```

The resulting log is shown in Output 7.2.1.
Output 7.2.1 MIPLIB PROC OPTMILP Log

NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0 fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 45 variables and 59 constraints.
NOTE: The MILP presolver removed 144 constraint coefficients.
NOTE: The MILP presolver modified 25 constraint coefficients.
NOTE: The presolved problem has 88 variables, 64 constraints, and 203 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Soln</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>869515</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>343</td>
<td>306</td>
<td>1</td>
<td>881651</td>
<td>874001</td>
<td>0.88%</td>
<td>0</td>
</tr>
<tr>
<td>1033</td>
<td>339</td>
<td>3</td>
<td>881438</td>
<td>874303</td>
<td>0.82%</td>
<td>0</td>
</tr>
<tr>
<td>1215</td>
<td>471</td>
<td>4</td>
<td>881438</td>
<td>874458</td>
<td>0.80%</td>
<td>0</td>
</tr>
<tr>
<td>1497</td>
<td>128</td>
<td>5</td>
<td>878651</td>
<td>874753</td>
<td>0.45%</td>
<td>0</td>
</tr>
<tr>
<td>2143</td>
<td>365</td>
<td>6</td>
<td>878430</td>
<td>874753</td>
<td>0.42%</td>
<td>0</td>
</tr>
<tr>
<td>2164</td>
<td>365</td>
<td>7</td>
<td>878430</td>
<td>874753</td>
<td>0.42%</td>
<td>0</td>
</tr>
<tr>
<td>2224</td>
<td>378</td>
<td>8</td>
<td>878430</td>
<td>874753</td>
<td>0.42%</td>
<td>0</td>
</tr>
<tr>
<td>5534</td>
<td>0</td>
<td>8</td>
<td>878430</td>
<td>878430</td>
<td>0.00%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Optimal.
NOTE: Objective = 878430.316.
NOTE: The Cloud Analytic Services server processed the request in 1.843911 seconds.
If there is an objective value that, even if it is not optimal, satisfies your requirements, then you can save time by using the `TARGET=` option. The following PROC OPTMILP call solves the problem with a target value of 880,000:

```plaintext
proc optmilp data=mycas.mpsdata allcuts=none heuristics=none
target=880000;
run;
```

The relevant results from this run are displayed in Output 7.2.2. In this case, there is a decrease in the number of nodes processed, but the objective value has increased.

**Output 7.2.2** MIPLIB PROC OPTMILP Log with TARGET= Option

---

NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0 fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 45 variables and 59 constraints.
NOTE: The MILP presolver removed 144 constraint coefficients.
NOTE: The MILP presolver modified 25 constraint coefficients.
NOTE: The presolved problem has 88 variables, 64 constraints, and 203 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Solns</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>.</td>
<td>869515</td>
<td>.</td>
<td>0</td>
</tr>
<tr>
<td>343</td>
<td>306</td>
<td>1</td>
<td>881651</td>
<td>874001</td>
<td>0.88%</td>
<td>0</td>
</tr>
<tr>
<td>1033</td>
<td>339</td>
<td>3</td>
<td>881438</td>
<td>874303</td>
<td>0.82%</td>
<td>0</td>
</tr>
<tr>
<td>1215</td>
<td>471</td>
<td>4</td>
<td>881438</td>
<td>874458</td>
<td>0.80%</td>
<td>0</td>
</tr>
<tr>
<td>1497</td>
<td>128</td>
<td>5</td>
<td>878651</td>
<td>874753</td>
<td>0.45%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Target reached.
NOTE: Objective of the best integer solution found = 878651.068.
NOTE: The Cloud Analytic Services server processed the request in 1.720273 seconds.
When the objective value of a solution is within a certain relative gap of the optimal objective value, the procedure stops. The acceptable relative gap can be changed using the `RELOBJGAP=` option, as demonstrated in the following example:

```
proc optmilp data=mycas.mpsdata allcuts=none heuristics=none
   relobjgap=0.3;
run;
```

The relevant results from this run are displayed in Output 7.2.3. In this case, since the specified `RELOBJGAP=` value is larger than the default value, the number of nodes processed has decreased from their values in the original run. Note that these savings are possible because PROC OPTMILP makes a weaker statement about how good the solution found is, which is indicated by the decreased best bound value.

Output 7.2.3  MIPLIB PROC OPTMILP Log with RELOBJGAP= Option

```
NOTE: The problem BELL3A has 133 variables (39 binary, 32 integer, 0 free, 0 fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 45 variables and 59 constraints.
NOTE: The MILP presolver removed 144 constraint coefficients.
NOTE: The MILP presolver modified 25 constraint coefficients.
NOTE: The presolved problem has 88 variables, 64 constraints, and 203 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>.</td>
<td>869515</td>
<td>.</td>
<td>0</td>
</tr>
<tr>
<td>343</td>
<td>306</td>
<td>1</td>
<td>881651</td>
<td>874001</td>
<td>0.88%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Optimal within relative gap.
NOTE: Objective = 881650.93.
NOTE: The Cloud Analytic Services server processed the request in 3.228142 seconds.
```
The **MAXTIME=** option enables you to accept the best solution produced by PROC OPTMILP in a specified amount of time. The following example illustrates the use of the **MAXTIME=** option:

```latex
proc optmilp data=mycas.mpsdata allcuts=none heuristics=none
   maxtime=0.5;
run;
```

The relevant results from this run are displayed in **Output 7.2.4**. In this case the solution process stopped early because the time limit was reached.

**Output 7.2.4** MIPLIB PROC OPTMILP Log with MAXTIME= Option

---

The **MAXNODES=** option enables you to limit the number of nodes generated by PROC OPTMILP. The following example illustrates the use of the **MAXNODES=** option:

```latex
proc optmilp data=mycas.mpsdata allcuts=none heuristics=none
   maxnodes=1000;
run;
```

The relevant results from this run are displayed in **Output 7.2.5**. PROC OPTMILP displays the best objective value of all the solutions produced within the first 1000 nodes.
Example 7.3: Facility Location

This advanced example demonstrates how to warm start PROC OPTMILP by using the PRIMALIN= option. The model is constructed in PROC OPTMODEL and saved in an MPS-format data table for use in PROC OPTMILP. This problem can also be solved from within PROC OPTMODEL; see Chapter 14 for details.

Consider the classical facility location problem. Given a set $L$ of customer locations and a set $F$ of candidate facility sites, you must decide on which sites to build facilities and assign coverage of customer demand to these sites so as to minimize cost. All customer demand $d_i$ must be satisfied, and each facility has a demand capacity limit $C$. The total cost is the sum of the distances $c_{ij}$ between facility $j$ and its assigned customer $i$, plus a fixed charge $f_j$ for building a facility at site $j$. Let $y_j = 1$ represent choosing site $j$ to build a facility, and 0 otherwise. Also, let $x_{ij} = 1$ represent the assignment of customer $i$ to facility $j$, and 0 otherwise. This model can be formulated as the following integer linear program:

$$\begin{align*}
\text{min} & \quad \sum_{i \in L} \sum_{j \in F} c_{ij} x_{ij} + \sum_{j \in F} f_j y_j \\
\text{s.t.} & \quad \sum_{j \in F} x_{ij} = 1 \quad \forall i \in L \quad \text{(assign_def)} \\
& \quad x_{ij} \quad \leq \quad y_j \quad \forall i \in L, j \in F \quad \text{(link)} \\
& \quad \sum_{i \in L} d_i x_{ij} \quad \leq \quad C y_j \quad \forall j \in F \quad \text{(capacity)} \\
& \quad x_{ij} \in \{0, 1\} \quad \forall i \in L, j \in F \\
& \quad y_j \in \{0, 1\} \quad \forall j \in F
\end{align*}$$

Example 7.2.5 MIPLIB PROC OPTMILP Log with MAXNODES= Option

NOTE: The problem BELLA3A has 133 variables (39 binary, 32 integer, 0 free, 0 fixed).
NOTE: The problem has 123 constraints (123 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 347 constraint coefficients.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 45 variables and 59 constraints.
NOTE: The MILP presolver removed 144 constraint coefficients.
NOTE: The MILP presolver modified 25 constraint coefficients.
NOTE: The presolved problem has 88 variables, 64 constraints, and 203 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td>869515</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>343</td>
<td>306</td>
<td>1</td>
<td>881651</td>
<td>874001</td>
<td>0.88</td>
<td>0</td>
</tr>
<tr>
<td>999</td>
<td>315</td>
<td>2</td>
<td>881651</td>
<td>874303</td>
<td>0.84</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Node limit reached.
NOTE: Objective of the best integer solution found = 881650.93.
NOTE: The Cloud Analytic Services server processed the request in 0.562693 seconds.
Constraint (assign_def) ensures that each customer is assigned to exactly one site. Constraint (link) forces a facility to be built if any customer has been assigned to that facility. Finally, constraint (capacity) enforces the capacity limit at each site.

Consider also a variation of this same problem where there is no cost for building a facility. This problem is typically easier to solve than the original problem. For this variant, let the objective be

$$\min \sum_{i \in L} \sum_{j \in F} c_{ij} x_{ij}$$

First, construct a random instance of this problem by using the following DATA steps:

```bash
%let NumCustomers = 50;
%let NumSites = 10;
%let SiteCapacity = 35;
%let MaxDemand = 10;
%let xmax = 200;
%let ymax = 100;
%let seed = 423;

/* generate random customer locations */
data mycas.cdata(drop=i);
call streaminit(&seed);
length name $8;
do i = 1 to &NumCustomers;
   name = compress('C'||put(i,best.));
   x = rand('UNIFORM') * &xmax;
   y = rand('UNIFORM') * &ymax;
   demand = rand('UNIFORM') * &MaxDemand;
   output;
end;
run;

/* generate random site locations and fixed charge */
data mycas.sdata(drop=i);
call streaminit(&seed);
length name $8;
do i = 1 to &NumSites;
   name = compress('SITE'||put(i,best.));
   x = rand('UNIFORM') * &xmax;
   y = rand('UNIFORM') * &ymax;
   fixed_charge = 30 * (abs(&xmax/2-x) + abs(&ymax/2-y));
   output;
end;
run;
```

The following PROC OPTMODEL statements generate the model and define both variants of the cost function:
Example 7.3: Facility Location

proc optmodel;
  set <str> CUSTOMERS;
  set <str> SITES init {};
  /* x and y coordinates of CUSTOMERS and SITES */
  num x {CUSTOMERS union SITES};
  num y {CUSTOMERS union SITES};
  num demand {CUSTOMERS};
  num fixed_charge {SITES};
  /* distance from customer i to site j */
  num dist {i in CUSTOMERS, j in SITES} = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2);
  read data mycas.cdata into CUSTOMERS=[name] x y demand;
  read data mycas.sdata into SITES=[name] x y fixed_charge;
  var Assign {CUSTOMERS, SITES} binary;
  var Build {SITES} binary;
  /* each customer assigned to exactly one site */
  con assign_def {i in CUSTOMERS}:
    sum {j in SITES} Assign[i,j] = 1;
  /* if customer i assigned to site j, then facility must be */
  /* built at j */
  con link {i in CUSTOMERS, j in SITES}:
    Assign[i,j] <= Build[j];
  /* each site can handle at most &SiteCapacity demand */
  con capacity {j in SITES}:
    sum {i in CUSTOMERS} demand[i] * Assign[i,j] <= &SiteCapacity * Build[j];
  min CostNoFixedCharge
    = sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j];
  save mps mycas.nofcdata;
  min CostFixedCharge
    = CostNoFixedCharge
      + sum {j in SITES} fixed_charge[j] * Build[j];
  save mps mycas.fcdata;
quit;

First solve the problem for the model with no fixed charge by using the following statements. The first PROC SQL call populates the macro variables varcostNo. This macro variable displays the objective value when the results are plotted. The second PROC SQL call generates a data table that is used to plot the results. The information printed in the log by PROC OPTMILP is displayed in Output 7.3.1.

proc optmilp data=mycas.nofcdata primalout=mycas.nofcout;
r
run;
proc sql noprint;
  select put(sum(_objcoef_ * _value_),6.1) into :varcostNo
  from mycas.nofcout;
quit;

proc sql;
  create table CostNoFixedCharge_Data as
    select
      scan(p._var_,2, '[],') as customer,
      scan(p._var_,3, '[],') as site,
      c.x as x1, c.y as y1, s.x as x2, s.y as y2
    from

mycas.cdata as c,  
mycas.sdata as s,  
mycas.nofcout(where=(substr(_var_,1,6)='Assign' and  
round(_value_) = 1)) as p  
where calculated customer = c.name and calculated site = s.name;
quit;

Output 7.3.1  PROC OPTMILP Log for Facility Location with No Fixed Charges

NOTE: The problem nofcdta has 510 variables (510 binary, 0 integer, 0 free, 0 fixed).
NOTE: The problem has 560 constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem has 2010 constraint coefficients.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 10 variables and 500 constraints.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 500 variables, 60 constraints, and 1000 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1249.9063818</td>
<td>1177.1539196</td>
<td>6.18%</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
NOTE: The MILP presolver is applied again.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1249.9063818</td>
<td>1177.1539196</td>
<td>6.18%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1249.9063818</td>
<td>1190.5447409</td>
<td>4.99%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>7</td>
<td>1192.6272823</td>
<td>1192.6272823</td>
<td>0.00%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The MILP solver added 11 cuts with 255 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 1192.6272823.
NOTE: The Cloud Analytic Services server processed the request in 1.087645 seconds.
NOTE: The data set MYCAS.NOFCOUNT has 510 observations and 8 variables.
Example 7.3: Facility Location

Next, solve the fixed-charge model by using the following statements. Note that the solution to the model with no fixed charge is feasible for the fixed-charge model and should provide a good starting point for PROC OPTMILP. The PRIMALIN= option provides an incumbent solution (“warm start”). The two PROC SQL calls perform the same functions as in the case with no fixed charges. The results from this approach are shown in Output 7.3.2.

```plaintext
proc optmilp data=mycas.fcd data primalin=mycas.nofcout primalout=mycas.fcout;
run;
proc sql noprint;
   select put(sum(_objcoef_ * _value_), 6.1) into :varcost
       from mycas.fcout(where=(substr(_var_,1,6)='Assign'));
   select put(sum(_objcoef_ * _value_), 5.1) into :fixcost
       from mycas.fcout(where=(substr(_var_,1,5)='Build'));
   select put(sum(_objcoef_ * _value_), 6.1) into :totalcost
       from mycas.fcout;
quit;
proc sql;
   create table CostFixedCharge_Data as
       select
         scan(p._var_,2,'[]','') as customer,
         scan(p._var_,3,'[]','') as site,
         c.x as x1, c.y as y1, s.x as x2, s.y as y2
       from
         mycas.cdata as c,
         mycas.sdata as s,
         mycas.fcout(where=(substr(_var_,1,6)='Assign' and
             round(_value_)=1)) as p
       where calculated customer = c.name and calculated site = s.name;
quit;
```
Output 7.3.2  PROC OPTMILP Log for Facility Location with Fixed Charges, Using Warm Start

NOTE: The problem fcd ata has 510 variables (510 binary, 0 integer, 0 free, 0 fixed).

NOTE: The problem has 560 constraints (510 LE, 50 EQ, 0 GE, 0 range).

NOTE: The problem has 2010 constraint coefficients.

NOTE: The initial MILP heuristics are applied.

NOTE: The MILP presolver value AUTOMATIC is applied.

NOTE: The MILP presolver removed 0 variables and 0 constraints.

NOTE: The MILP presolver removed 0 constraint coefficients.

NOTE: The MILP presolver modified 0 constraint coefficients.

NOTE: The presolved problem has 510 variables, 560 constraints, and 2010 constraint coefficients.

NOTE: The MILP solver is called.

NOTE: The parallel Branch and Cut algorithm is used.

NOTE: The Branch and Cut algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>24086.8916716</td>
<td>0</td>
<td>24087</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>24086.8916716</td>
<td>19197.7909681</td>
<td>25.47%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>24086.8916716</td>
<td>19206.4233772</td>
<td>25.41%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>24086.8916716</td>
<td>19210.6350488</td>
<td>25.38%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>24086.8916716</td>
<td>19215.5729975</td>
<td>25.35%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
<td>21598.5625350</td>
<td>19223.0707237</td>
<td>12.36%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
<td>21598.5625350</td>
<td>19223.8657285</td>
<td>12.35%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>9</td>
<td>21558.5191802</td>
<td>19224.8824948</td>
<td>12.14%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>9</td>
<td>21558.5191802</td>
<td>19227.7927133</td>
<td>12.12%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>9</td>
<td>21558.5191802</td>
<td>19229.9736676</td>
<td>12.11%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The MILP solver added 24 cuts with 855 cut coefficients at the root.

| 100  | 23     | 10   | 21550.8232111 | 19835.9645633 | 8.65% | 0    |
| 133  | 18     | 11   | 21550.7090288 | 20010.7850290 | 7.70% | 0    |
| 163  | 28     | 12   | 21549.6785006 | 20158.5748939 | 6.90% | 0    |
| 172  | 24     | 13   | 21549.5643182 | 20385.0559999 | 5.71% | 0    |
| 242  | 49     | 14   | 21548.6159565 | 20438.0425736 | 5.43% | 0    |
| 256  | 54     | 15   | 21548.1764622 | 20438.0425736 | 5.43% | 0    |
| 402  | 37     | 15   | 21548.1764622 | 21546.0755057 | 0.01% | 0    |

NOTE: Optimal within relative gap.

NOTE: Objective = 21548.176462.

NOTE: The Cloud Analytic Services server processed the request in 1.956676 seconds.

NOTE: The data set MYCAS.FCOUT has 510 observations and 8 variables.
The following two SAS programs produce a plot of the solutions for both variants of the model, using data tables produced by PROC SQL from the PRIMALOUT= data tables produced by PROC OPTMILP.

```sas
title1 "Facility Location Problem";
title2 "TotalCost = &varcostNo (Variable = &varcostNo, Fixed = 0)";
data mycas.csdata;
   set mycas.cdata(rename=(y=cy)) mycas.sdata(rename=(y=sy));
run;
/* create Annotate data set to draw line between customer and */
/* assigned site */
data anno;
   retain function "line" drawspace "datavalue"
      linethickness 1 linecolor "black";
   set CostNoFixedCharge_Data(keep=x1 y1 x2 y2);
run;
proc sgplot data=mycas.csdata sganno=anno noautolegend;
   scatter x=x y=cy / datalabel=name datalabelattrs=(size=6pt)
      markerattrs=(symbol=circlefilled color=black size=6pt);
   scatter x=x y=sy / datalabel=name datalabelattrs=(size=6pt)
      markerattrs=(symbol=diamond color=blue size=6pt);
   xaxis display=(nolabel);
   yaxis display=(nolabel);
run;
```

The output from the first program appears in Output 7.3.3.
Output 7.3.3 Solution Plot for Facility Location with No Fixed Charges

```
title1 "Facility Location Problem";
title2 "TotalCost = &totalcost (Variable = &varcost, Fixed = &fixcost)";
/* create Annotate data set to draw line between customer and */
/* assigned site */
data anno;
  retain function "line" drawspace "datavalue"
    linethickness 1 linecolor "black";
  set CostFixedCharge_Data(keep=x1 y1 x2 y2);
run;
proc sgplot data=mycas.csdata sganno=anno noautolegend;
  scatter x=x y=cy / datalabel=name datalabelattrs=(size=6pt)
    markerattrs=(symbol=circlefilled color=black size=6pt);
  scatter x=x y=sy / datalabel=name datalabelattrs=(size=6pt)
    markerattrs=(symbol=diamond color=blue size=6pt);
  xaxis display=(nolabel);
  yaxis display=(nolabel);
run;
```

The output from the second program appears in Output 7.3.4.
Output 7.3.4 Solution Plot for Facility Location with Fixed Charges

The economic trade-off for the fixed-charge model forces you to build fewer sites and push more demand to each site.

Example 7.4: Scheduling

This example is intended for users who prefer to use the SAS DATA step, PROC SQL, and similar programming methods to prepare data for input to SAS/OR optimization procedures. SAS/OR users who prefer to use the algebraic modeling capabilities of PROC OPTMODEL to specify optimization models should consult Example 14.1 in Chapter 14, “The Mixed Integer Linear Programming Solver,” for a discussion of the same business problem in a PROC OPTMODEL context.

Scheduling is an application area where techniques in model generation can be valuable. Problems that involve scheduling are often solved with integer programming and are similar to assignment problems. In this example, you have eight one-hour time slots in each of five days. You have to assign four people to these time slots so that each slot is covered every day. You allow the people to specify preference data for each slot on each day. In addition, there are constraints that must be satisfied:

- Each person has some slots for which he is unavailable.
• Each person must have either slot 4 or 5 off for lunch.
• Each person can work only two time slots in a row.
• Each person can work only a specified number of hours in the week.

To formulate this problem, let \( i \) denote person, \( j \) denote time slot, and \( k \) denote day. Then, let \( x_{ijk} = 1 \) if person \( i \) is assigned to time slot \( j \) on day \( k \), and 0 otherwise; let \( p_{ijk} \) denote the preference of person \( i \) for slot \( j \) on day \( k \); and let \( h_i \) denote the number of hours in a week that person \( i \) will work. Then, you get

\[
\begin{align*}
\text{max} & \quad \sum_{ijk} p_{ijk} x_{ijk} \\
\text{subject to} & \quad \sum_i x_{ijk} = 1 \\
& \quad x_{i4k} + x_{i5k} \leq 1 \\
& \quad x_{i,\ell,k} + x_{i,\ell+1,k} + x_{i,\ell+2,k} \leq 2 \\
& \quad \sum_{jk} x_{ijk} \leq h_{i} \\
& \quad x_{ijk} = 0 \text{ or } 1
\end{align*}
\]

for all \( j \) and \( k \)
for all \( i \) and \( k \)
for all \( i \) and \( k \), and \( \ell = 1, \ldots, 6 \)
for all \( i \)
for all \( i \) and \( k \) such that \( p_{ijk} > 0 \),
otherwise \( x_{ijk} = 0 \)

To solve this problem, create a data set that has the hours and preference data for each individual, time slot, and day. A 10 represents the most desirable time slot, and a 1 represents the least desirable time slot. In addition, a 0 indicates that the time slot is not available.

```plaintext
data raw;
  input name dollar 20 hour slot mon tue wed thu fri;
datalines;
  marc 20 1 10 10 10 10 10 10 10 10
  marc 20 2 9 9 9 9 9 9 9 9
  marc 20 3 8 8 8 8 8 8 8 8
  marc 20 4 1 1 1 1 1 1 1 1
  marc 20 5 1 1 1 1 1 1 1 1
  marc 20 6 1 1 1 1 1 1 1 1
  marc 20 7 1 1 1 1 1 1 1 1
  marc 20 8 1 1 1 1 1 1 1 1
  mike 20 1 10 9 8 7 6 7 6 7 6
  mike 20 2 10 9 8 7 6 7 6 7 6
  mike 20 3 10 9 8 7 6 7 6 7 6
  mike 20 4 10 3 3 3 3 3 3 3 3
  mike 20 5 1 1 1 1 1 1 1 1 1
  mike 20 6 1 2 3 4 5 4 5 4 5
  mike 20 7 1 2 3 4 5 4 5 4 5
  mike 20 8 1 2 3 4 5 4 5 4 5
  bill 20 1 10 10 10 10 10 10 10 10 10
  bill 20 2 9 9 9 9 9 9 9 9 9
  bill 20 3 8 8 8 8 8 8 8 8 8
  bill 20 4 0 0 0 0 0 0 0 0 0
  bill 20 5 1 1 1 1 1 1 1 1 1
  bill 20 6 1 1 1 1 1 1 1 1 1
  bill 20 7 1 1 1 1 1 1 1 1 1
  bill 20 8 1 1 1 1 1 1 1 1 1
  bob 20 1 10 9 8 7 6 7 6 7 6
  bob 20 2 10 9 8 7 6 7 6 7 6
  bob 20 3 10 9 8 7 6 7 6 7 6
```
```
Example 7.4: Scheduling

These data are read by the following DATA step, and an integer program is built to solve the problem. The model is saved in the data set named MODEL, which is constructed in the following steps:

1. The objective function is built using the data saved in the RAW data set.
2. The constraints that ensure that no one works during a time slot during which he is unavailable are built.
3. The constraints that require a person to be working in each time slot are built.
4. The constraints that allow each person time for lunch are added.
5. The constraints that restrict people to only two consecutive hours are added.
6. The constraints that limit the time that any one person works in a week are added.

The statements to build each of these constraints follow the formulation closely.

```plaintext
data model;
    array workweek{5} mon tue wed thu fri;
    array hours{4} hours1 hours2 hours3 hours4;
    retain hours1-hours4;

    set raw end=eof;

    length _row_ $ 8 _col_ $ 8 _type_ $ 8;
    keep _type_ _col_ _row_ _coef_;

    if name='marc' then i=1;
    else if name='mike' then i=2;
    else if name='bill' then i=3;
    else if name='bob' then i=4;

    hours(i)=hour;

    /* build the objective function */
    do k=1 to 5;
        _col_='x'||put(i,1.)||put(slot,1.)||put(k,1.);
        _row_='object';
        _coef_=workweek{k} * 1000;
        output;
    end;

    /* build the rest of the model */

    /* cannot work during unavailable slots */
    do k=1 to 5;
```
if workweek(k)=0 then do;
    _row_='off'||put(i,1.)||put(slot,1.)||put(k,1.);
    _type_='eq';
    _col_='RHS_';
    _coef_=0;
    output;
    _col_='x'||put(i,1.)||put(slot,1.)||put(k,1.);
    _coef_=1;
    _type_=' ';  
    output;
end;
end;

if eof then do;
    _coef_=.;
    _col_=' ';
end;

/* every hour 1 person working */
do j=1 to 8;
do k=1 to 5;
    _row_='work'||put(j,1.)||put(k,1.);
    _type_='eq';
    _col_='RHS_';
    _coef_=1;
    output;
    _coef_=1;
    _type_=' ';
    _col_='x'||put(i,1.)||put(j,1.)||put(k,1.);
    output;
end;
end;

/* each person has a lunch */
do i=1 to 4;
do k=1 to 5;
    _row_='lunch'||put(i,1.)||put(k,1.);
    _type_='le';
    _col_='RHS_';
    _coef_=1;
    output;
    _coef_=1;
    _type_=' ';
    _col_='x'||put(i,1.)||put(k,1.);  
    output;
    _col_='x'||put(i,1.)||put(k,1.);  
    output;
end;
end;

/* work at most 2 slots in a row */
do i=1 to 4;
do k=1 to 5;
do l=1 to 6;
  _row_='seq'||put(i,1.)||put(k,1.)||put(l,1.);
  _type_='le';
  _col_='RHS_';
  _coef_=2;
  output;
  _coef_=1;
  _type_=' ';
  do j=0 to 2;
    _col_='x'||put(i,1.)||put(l+j,1.)||put(k,1.);
    output;
  end;
end;
end;
end;

/* work at most n hours in a week */
do i=1 to 4;
  _row_='capacit'||put(i,1.);
  _type_='le';
  _col_='RHS_';
  _coef=hours{i};
  output;
  _coef=1;
  _type_=' ';
  do j=1 to 8;
    do k=1 to 5;
      _col_='x'||put(i,1.)||put(j,1.)||put(k,1.);
      output;
    end;
  end;
end;
end;
run;

Next, this SAS data set is converted to an MPS-format data table by establishing the structure of the MPS format and through very minor conversions of the data.

/* the following code transforms the above sparse data set */
/* into an MPS-format data set */

/* generate the header of the MPS-format data set */
data mps0;
  format field1 field2 field3 $10.;
  format field4 10.;
  format field5 $10.;
  format field6 10.;
  field1 = 'NAME';
  field2 = ' ';
  field3 = 'PROBLEM';
  field4 = .;
  field5 = ' ';
  field6 = .;
  output;
field1 = 'ROWS';
field3 = '';
output;
field1 = 'MAX';
field2 = 'object';
field3 = '';
output;
run;

/* generate rows */
proc sql;
create table mps1 as
  select _type_ as field1, _row_ as field2 from model
  where _row_ eq 'object' and _type_ ne '' union
  select 'E' as field1, _row_ as field2 from model
  where _type_ eq 'eq' union
  select 'L' as field1, _row_ as field2 from model
  where _type_ eq 'le' union
  select 'G' as field1, _row_ as field2 from model
  where _type_ eq 'ge';
quit;

/* indicate start of columns section and declare type of all */
data mps2;
  format field1 field2 field3 $10.;
  format field4 10.;
  format field5 $10.;
  format field6 10.;
  field1 = 'COLUMNS';
  field2 = '';
  field3 = '';
  field4 = .;
  field5 = 'MARK0';
  field6 = .;
output;
  field1 = '
  field2 = '.MARK0';
  field3 = "MARKER";
  field4 = .;
  field5 = "INTORG";
  field6 = .;
output;
run;

/* generate columns */
data mps3;
  set model;
  format field1 field2 field3 field $10.;
  format field4 10.;
  format field5 $10.;
  format field6 10.;
  keep field1-field6;
Example 7.4: Scheduling F 217

field1 = '
';
field2 = _col_;
field3 = _row_;
field4 = _coef_;
field5 = '
';
field6 = .;
if field2 ne '_RHS_' then output;
run;
/* sort columns by variable names */
proc sort data=mps3;
by field2;
run;
/* indicate the end of the columns section */
data mps4;
format field1 field2 field3 $10.;
format field4 10.;
format field5 $10.;
format field6 10.;
field1 = '
';
field2 = '.MARK1';
field3 = "'MARKER'";
field4 = .;
field5 = "'INTEND'";
field6 = .;
output;
run;
/* indicate the start of the RHS section */
data mps5;
format field1 field2 field3 $10.;
format field4 10.;
format field5 $10.;
format field6 10.;
field1 = 'RHS';
run;
/* generate RHS entries */
data mps6;
set model;
format field1 field2 field3 $10.;
format field4 10.;
format field5 $10.;
format field6 10.;
keep field1-field6;
field1 = '
';
field2 = _col_;
field3 = _row_;
field4 = _coef_;
field5 = '
';
field6 = .;
if field2 eq '_RHS_' then output;
run;


The model is solved using the OPTMILP procedure. The option PRIMALOUT=SOLUTION causes PROC OPTMILP to save the primal solution in the data table named SOLUTION.

The following DATA step takes the solution data table SOLUTION and generates a report data table named ASSIGNMENTS. It restores the original interpretation (person, shift, day) of the variable names $x_{ijk}$ so that a more meaningful report can be written. Then a data step and PROC PRINT are used to display a schedule that shows how the eight time slots are covered for the week.
Example 7.4: Scheduling

/* report the solution */
title 'Reported Solution';

data mycas.assignments;
  set mycas.solution;
  keep slot mon tue wed thu fri;
  if substr(_var_,1,1)='x' then do;
    if _value_>0 then do;
      n=input(substr(_var_,2,1), 8.);
      slot=input(substr(_var_,3,1), 8.);
      d=substr(_var_,4,1);
      if d='1' then mon=n;
      else if d='2' then tue=n;
      else if d='3' then wed=n;
      else if d='4' then thu=n;
      else fri=n;
    output;
  end;
end;
run;

proc format;
  value namefmt 1='Marc' 2='Mike' 3='Bill' 4='Bob' .= ' ';
run;

data report;
  do until (last.slot);
    set mycas.assignments;
    by slot notsorted;
    if mon ne . then Monday = mon;
    if tue ne . then Tuesday = tue;
    if wed ne . then Wednesday = wed;
    if thu ne . then Thursday = thu;
    if fri ne . then Friday = fri;
  end;
  drop mon tue wed thu fri;
  format Monday Tuesday Wednesday Thursday Friday namefmt.;
run;

proc print data=report;
  id slot;
run;

Output 7.4.1 summarizes the schedule. Notice that the constraint that requires a person to be assigned to each possible time slot on each day is satisfied.
Recall that PROC OPTMILP puts a character string in the macro variable _OROPTMILP_ that describes the characteristics of the solution on termination. This string can be parsed using macro functions, and the information obtained can be used in report writing. The variable can be written to the log with the following command:

```plaintext
%put &_OROPTMILP;
```

This command produces the output shown in Output 7.4.2.

### Output 7.4.2 _OROPTMILP_ Macro Variable

```
STATUS=OK ALGORITHM=BAC SOLUTION_STATUS=OPTIMAL OBJECTIVE=211000 RELATIVE_GAP=0
ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0
INTEGER_INFEASIBILITY=0 BEST_BOUND=211000 NODES=1 ITERATIONS=63
PRESOLVE_TIME=0.00 SOLUTION_TIME=0.01
```

From this output you learn, for example, that at termination the solution is integer-optimal and has an objective value of 211,000.
References


Chapter 8
The OPTQP Procedure

Overview: OPTQP Procedure

The OPTQP procedure solves quadratic programs—problems with quadratic objective function and a collection of linear constraints, including lower or upper bounds (or both) on the decision variables.

Mathematically, a quadratic programming (QP) problem can be stated as follows:

\[
\begin{align*}
\min \quad & \frac{1}{2} x^T Q x + c^T x \\
\text{subject to} \quad & A x \ {\geq, =, \leq} \ b \\
& 1 \leq x \leq u
\end{align*}
\]

where
The quadratic matrix $Q$ is assumed to be symmetric; that is,

$$q_{ij} = q_{ji}, \quad \forall i, j = 1, \ldots, n$$

Indeed, it is easy to show that even if $Q \neq Q^T$, the simple modification

$$\tilde{Q} = \frac{1}{2}(Q + Q^T)$$

produces an equivalent formulation $x^TQx \equiv x^T\tilde{Q}x$; hence symmetry is assumed. When you specify a quadratic matrix, it suffices to list only lower triangular coefficients.

In addition to being symmetric, $Q$ is also required to be positive semidefinite,

$$x^TQx \geq 0, \quad \forall x \in \mathbb{R}^n$$

for minimization type of models; it is required to be negative semidefinite for the maximization type of models. Convexity can come as a result of a matrix-matrix multiplication

$$Q = LL^T$$

or as a consequence of physical laws, and so on. See Figure 8.1 for examples of convex, concave, and nonconvex objective functions.

The order of constraints is insignificant. Some or all components of $l$ or $u$ (lower and upper bounds, respectively) can be omitted.
Consider a small illustrative example. Suppose you want to minimize a two-variable quadratic function \( f(x_1, x_2) \) on the nonnegative quadrant, subject to two constraints:

\[
\begin{align*}
\min & \quad 2x_1 + 3x_2 + x_1^2 + 10x_2^2 + 2.5x_1x_2 \\
\text{subject to} & \quad x_1 - x_2 \leq 1 \\
& \quad x_1 + 2x_2 \geq 100 \\
& \quad x_1 \geq 0 \\
& \quad x_2 \geq 0
\end{align*}
\]

The linear objective function coefficients, vector of right-hand sides, and lower and upper bounds are identified immediately as

\[
c = \begin{bmatrix} 2 \\ 3 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 100 \end{bmatrix}, \quad l = \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \quad u = \begin{bmatrix} +\infty \\ +\infty \end{bmatrix}
\]

Carefully construct the quadratic matrix \( Q \). Observe that you can use symmetry to separate the main-diagonal and off-diagonal elements:

\[
\frac{1}{2}x^TQx = \frac{1}{2} \sum_{i,j=1}^{n} x_i q_{ij} x_j = \frac{1}{2} \sum_{i=1}^{n} q_{ii} x_i^2 + \sum_{i>j} x_i q_{ij} x_j
\]
Chapter 8: The OPTQP Procedure

The first expression
\[ \frac{1}{2} \sum_{i=1}^{n} q_{ii} x_i^2 \]
sums the main-diagonal elements. Thus, in this case you have
\[ q_{11} = 2, \quad q_{22} = 20 \]
Notice that the main-diagonal values are doubled in order to accommodate the 1/2 factor. Now the second term
\[ \sum_{i>j} x_i q_{ij} x_j \]
sums the off-diagonal elements in the strict lower triangular part of the matrix. The only off-diagonal \((x_i x_j, i \neq j)\) term in the objective function is \(2.5 x_1 x_2\), so you have
\[ q_{21} = 2.5 \]
Notice that you do not need to specify the upper triangular part of the quadratic matrix.

Finally, the matrix of constraints is as follows:

\[
A = \begin{bmatrix}
1 & -1 \\
1 & 1 \\
1 & 2 \\
\end{bmatrix}
\]

The input data table with a quadratic programming system (QPS) format for the preceding problem can be expressed in the following manner:

```plaintext
data gsdata;
  input field1 $ field2 $ field3 $ field4 field5 $ field6 @;
datalines;
  NAME . EXAMPLE . . . .
  ROWS . . . . . .
  N OBJ . . . . . .
  L R1 . . . . . .
  G R2 . . . . . .
  COLUMNS . . . . . .
    . X1 R1 1.0 R2 1.0
    . X1 OBJ 2.0 . .
    . X2 R1 -1.0 R2 2.0
    . X2 OBJ 3.0 . .
  RHS . . . . . .
    . RHS R1 1.0 . .
    . RHS R2 100 . .
  RANGES . . . . . .
  BOUNDS . . . . . .
  QUADOBJ . . . . . .
    . X1 X1 2.0 . .
    . X1 X2 2.5 . .
    . X2 X2 20 . .
ENDATA . . . . . .
;
```
For more information about the QPS-format data table, see Chapter 9, “The MPS-Format Data Table.”

Alternatively, if you have a QPS-format flat file named gs.qps, then the following call to the SAS macro %MPS2SASD translates that file into a SAS data set named gsdata:

```sas
%mps2sasd(mpsfile =gs.qps, outdata = gsdata);
```

**NOTE:** The SAS macro %MPS2SASD is provided in SAS/OR software. For more information, see the section “Converting an MPS/QPS-Format File: %MPS2SASD” on page 264.

To make the created data set available in the CAS server, you need to upload it as shown in the following DATA step. At the same time, you also need to add an _id_ column to ensure that the MPS reader can read the file in the correct order even if it is distributed across several machines and hence loses its original ordering.

For more information about the MPS format, see Chapter 9, “The MPS-Format Data Table.”

```sas
data mycas.gsdata;
    set gsdata;
    _id_ = _n_;  
run;
```

Alternatively, you can create the MPS data table directly in CAS and include the _id_ column in the datalines input.

You can use the following call to PROC OPTQP:

```sas
proc optqp data = mycas.gsdata
    primalout = mycas.gspout
    dualout = mycas.gsdout;
run;
```

The procedure output is displayed in Figure 8.2.
## Figure 8.2 Procedure Output

### The OPTQP Procedure

#### Problem Summary

<table>
<thead>
<tr>
<th>Problem Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Name</td>
<td>EXAMPLE</td>
</tr>
<tr>
<td>Objective Sense</td>
<td>Minimization</td>
</tr>
<tr>
<td>Objective Function</td>
<td>OBJ</td>
</tr>
<tr>
<td>RHS</td>
<td>RHS</td>
</tr>
<tr>
<td>Number of Variables</td>
<td>2</td>
</tr>
<tr>
<td>Bounded Above</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below</td>
<td>2</td>
</tr>
<tr>
<td>Bounded Above and Below</td>
<td>0</td>
</tr>
<tr>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>Fixed</td>
<td>0</td>
</tr>
<tr>
<td>Number of Constraints</td>
<td>2</td>
</tr>
<tr>
<td>LE ((&lt;=))</td>
<td>1</td>
</tr>
<tr>
<td>EQ ((=))</td>
<td>0</td>
</tr>
<tr>
<td>GE ((&gt;=))</td>
<td>1</td>
</tr>
<tr>
<td>Range</td>
<td>0</td>
</tr>
<tr>
<td>Constraint Coefficients</td>
<td>4</td>
</tr>
<tr>
<td>Hessian Diagonal Elements</td>
<td>2</td>
</tr>
<tr>
<td>Hessian Elements Below Diagonal</td>
<td>1</td>
</tr>
</tbody>
</table>

#### Solution Summary

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>QP</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Interior Point</td>
</tr>
<tr>
<td>Objective Function</td>
<td>OBJ</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Optimal</td>
</tr>
<tr>
<td>Objective Value</td>
<td>15018</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Dual Infeasibility</td>
<td>2.468474E-14</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Duality Gap</td>
<td>1.211126E-16</td>
</tr>
<tr>
<td>Complementarity</td>
<td>0</td>
</tr>
<tr>
<td>Iterations</td>
<td>6</td>
</tr>
<tr>
<td>Presolve Time</td>
<td>0.00</td>
</tr>
<tr>
<td>Solution Time</td>
<td>0.04</td>
</tr>
</tbody>
</table>

#### Output CAS Tables

<table>
<thead>
<tr>
<th>CAS Library</th>
<th>Name</th>
<th>Number of Rows</th>
<th>Number of Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASUSERHDFS(amlari)</td>
<td>GSPOUT</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>CASUSERHDFS(amlari)</td>
<td>GSDOUT</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>
The optimal primal solution is stored in the data table `gspout` and is displayed in Figure 8.3.

```sas
proc print data = mycas.gspout label;
run;
```

![Figure 8.3 Optimal Solution](image)

<table>
<thead>
<tr>
<th>Obs ID</th>
<th>Objective Function ID</th>
<th>RHS Variable Name</th>
<th>Variable Type</th>
<th>Linear Objective Coefficient</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
<th>Variable Value</th>
<th>Variable Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OBJ</td>
<td>RHS X1</td>
<td>N</td>
<td>2</td>
<td>0</td>
<td>1.7977E308</td>
<td>34</td>
<td>O</td>
</tr>
<tr>
<td>2</td>
<td>OBJ</td>
<td>RHS X2</td>
<td>N</td>
<td>3</td>
<td>0</td>
<td>1.7977E308</td>
<td>33</td>
<td>O</td>
</tr>
</tbody>
</table>

The SAS log shown in Figure 8.4 provides information about the problem, convergence information after each iteration, and the optimal objective value.
Figure 8.4 Iteration Log

NOTE: The problem EXAMPLE has 2 variables (0 free, 0 fixed).
NOTE: The problem has 2 constraints (1 LE, 0 EQ, 1 GE, 0 range).
NOTE: The problem has 4 constraint coefficients.
NOTE: The objective function has 2 Hessian diagonal elements and 1 Hessian elements above the diagonal.
NOTE: The QP presolver value AUTOMATIC is applied.
NOTE: The QP presolver removed 0 variables and 0 constraints.
NOTE: The QP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 2 variables, 2 constraints, and 4 constraint coefficients.
NOTE: The QP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Interior Point algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Iter</th>
<th>Primal</th>
<th>Bound</th>
<th>Dual</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Complement</td>
<td>Duality Gap</td>
<td>Infeas</td>
<td>Infeas</td>
</tr>
<tr>
<td>0</td>
<td>3.5863E+03</td>
<td>4.8823E+00</td>
<td>1.0251E+00</td>
<td>1.0354E+02</td>
</tr>
<tr>
<td>1</td>
<td>1.9455E+03</td>
<td>9.6064E-01</td>
<td>4.4173E-01</td>
<td>4.4617E+01</td>
</tr>
<tr>
<td>2</td>
<td>2.2355E+03</td>
<td>1.2412E-01</td>
<td>4.4173E-03</td>
<td>4.4617E-01</td>
</tr>
<tr>
<td>3</td>
<td>5.1970E+01</td>
<td>3.3564E-03</td>
<td>4.4173E-05</td>
<td>4.4617E-03</td>
</tr>
<tr>
<td>4</td>
<td>5.1942E-01</td>
<td>3.3642E-05</td>
<td>4.4173E-07</td>
<td>4.4617E-05</td>
</tr>
<tr>
<td>5</td>
<td>5.1946E-03</td>
<td>3.3646E-07</td>
<td>4.4173E-09</td>
<td>4.4617E-07</td>
</tr>
<tr>
<td>6</td>
<td>0.0000E+00</td>
<td>0.0000E+00</td>
<td>1.5730E-16</td>
<td>0.0000E+00</td>
</tr>
</tbody>
</table>

NOTE: Optimal.
NOTE: Objective = 15018.
NOTE: The Interior Point solve time is 0.02 seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.219302 seconds.
NOTE: The data set MYCAS.GSPOUT has 2 observations and 9 variables.
NOTE: The data set MYCAS.GSDOUT has 2 observations and 10 variables.

See the section “Interior Point Algorithm: Overview” on page 237 and the section “Iteration Log for the OPTQP Procedure” on page 239 for more details about convergence information given by the iteration log.

Syntax: OPTQP Procedure

The following statements are available in the OPTQP procedure:

PROC OPTQP <options> ;
Functional Summary

Table 8.1 outlines the options available for the OPTQP procedure classified by function.

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Table Options</td>
<td></td>
</tr>
<tr>
<td>Specifies a QPS-format input data table</td>
<td>DATA=</td>
</tr>
<tr>
<td>Specifies a dual solution output data table</td>
<td>DUALOUT=</td>
</tr>
<tr>
<td>Specifies whether the QP model is a maximization or minimization problem</td>
<td>OBJSENSE=</td>
</tr>
<tr>
<td>Specifies the primal solution output data table</td>
<td>PRIMALOUT=</td>
</tr>
<tr>
<td>Solver Options</td>
<td></td>
</tr>
<tr>
<td>Enables or disables IIS detection</td>
<td>IIS=</td>
</tr>
<tr>
<td>Control Options</td>
<td></td>
</tr>
<tr>
<td>Specifies the stopping criterion based on duality gap</td>
<td>DUALITYGAP=</td>
</tr>
<tr>
<td>Specifies the dual feasibility tolerance</td>
<td>DUALTOL=</td>
</tr>
<tr>
<td>Specifies how often to print the solution progress</td>
<td>LOGFREQ=</td>
</tr>
<tr>
<td>Specifies how much solution progress detail to print in log</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the maximum number of iterations</td>
<td>MAXITER=</td>
</tr>
<tr>
<td>Specifies the time limit for the optimization process</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Specifies the maximum number of threads</td>
<td>NTHREADS=</td>
</tr>
<tr>
<td>Specifies the parallel processing mode</td>
<td>PARALLELMODE=</td>
</tr>
<tr>
<td>Specifies the primal feasibility tolerance</td>
<td>PRIMALTOL=</td>
</tr>
<tr>
<td>Enables or disables printing summary</td>
<td>PRINTLEVEL=</td>
</tr>
<tr>
<td>Specifies units of CPU time or real time</td>
<td>TIMETYPE=</td>
</tr>
</tbody>
</table>

PROC OPTQP Statement

The following options can be specified in the PROC OPTQP statement.

**DATA=** *CAS-libref.data-table*

specifies the input data table. This data table can also be created from a QPS-format flat file by using the SAS macro %MPS2SASD. For more information, see Chapter 9, “The MPS-Format Data Table.” If the DATA= option is not specified, PROC OPTQP uses the most recently created data table.

**DUALITYGAP=** $\delta$

specifies the desired relative duality gap, $\delta \in [1E-9, 1E-4]$. This is the relative difference between the primal and dual objective function values and is the primary solution quality parameter. For more information, see the section “Interior Point Algorithm: Overview” on page 237. The default value is $1E-6$. 
**DUALOUT=** `CAS-libref.data-table`  
**DOUT=** `CAS-libref.data-table`  
- Specifies the output data table to contain the dual solution. For more information, see the section “Output Data Tables” on page 235.

**DUALTOL=** $\beta$  
**OPTTOL=** $\beta$  
- Specifies the maximum relative dual constraints violation, $\beta \in [1E^{-9}, 1E^{-4}]$. For more information, see the section “Interior Point Algorithm: Overview” on page 237. The default value is $1E^{-6}$.

**IIS=FALSE | TRUE**  
- Specifies whether to attempt to identify a set of constraints and variables that form an irreducible infeasible set (IIS). You can specify the following values:

  - **FALSE**: Disables IIS detection.  
  - **TRUE**: Enables IIS detection.

If an IIS is found, information about infeasible constraints or variable bounds is written to the data tables that are specified in the DUALOUT= and PRIMALOUT= options. For more information, see the section “Irreducible Infeasible Set” on page 244. By default, IIS=FALSE.

**LOGFREQ=** $k$  
**PRINTFREQ=** $k$  
- Prints the solution progress to the iteration log after every $k$ iterations, where $k$ is an integer between 0 and the largest four-byte, signed integer, which is $2^{31} - 1$. The value $k = 0$ suppresses printing of the progress of the solution. By default, LOGFREQ=1.

**LOGLEVEL=** **NONE | BASIC | MODERATE | AGGRESSIVE**  
**PRINTLEVEL2=** **NONE | BASIC | MODERATE | AGGRESSIVE**  
- Controls the amount of information displayed in the SAS log. You can specify the following values:

  - **NONE**: Turns off all solver-related messages in the SAS log.  
  - **BASIC**: Displays a solver summary after stopping.  
  - **MODERATE**: Prints a solver summary and an iteration log by using the interval specified in the LOGFREQ= option.  
  - **AGGRESSIVE**: Prints a detailed solver summary and an iteration log by using the interval specified in the LOGFREQ= option.

By default, LOGLEVEL=MODERATE.

**MAXITER=** $k$  
- Specifies the maximum number of predictor-corrector iterations performed by the interior point algorithm (see the section “Interior Point Algorithm: Overview” on page 237). The value $k$ is an integer between 1 and the largest four-byte, signed integer, which is $2^{31} - 1$. If you do not specify this option, the procedure does not stop based on the number of iterations performed.
**PROC OPTQP Statement**

**MAXTIME=** \( t \)

specifies an upper limit of \( t \) seconds of time for reading in the data and performing the optimization process. The value of the TIMETYPE= option determines the type of units used. If you do not specify this option, the procedure does not stop based on the amount of time elapsed. The value of \( t \) can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

**NTHREADS=** \( k \)

**NUMTHREADS=** \( k \)

specifies the number of threads that PROC OPTQP can use, where \( k \) can be any integer between 1 and 256, inclusive. This option overrides the THREADS | NOTHREADS SAS system option. The default value of this option is the value of the CPUCOUNT= SAS system option.

Specifying \( k \) as a number greater than the actual number of available cores might result in reduced performance. Specifying a high value for \( k \) does not guarantee shorter solution time; the actual change in solution time depends on the computing hardware and the scalability of the underlying algorithms in PROC OPTQP. In some circumstances, PROC OPTQP might use fewer than \( k \) threads because the procedure’s internal algorithms have determined that a smaller number is preferable.

**OBJSENSE=** \( \text{MIN} \) | \( \text{MAX} \)

specifies whether the QP model is a minimization or maximization problem. You can specify the following values:

- **MIN**
  - treat the QP model as a minimization problem.

- **MAX**
  - treat the QP model as a maximization problem.

Alternatively, you can specify the objective sense in the input data table; for more information, see the section “ROWS Section” on page 257. This option supersedes any objective sense that is specified in the input data table. If the objective sense is not specified anywhere, then PROC OPTQP interprets and solves the quadratic program as a minimization problem.

**PARALLELMODE=** \( \text{DETERMINISTIC} \) | \( \text{NONDETERMINISTIC} \)

specifies the parallel processing mode. This mode determines the solution results that are obtained from running the same model with the same option values on the same platform multiple times. You can specify the following values:

- **DETERMINISTIC**
  - requires algorithms to produce the same results every time.

- **NONDETERMINISTIC**
  - permits algorithms to produce different solution results. This mode requires less synchronization and might attain better performance than DETERMINISTIC mode.

By default, PARALLELMODE=DETERMINISTIC.

**PRESOLVER=** \( \text{AUTOMATIC} \) | \( \text{NONE} \) | \( \text{BASIC} \) | \( \text{MODERATE} \) | \( \text{AGGRESSIVE} \)

**PRESOL=** \( \text{AUTOMATIC} \) | \( \text{NONE} \) | \( \text{BASIC} \) | \( \text{MODERATE} \) | \( \text{AGGRESSIVE} \)

specifies the presolve level. You can specify the following values:
Chapter 8: The OPTQP Procedure

AUTOMATIC applies the presolver by using default setting.
NONE disables the presolver.
BASIC applies the basic presolver.
MODERATE applies the moderate presolver.
AGGRESSIVE applies the aggressive presolver.

By default, PRESOLVER=AUTOMATIC.

PRIMALOUT=\texttt{CAS-libref.data-table}

POUT=\texttt{CAS-libref.data-table}

specifies the output data table to contain the primal solution. For more information, see the section “Output Data Tables” on page 235.

PRIMALTOL=\alpha
FEAUSTOL=\alpha

specifies the maximum relative bound and primal constraints violation, \(\alpha \in [1\!E^{-9}, \!1\!E^{-4}]\). For more information, see the section “Interior Point Algorithm: Overview” on page 237. The default value is 1E–6.

PRINTLEVEL=0 | 1 | 2

specifies whether to print summary Output Delivery System (ODS) tables of the problem and solution. You can specify the following values:

0 does not produce or print any ODS tables.
1 produces and prints the following ODS tables: ProblemSummary, SolutionSummary, and optional OutputCasTables.
2 produces and prints the following ODS tables: ProblemSummary, SolutionSummary, ProblemStatistics, Timing, and optional OutputCasTables.

For more information about the ODS tables that PROC OPTQP creates, see the section “ODS Tables” on page 240. By default, PRINTLEVEL=1.

TIMETYPE=CPU | REAL

specifies whether CPU time or real time is used for the MAXTIME= option and the \_OROPTQP\_ macro variable in a PROC OPTQP call. You can specify the following values:

CPU specifies that units are in CPU time.
REAL specifies that units are in real time.

By default, TIMETYPE=REAL.
Details: OPTQP Procedure

Output Data Tables

This section describes the output data tables that are specified in the PRIMALOUT= and DUALOUT= options.

Definitions of Variables in the PRIMALOUT= Data Table

The PRIMALOUT= data table contains the primal solution to the quadratic programming (QP) model. The variables in the data table have the following names and meanings.

_]OBJ_ID_ specifies the name of the objective function. Naming objective functions is particularly useful when there are multiple objective functions, in which case each objective function has a unique name. For more information, see the section “ROWS Section” on page 257.

**NOTE:** PROC OPTQP does not support simultaneous optimization of multiple objective functions in this release.

_]RHS_ID_ specifies the name of the variable that contains the right-hand-side value of each constraint. For more information, see the section “RHS Section (Optional)” on page 259.

_]VAR_ specifies the name of the decision variable.

_]TYPE_ specifies the type of the decision variable. _]TYPE_ can take one of the following values:

N nonnegative variable
D bounded variable (with both lower and upper bound)
F free variable
X fixed variable
O other (with either lower or upper bound)

_]OBJCOEF_ specifies the coefficient of the decision variable in the linear component of the objective function.

_]LBOUND_ specifies the lower bound on the decision variable.

_]UBOUND_ specifies the upper bound on the decision variable.
Chapter 8: The OPTQP Procedure

.VALUE_
specifies the value of the decision variable.

.STATUS_
specifies the status of the decision variable. _STATUS_ can indicate one of the following two cases:

  O   The QP problem is optimal.
  I   The QP problem could be infeasible or unbounded, or PROC OPTQP was not able to solve the problem.

The following values can appear only if IIS=TRUE. For more information, see the section “Irreducible Infeasible Set” on page 244.

  I_L   The lower bound of the variable is needed for the IIS.
  I_U   The upper bound of the variable is needed for the IIS.
  I_F   Both bounds of the variable are needed for the IIS (the variable is fixed or has conflicting bounds).

Definitions of Variables in the DUALOUT= Data Table

The DUALOUT= data table contains the dual solution to the QP model. Information about the objective rows of the QP problems is not included. The variables in the data table have the following names and meanings.

.OBJ_ID_
specifies the name of the objective function. Naming objective functions is particularly useful when there are multiple objective functions, in which case each objective function has a unique name. For more information, see the section “ROWS Section” on page 257.

.NOTE: PROC OPTQP does not support simultaneous optimization of multiple objective functions in this release.

.RHS_ID_
specifies the name of the variable that contains the right-hand-side value of each constraint. For more information, see the section “RHS Section (Optional)” on page 259.

.ROW_
specifies the name of the constraint. For more information, see the section “ROWS Section” on page 257.

.TYPE_
specifies the type of the constraint. _TYPE_ can take one of the following values:

  L   “less than or equals” constraint
  E   equality constraint
  G   “greater than or equals” constraint
  R   ranged constraint (both “less than or equals” and “greater than or equals”)

For more information, see the sections “ROWS Section” on page 257 and “RANGES Section (Optional)” on page 260.
Interior Point Algorithm: Overview

The interior point solver in PROC OPTQP implements an infeasible primal-dual predictor-corrector interior point algorithm. To illustrate the algorithm and the concepts of duality and dual infeasibility, consider the following QP formulation (the primal):

$$
\begin{align*}
\text{min} & \quad \frac{1}{2} x^T Q x + c^T x \\
\text{subject to} & \quad A x \geq b \\
& \quad x \geq 0
\end{align*}
$$

The corresponding dual is as follows:

$$
\begin{align*}
\text{max} & \quad -\frac{1}{2} x^T Q x + b^T y \\
\text{subject to} & \quad -Q x + A^T y + w = c \\
& \quad y \geq 0 \\
& \quad w \geq 0
\end{align*}
$$

_RHS_

specifies the value of the right-hand side of the constraint. It takes a missing value for a ranged constraint.

_L_RHS_

specifies the lower bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

_U_RHS_

specifies the upper bound of a ranged constraint. It takes a missing value for a non-ranged constraint.

_VALUE_

specifies the value of the dual variable associated with the constraint.

_STATUS_

specifies the status of the constraint. _STATUS_ can indicate one of the following two cases:

O The QP problem is optimal.

I The QP problem could be infeasible or unbounded, or PROC OPTQP was not able to solve the problem.

The following values can appear only if option IIS=TRUE. For more information, see the section “Irreducible Infeasible Set” on page 244.

I_L The “GE” (≥) condition of the constraint is needed for the IIS.

I_U The “LE” (≤) condition of the constraint is needed for the IIS.

I_F Both conditions of the constraint are needed for the IIS (the constraint is an equality or a range constraint with conflicting bounds).

_ACTIVITY_

specifies the value of a constraint. In other words, the value of _ACTIVITY_ for the ith constraint is equal to \(a_i^T x\), where \(a_i\) refers to the ith row of the constraints matrix and \(x\) denotes the vector of current decision variable values.
where \( y \in \mathbb{R}^m \) refers to the vector of dual variables and \( w \in \mathbb{R}^n \) refers to the vector of slack variables in the dual problem.

The dual makes an important contribution to the certificate of optimality for the primal. The primal and dual constraints combined with complementarity conditions define the first-order optimality conditions, also known as KKT (Karush-Kuhn-Tucker) conditions, which can be stated as follows:

\[
\begin{align*}
Ax - s &= b \quad \text{(primal feasibility)} \\
-Qx + A^T y + w &= c \quad \text{(dual feasibility)} \\
WXe &= 0 \quad \text{(complementarity)} \\
SYe &= 0 \quad \text{(complementarity)}
\end{align*}
\]

where \( e \equiv (1, \ldots, 1)^T \) is of appropriate dimension and \( s \in \mathbb{R}^m \) is the vector of primal slack variables.

**Note:** Slack variables (the \( s \) vector) are automatically introduced by the solver when necessary; it is therefore recommended that you not introduce any slack variables explicitly. This enables the solver to handle slack variables much more efficiently.

The letters \( X, Y, W, \text{ and } S \) denote matrices with corresponding \( x, y, w, \) and \( s \) on the main diagonal and zero elsewhere, as in the following example:

\[
X \equiv \begin{bmatrix}
x_1 & 0 & \cdots & 0 \\
0 & x_2 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & x_n
\end{bmatrix}
\]

If \((x^*, y^*, w^*, s^*)\) is a solution of the previously defined system of equations that represent the KKT conditions, then \( x^* \) is also an optimal solution to the original QP model.

At each iteration the interior point algorithm solves a large, sparse system of linear equations as follows:

\[
\begin{bmatrix}
Y^{-1} S & A \\
A^T & -Q - X^{-1} W
\end{bmatrix}
\begin{bmatrix}
\Delta y \\
\Delta x
\end{bmatrix} =
\begin{bmatrix}
\Xi \\
\Theta
\end{bmatrix}
\]

where \( \Delta x \) and \( \Delta y \) denote the vector of search directions in the primal and dual spaces, respectively, and \( \Theta \) and \( \Xi \) constitute the vector of the right-hand sides.

The preceding system is known as the reduced KKT system. PROC OPTQP uses a preconditioned quasi-minimum residual algorithm to solve this system of equations efficiently.

An important feature of the interior point solver is that it takes full advantage of the sparsity in the constraint and quadratic matrices, thereby enabling it to efficiently solve large-scale quadratic programs.

The interior point algorithm works simultaneously in the primal and dual spaces. It attains optimality when both primal and dual feasibility are achieved and when complementarity conditions hold. Therefore, it is of interest to observe the following four measures where \( \|v\|_2 \) is the Euclidean norm of the vector \( v \):

- relative primal infeasibility measure \( \alpha \):

\[
\alpha = \frac{\|Ax - b - s\|_2}{\|b\|_2 + 1}
\]
• relative dual infeasibility measure $\beta$:
$$\beta = \frac{\|Qx + c - ATy - w\|_2}{\|e\|_2 + 1}$$

• relative duality gap $\delta$
$$\delta = \frac{|x^TQx + c^Tx - b^Ty|}{\frac{1}{2}x^TQx + c^Tx} + 1$$

• absolute complementarity $\gamma$
$$\gamma = \sum_{i=1}^{n} x_i w_i + \sum_{i=1}^{m} y_i s_i$$

These measures are displayed in the iteration log.

---

**Iteration Log for the OPTQP Procedure**

The interior point solver in PROC OPTQP implements an infeasible primal-dual predictor-corrector interior point algorithm. The following information is displayed in the iteration log:

- **Iter** indicates the iteration number.
- **Complement** indicates the (absolute) complementarity.
- **Duality Gap** indicates the (relative) duality gap.
- **Primal Infeas** indicates the (relative) primal infeasibility measure.
- **Bound Infeas** indicates the (relative) bound infeasibility measure.
- **Dual Infeas** indicates the (relative) dual infeasibility measure.
- **Time** indicates the time elapsed (in seconds).

If the sequence of solutions converges to an optimal solution of the problem, you should see all columns in the iteration log converge to zero or very close to zero. Nonconvergence can be the result of insufficient iterations being performed to reach optimality. In this case, you might need to increase the value that you specify in the **MAXITER=** or **MAXTIME=** option. If the complementarity or the duality gap does not converge, the problem might be infeasible or unbounded. If the infeasibility columns do not converge, the problem might be infeasible.
PROC OPTQP creates two Output Delivery System (ODS) tables by default: the ProblemSummary table is a summary of the input QP problem, and the SolutionSummary table is a brief summary of the solution status. If either or both of the PRIMALOUT= and DUALOUT= options are specified, then PROC OPTQP creates a third ODS table, OutputCasTables, which is the summary of output CAS table information. You can use ODS table names to select tables and create output data sets. For more information about ODS, see the SAS Output Delivery System: User’s Guide.

If you specify PRINTLEVEL=2, then two additional ODS tables are produced: The Timing table, which is a summary of the time spent on different PROC OPTQP tasks, and the ProblemStatistics table, which contains information about the problem data. For more information about the ProblemStatistics table, see the section “Problem Statistics” on page 243.

Table 8.2 lists all the ODS tables that can be produced by the OPTQP procedure, along with the statement and option specifications required to produce each table.

<table>
<thead>
<tr>
<th>ODS Table Name</th>
<th>Description</th>
<th>Statement</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProblemSummary</td>
<td>Summary of the input QP problem</td>
<td>PROC OPTQP</td>
<td>PRINTLEVEL=1 (default)</td>
</tr>
<tr>
<td>SolutionSummary</td>
<td>Summary of the solution status</td>
<td>PROC OPTQP</td>
<td>PRINTLEVEL=1 (default)</td>
</tr>
<tr>
<td>OutputCasTables</td>
<td>Summary of output CAS tables</td>
<td>PROC OPTQP</td>
<td>PRIMALOUT= or DUALOUT=</td>
</tr>
<tr>
<td>ProblemStatistics</td>
<td>Description of input problem data</td>
<td>PROC OPTQP</td>
<td>PRINTLEVEL=2</td>
</tr>
<tr>
<td>Timing</td>
<td>Summary of time consumption</td>
<td>PROC OPTQP</td>
<td>PRINTLEVEL=2</td>
</tr>
</tbody>
</table>

A typical output of PROC OPTQP is shown in Output 8.5.
**Figure 8.5** Typical OPTQP Output

### The OPTQP Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Name</td>
</tr>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>RHS</td>
</tr>
<tr>
<td><strong>Number of Variables</strong></td>
</tr>
<tr>
<td>Bounded Above</td>
</tr>
<tr>
<td>Bounded Below</td>
</tr>
<tr>
<td>Bounded Above and Below</td>
</tr>
<tr>
<td>Free</td>
</tr>
<tr>
<td>Fixed</td>
</tr>
</tbody>
</table>

| **Number of Constraints**        | 305                 |
| LE (<=)                          | 0                   |
| EQ (==)                          | 305                 |
| GE (>=)                          | 0                   |
| Range                            | 0                   |

| Constraint Coefficients          | 2494                |
| Hessian Diagonal Elements        | 25                  |
| Hessian Elements Below Diagonal  | 16                  |

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Primal Infeasibility</td>
</tr>
<tr>
<td>Dual Infeasibility</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
</tr>
<tr>
<td>Duality Gap</td>
</tr>
<tr>
<td>Complementarity</td>
</tr>
</tbody>
</table>

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output CAS Tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAS Library</td>
</tr>
<tr>
<td>CASUSERHDFS(amnrai) P</td>
</tr>
<tr>
<td>CASUSERHDFS(amnrai) D</td>
</tr>
</tbody>
</table>
You can create output data sets from these tables by using the ODS OUTPUT statement. This can be useful, for example, when you want to create a report to summarize multiple PROC OPTQP runs. The output data sets that correspond to the preceding output are shown in Output 8.6, where you can also find (in the row following the heading of each data set in the display) the variable names that are used in the table definition (template) of each table.

**Figure 8.6  ODS Output Data Sets**

**Problem Summary**

<table>
<thead>
<tr>
<th>Obs</th>
<th>Name1</th>
<th>Label1</th>
<th>cValue1</th>
<th>nValue1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>problemName</td>
<td>Problem Name</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>objectiveSense</td>
<td>Objective Sense</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>objectiveName</td>
<td>Objective Function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>rhs</td>
<td>RHS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>numVars</td>
<td>Number of Variables</td>
<td>472</td>
<td>472.000000</td>
</tr>
<tr>
<td>7</td>
<td>numBoundedAbove</td>
<td>Bounded Above</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>numBoundedBelow</td>
<td>Bounded Below</td>
<td>472</td>
<td>472.000000</td>
</tr>
<tr>
<td>9</td>
<td>numBounded</td>
<td>Bounded Above and Below</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>numFree</td>
<td>Free</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>numFixed</td>
<td>Fixed</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>numRows</td>
<td>Number of Constraints</td>
<td>305</td>
<td>305.000000</td>
</tr>
<tr>
<td>14</td>
<td>numLE</td>
<td>LE (&lt;=)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>numEQ</td>
<td>EQ (=)</td>
<td>305</td>
<td>305.000000</td>
</tr>
<tr>
<td>16</td>
<td>numGE</td>
<td>GE (&gt;=)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>numRange</td>
<td>Range</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>numElems</td>
<td>Constraint Coefficients</td>
<td>2494</td>
<td>2494.000000</td>
</tr>
<tr>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>numQDiagonal</td>
<td>Hessian Diagonal Elements</td>
<td>25</td>
<td>25.000000</td>
</tr>
<tr>
<td>22</td>
<td>numQAbove</td>
<td>Hessian Elements Below Diagonal 16</td>
<td>16</td>
<td>16.000000</td>
</tr>
</tbody>
</table>
Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of $10^9$) can cause difficulty when the remaining entries are single-digit numbers. The `PRINTLEVEL=2` option in the OPTQP procedure causes the ODS table `ProblemStatistics` to be generated. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Output 8.7 demonstrates the contents of the ODS table `ProblemStatistics`.
Irreducible Infeasible Set

For a quadratic programming problem, an irreducible infeasible set (IIS) is an infeasible subset of constraints and variable bounds that becomes feasible if any single constraint or variable bound is removed. It is possible to have more than one IIS in an infeasible QP. Identifying an IIS can help isolate the structural infeasibility in a QP. The IIS=TRUE option directs the OPTQP procedure to search for an IIS in a specified QP.

Whether a quadratic programming problem is feasible or infeasible is determined by its constraints and variable bounds, which have nothing to do with its objective function. When you specify the IIS=TRUE option, the OPTQP procedure treats this problem as a linear programming problem by ignoring its objective function. Then finding IIS is the same as what PROC OPTLP does with the IIS=TRUE option. For more information about the irreducible infeasible set, see the section “Irreducible Infeasible Set” on page 133 in Chapter 6, “The OPTLP Procedure.”
The OPTQP procedure defines a macro variable named _OROPTQP_. This variable contains a character string that indicates the status of the OPTQP procedure upon termination. The various terms of the variable are interpreted as follows.

**STATUS**
indicates the solver status at termination. It can take one of the following values:

- **OK** The procedure terminated normally.
- **SYNTAX_ERROR** Incorrect syntax was used.
- **DATA_ERROR** The input data were inconsistent.
- **OUT_OF_MEMORY** Insufficient memory was allocated to the procedure.
- **IO_ERROR** A problem occurred in reading or writing data.
- **ERROR** The status cannot be classified into any of the preceding categories.

**ALGORITHM**
indicates the algorithm that produced the solution data in the macro variable. This term appears only when **STATUS=OK**. It can take the following value:

- **IP** The interior point algorithm produced the solution data.
- **IIS** The IIS functionality produced the solution data.

**SOLUTION_STATUS**
indicates the solution status at termination. It can take one of the following values:

- **OPTIMAL** The solution is optimal.
- **CONDITIONAL_OPTIMAL** The solution is optimal, but some infeasibilities (primal, dual or bound) exceed tolerances due to scaling or preprocessing.
- **FEASIBLE** The problem is feasible (IIS).
- **INFEASIBLE** The problem is infeasible.
- **UNBOUNDED** The problem is unbounded.
- **INFEASIBLE_OR_UNBOUNDED** The problem is infeasible or unbounded.
- **ITERATION_LIMIT_REACHED** The maximum allowable number of iterations was reached.
- **TIME_LIMIT_REACHED** The maximum time limit was reached.
- **ABORTED** The solver was interrupted externally.
- **FAILED** The solver failed to converge, possibly due to numerical issues.
- **NONCONVEX** The quadratic matrix is nonconvex (minimization).
- **NONCONCAVE** The quadratic matrix is nonconcave (maximization).
OBJECTIVE
indicates the objective value obtained by the solver at termination.

PRIMAL_INFEASIBILITY
indicates the (relative) infeasibility of the primal constraints at the solution. For more information, see the section “Interior Point Algorithm: Overview” on page 237.

DUAL_INFEASIBILITY
indicates the (relative) infeasibility of the dual constraints at the solution. For more information, see the section “Interior Point Algorithm: Overview” on page 237.

BOUND_INFEASIBILITY
indicates the (relative) violation by the solution of the lower or upper bounds (or both). For more information, see the section “Interior Point Algorithm: Overview” on page 237.

DUALITY_GAP
indicates the (relative) duality gap. For more information, see the section “Interior Point Algorithm: Overview” on page 237.

COMPLEMENTARITY
indicates the (absolute) complementarity at the solution. For more information, see the section “Interior Point Algorithm: Overview” on page 237.

ITERATIONS
indicates the number of iterations taken to solve the problem.

PRESOLVE_TIME
indicates the time (in seconds) taken for preprocessing.

SOLUTION_TIME
indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time that is reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

Examples: OPTQP Procedure

This section contains examples that illustrate the use of the OPTQP procedure. Example 8.1 illustrates how to model a linear least squares problem and solve it by using PROC OPTQP. Example 8.2 and Example 8.3 explain in detail how to model the portfolio optimization and selection problems.

Example 8.1: Linear Least Squares Problem

The linear least squares problem arises in the context of determining a solution to an overdetermined set of linear equations. In practice, these equations could arise in data fitting and estimation problems. An overdetermined system of linear equations can be defined as

\[ Ax = b \]
where $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, $b \in \mathbb{R}^m$, and $m > n$. Since this system usually does not have a solution, you need to be satisfied with some sort of approximate solution. The most widely used approximation is the least squares solution, which minimizes $\|Ax - b\|_2^2$.

This problem is called a least squares problem for the following reason. Let $A$, $x$, and $b$ be defined as previously. Let $k_i(x)$ be the $k$th component of the vector $Ax - b$:

$$k_i(x) = a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n - b_i, \; i = 1, 2, \ldots, m$$

By definition of the Euclidean norm, the objective function can be expressed as follows:

$$\|Ax - b\|_2^2 = \sum_{i=1}^{m} k_i(x)^2$$

Therefore, the function you minimize is the sum of squares of $m$ terms $k_i(x)$; hence the term least squares. The following example is an illustration of the linear least squares problem; that is, each of the terms $k_i$ is a linear function of $x$. Each term $\sum_{ij} a_{ij}x_j$ plus a constant, $-b_i$.

Consider the following least squares problem defined by

$$A = \begin{bmatrix} 4 & 0 \\ -1 & 1 \\ 3 & 2 \end{bmatrix}, \quad b = \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

This translates to the following set of linear equations:

$$4x_1 = 1, \quad -x_1 + x_2 = 0, \quad 3x_1 + 2x_2 = 1$$

The corresponding least squares problem is

$$\text{minimize} \quad (4x_1 - 1)^2 + (-x_1 + x_2)^2 + (3x_1 + 2x_2 - 1)^2$$

The preceding objective function can be expanded to

$$\text{minimize} \quad 26x_1^2 + 5x_2^2 + 10x_1x_2 - 14x_1 - 4x_2 + 2$$

In addition, you impose the following constraint so that the equation $3x_1 + 2x_2 = 1$ is satisfied within a tolerance of 0.1:

$$0.9 \leq 3x_1 + 2x_2 \leq 1.1$$

You can create the QPS-format input data set by using the following SAS statements:

```sas
data mycas.lsdata;
  input _id_ field1 $ field2 $ field3 $ field4 field5 $ field6 @;
  datalines;
1 NAME . LEASTSQ . . . .
2 ROWS . . . . .
3 N OBJ . . . .
4 G EQ3 . . . .
5 COLUMNS . . . . .
6 . X1 OBJ -14 EQ3 3
7 . X2 OBJ -4 EQ3 2
```
The decision variables $x_1$ and $x_2$ are free, so they have bound type FR in the BOUNDS section of the QPS-format data set.

You can use the following SAS statements to solve the least squares problem:

```sas
proc optqp data = mycas.lsdata
  printlevel = 0
  primalout = mycas.lspout;
run;
```

The optimal solution is displayed in **Output 8.1.1**.

**Output 8.1.1** Solution to the Least Squares Problem

### Primal Solution

<table>
<thead>
<tr>
<th>Obs</th>
<th>Objective Function ID</th>
<th>RHS ID</th>
<th>Variable Name</th>
<th>Variable Type</th>
<th>Linear Objective Coefficient</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
<th>Variable Value</th>
<th>Variable Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OBJ</td>
<td>RHS</td>
<td>X1</td>
<td>F</td>
<td>-14</td>
<td>-1.7977E308</td>
<td>1.7977E308</td>
<td>0.23810</td>
<td>O</td>
</tr>
<tr>
<td>2</td>
<td>OBJ</td>
<td>RHS</td>
<td>X2</td>
<td>F</td>
<td>-4</td>
<td>-1.7977E308</td>
<td>1.7977E308</td>
<td>0.16190</td>
<td>O</td>
</tr>
</tbody>
</table>

The iteration log is shown in **Output 8.1.2**.
Example 8.2: Portfolio Optimization

Consider a portfolio optimization example. The two competing goals of investment are (1) long-term growth of capital and (2) low risk. A good portfolio grows steadily without wild fluctuations in value. The Markowitz model is an optimization model for balancing the return and risk of a portfolio. The decision variables are the amounts invested in each asset. The objective is to minimize the variance of the portfolio’s total return, subject to the constraints that (1) the expected growth of the portfolio reaches at least some target level and (2) you do not invest more capital than you have.

Let $x_1, \ldots, x_n$ be the amount invested in each asset, $B$ be the amount of capital you have, $R$ be the random vector of asset returns over some period, and $r$ be the expected value of $R$. Let $G$ be the minimum growth you hope to obtain, and $C$ be the covariance matrix of $R$. The objective function is $\text{Var} \left( \sum_{i=1}^{n} x_i R_i \right)$, which can be equivalently denoted as $x^T C x$. 

---

Output 8.1.2 Iteration Log

NOTE: The problem LEASTSQ has 2 variables (2 free, 0 fixed).
NOTE: The problem has 1 constraints (0 LE, 0 EQ, 0 GE, 1 range).
NOTE: The problem has 2 constraint coefficients.
NOTE: The objective function has 2 Hessian diagonal elements and 1 Hessian elements above the diagonal.
NOTE: The QP presolver value AUTOMATIC is applied.
NOTE: The QP presolver removed 0 variables and 0 constraints.
NOTE: The QP presolver removed 0 constraint coefficients.
NOTE: The presolved problem has 2 variables, 1 constraints, and 2 constraint coefficients.
NOTE: The QP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Interior Point algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Iter</th>
<th>Primal</th>
<th>Bound</th>
<th>Dual</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Complement</td>
<td>Duality Gap</td>
<td>Infeas</td>
</tr>
<tr>
<td>0</td>
<td>1.918E-02</td>
<td>5.893E-03</td>
<td>1.963E-08</td>
</tr>
<tr>
<td>1</td>
<td>9.047E-04</td>
<td>2.831E-04</td>
<td>8.652E-10</td>
</tr>
<tr>
<td>2</td>
<td>1.552E-05</td>
<td>4.996E-06</td>
<td>6.410E-11</td>
</tr>
<tr>
<td>3</td>
<td>0.0000E+00</td>
<td>7.665E-07</td>
<td>1.744E-12</td>
</tr>
</tbody>
</table>

NOTE: Optimal.
NOTE: Objective = 0.0095238095.
NOTE: The Interior Point solve time is 0.00 seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.031862 seconds.
NOTE: The data set MYCAS.LSPOUT has 2 observations and 9 variables.
Assume, for example, \( n = 4 \). Let \( B = 10,000 \), \( G = 1000 \), \( r = [0.05, -0.2, 0.15, 0.30] \), and

\[
C = \begin{bmatrix}
0.08 & -0.05 & -0.05 & -0.05 \\
-0.05 & 0.16 & -0.02 & -0.02 \\
-0.05 & -0.02 & 0.35 & 0.06 \\
-0.05 & -0.02 & 0.06 & 0.35
\end{bmatrix}
\]

The QP formulation can be written as follows:

\[
\begin{align*}
\text{min} & \quad 0.08x_1^2 - 0.1x_1x_2 - 0.1x_1x_3 - 0.1x_1x_4 + 0.16x_2^2 \\
& - 0.04x_2x_3 - 0.04x_2x_4 + 0.35x_3^2 + 0.12x_3x_4 + 0.35x_4^2 \\
\text{subject to} & \\
\text{(budget)} & \quad x_1 + x_2 + x_3 + x_4 \leq 10000 \\
\text{(growth)} & \quad 0.05x_1 - 0.2x_2 + 0.15x_3 + 0.30x_4 \geq 1000 \\
& \quad x_1, x_2, x_3, x_4 \geq 0
\end{align*}
\]

The corresponding QPS-format input data set is as follows:

```plaintext
data portdata;
  input field1 $ field2 $ field3 $ field4 field5 $ field6 @;
datalines;
NAME . PORT . . . .
ROWS . . . . . .
N OBJ.FUNC . . . .
L BUDGET . . . .
G GROWTH . . . .
COLUMNS . . . . .
. X1 BUDGET 1.0 GROWTH 0.05
. X2 BUDGET 1.0 GROWTH -.20
. X3 BUDGET 1.0 GROWTH 0.15
. X4 BUDGET 1.0 GROWTH 0.30
RHS . . . . . .
. RHS BUDGET 10000 . .
. RHS GROWTH 1000 . .
RANGES . . . . . .
BOUNDS . . . . . .
QUADOBJ . . . . . .
. X1 X1 0.16 . .
. X1 X2 -.10 . .
. X1 X3 -.10 . .
. X1 X4 -.10 . .
. X2 X2 0.32 . .
. X2 X3 -.04 . .
. X2 X4 -.04 . .
. X3 X3 0.70 . .
. X3 X4 0.12 . .
. X4 X4 0.70 . .
ENDATA . . . . . .
;
```
Example 8.2: Portfolio Optimization

Use the following SAS statements to solve the problem:

```sas
data mycas.portdata;
  set portdata;
  _id_ = _n_;
run;

proc optqp data = mycas.portdata
  printlevel = 0
  primalout = mycas.portpout
  dualout = mycas.portdout;
run;
```

The optimal solution is shown in Output 8.2.1.

**Output 8.2.1** Portfolio Optimization

The OPTQP Procedure

Primal Solution

<table>
<thead>
<tr>
<th>Obs</th>
<th>Objective Function ID</th>
<th>RHS ID</th>
<th>Variable Name</th>
<th>Variable Type</th>
<th>Linear Objective Coefficient</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
<th>Variable Value</th>
<th>Variable Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OBJ.FUNC RHS X1</td>
<td>N</td>
<td>X1</td>
<td>N</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.7977E308</td>
<td>3452.86</td>
</tr>
<tr>
<td>2</td>
<td>OBJ.FUNC RHS X2</td>
<td>N</td>
<td>X2</td>
<td>N</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.7977E308</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td>OBJ.FUNC RHS X3</td>
<td>N</td>
<td>X3</td>
<td>N</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.7977E308</td>
<td>1068.81</td>
</tr>
<tr>
<td>4</td>
<td>OBJ.FUNC RHS X4</td>
<td>N</td>
<td>X4</td>
<td>N</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1.7977E308</td>
<td>2223.45</td>
</tr>
</tbody>
</table>

Thus, the minimum variance portfolio that earns an expected return of at least 10% is $x_1 = 3452.86$, $x_2 = 0$, $x_3 = 1068.81$, $x_4 = 2223.45$. Asset 2 gets nothing, because its expected return is $-20\%$ and its covariance with the other assets is not sufficiently negative for it to bring any diversification benefits. What if you drop the nonnegativity assumption? You need to update the BOUNDS section in the existing QPS-format data set to indicate that the decision variables are free.

```
... RANGES . . . . . . . . .
BOUNDS . . . . . . . . .
FR BND1 X1 . . . . . . .
FR BND1 X2 . . . . . . .
FR BND1 X3 . . . . . . .
FR BND1 X4 . . . . . . .
QUADOBJ . . . . . . . . .
...```

Financially, that means you are allowed to short-sell—that is, sell low-mean-return assets and use the proceeds to invest in high-mean-return assets. In other words, you put a negative portfolio weight in low-mean assets and “more than 100%” in high-mean assets. You can see in the optimal solution displayed in Output 8.2.2 that the decision variable $x_2$, denoting Asset 2, is equal to $-1563.61$, which means short sale of that asset.
Example 8.3: Portfolio Selection with Transactions

Consider a portfolio selection problem with a slight modification. You are now required to take into account the current position and transaction costs associated with buying and selling assets. The objective is to find the minimum variance portfolio. In order to understand the scenario better, consider the following data.

You are given three assets. The current holding of the three assets is denoted by the vector \( c = [200, 300, 500] \), the amount of asset bought and sold is denoted by \( b_i \) and \( s_i \), respectively, and the net investment in each asset is denoted by \( x_i \) and is defined by the following relation:

\[
x_i - b_i + s_i = c_i, \quad i = 1, 2, 3
\]

Suppose you pay a transaction fee of 0.01 every time you buy or sell. Let the covariance matrix \( C \) be defined as

\[
C = \begin{bmatrix}
0.027489 & -0.00874 & -0.00015 \\
-0.00874 & 0.109449 & -0.00012 \\
-0.00015 & -0.00012 & 0.000766
\end{bmatrix}
\]

Assume that you hope to obtain at least 12% growth. Let \( r = [1.109048, 1.169048, 1.074286] \) be the vector of expected return on the three assets, and let \( B=1000 \) be the available funds. Mathematically, this problem can be written in the following manner:

\[
\begin{align*}
\min & \quad 0.027489x_1^2 - 0.01748x_1x_2 - 0.0003x_1x_3 + 0.109449x_2^2 \\
& \quad -0.00024x_2x_3 + 0.000766x_3^2 \\
\text{(return)} & \quad \sum_{i=1}^{3} r_i x_i \geq 1.12B \\
\text{(budget)} & \quad \sum_{i=1}^{3} x_i + \sum_{i=1}^{3} 0.01(b_i + s_i) = B \\
\text{(balance)} & \quad x_i - b_i + s_i = c_i, \quad i = 1, 2, 3 \\
& \quad x_i, b_i, s_i \geq 0, \quad i = 1, 2, 3
\end{align*}
\]

The QPS-format input data set is as follows:
data potrdata;
    input field1 $ field2 $ field3 $ field4 field5 $ field6 @;
datalines;
NAME . POTRAN . . . .
ROWS . . . . . . .
N OBJ.FUNC . . . .
G RETURN . . . .
E BUDGET . . . .
E BALANC1 . . . .
E BALANC2 . . . .
E BALANC3 . . . .
COLUMNS . . . . . .
   X1 RETURN 1.109048 BUDGET 1.0
   X1 BALANC1 1.0 . .
   X2 RETURN 1.169048 BUDGET 1.0
   X2 BALANC2 1.0 . .
   X3 RETURN 1.074286 BUDGET 1.0
   X3 BALANC3 1.0 . .
   B1 BUDGET .01 BALANC1 -1.0
   B2 BUDGET .01 BALANC2 -1.0
   B3 BUDGET .01 BALANC3 -1.0
   S1 BUDGET .01 BALANC1 1.0
   S2 BUDGET .01 BALANC2 1.0
   S3 BUDGET .01 BALANC3 1.0
RHS . . . . . .
   RHS RETURN 1120 . .
   RHS BUDGET 1000 . .
   RHS BALANC1 200 . .
   RHS BALANC2 300 . .
   RHS BALANC3 500 . .
RANGES . . . . . .
BOUNDS . . . . . .
QUADOBJ . . . . .
   X1 X1 0.054978 . .
   X1 X2 -.01748 . .
   X1 X3 -.0003 . .
   X2 X2 0.218898 . .
   X2 X3 -.00024 . .
   X3 X3 0.001532 . .
ENDATA . . . . . .
;

Use the following SAS statements to solve the problem:

data mycas.potrdata;
    set potrdata;
    _id_ = _n_; run;

proc optqp data = mycas.potrdata
    printlevel = 0
    primalout = mycas.potrpout
dualout = mycas.potrdout;
run;
The optimal solution is displayed in Output 8.3.1.

Output 8.3.1 Portfolio Selection with Transactions

The OPTQP Procedure
Primal Solution

<table>
<thead>
<tr>
<th>Obs</th>
<th>Objective Function ID</th>
<th>RHS ID</th>
<th>Variable Name</th>
<th>Variable Type</th>
<th>Linear Objective Coefficient</th>
<th>Lower Bound</th>
<th>Upper Bound</th>
<th>Variable Value</th>
<th>Variable Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OBJ.FUNC RHS X1</td>
<td>N</td>
<td></td>
<td>0</td>
<td>0.7977E308</td>
<td>397.584 O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>OBJ.FUNC RHS X2</td>
<td>N</td>
<td></td>
<td>0</td>
<td>0.7977E308</td>
<td>406.115 O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>OBJ.FUNC RHS X3</td>
<td>N</td>
<td></td>
<td>0</td>
<td>0.7977E308</td>
<td>190.165 O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>OBJ.FUNC RHS B1</td>
<td>N</td>
<td></td>
<td>0</td>
<td>0.7977E308</td>
<td>197.584 O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>OBJ.FUNC RHS B2</td>
<td>N</td>
<td></td>
<td>0</td>
<td>0.7977E308</td>
<td>106.115 O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>OBJ.FUNC RHS B3</td>
<td>N</td>
<td></td>
<td>0</td>
<td>0.7977E308</td>
<td>0.000 O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>OBJ.FUNC RHS S1</td>
<td>N</td>
<td></td>
<td>0</td>
<td>0.7977E308</td>
<td>0.000 O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>OBJ.FUNC RHS S2</td>
<td>N</td>
<td></td>
<td>0</td>
<td>0.7977E308</td>
<td>0.000 O</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>OBJ.FUNC RHS S3</td>
<td>N</td>
<td></td>
<td>0</td>
<td>0.7977E308</td>
<td>309.835 O</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

References


Chapter 9
The MPS-Format Data Table

Contents

Overview: MPS-Format Data Table ................................................. 255
Observations ................................................................. 256
Order of Sections .......................................................... 256
Sections Format: MPS-Format Data Table ................................. 257
NAME Section .............................................................. 257
ROWS Section .............................................................. 257
COLUMNS Section .......................................................... 258
RHS Section (Optional) ...................................................... 259
RANGES Section (Optional) ................................................. 260
BOUNDS Section (Optional) ................................................. 261
BRANCH Section (Optional) ............................................... 262
QSECTION Section (Optional) .............................................. 263
ENDATA Section ............................................................ 264
Details: MPS-Format Data Table ................................................. 264
Converting an MPS/QPS-Format File: %MPS2SASD .......................... 264
Length of Variables .......................................................... 265
Examples: MPS-Format Data Table ........................................... 266
Example 9.1: MPS-Format Data Table for a Product Mix Problem ........ 266
Example 9.2: Fixed-MPS-Format File ....................................... 267
Example 9.3: Free-MPS-Format File ....................................... 267
Example 9.4: Using the %MPS2SASD Macro ............................... 268
References ........................................................................... 270

Overview: MPS-Format Data Table

The MPS file format is a format commonly used in industry to describe linear programming (LP) and mixed integer linear programming (MILP) problems (Murtagh 1981; IBM 1988). It can be extended to the QPS format (Maros and Mészáros 1999), which describes quadratic programming (QP) problems. MPS-format and QPS-format files are in text format and have specific conventions for the order in which the different pieces of the mathematical model are specified. The MPS-format data table corresponds closely to the format of an MPS-format or QPS-format file and is used to describe linear programming, mixed integer linear programming, and quadratic programming problems for SAS.
Observations

An MPS-format data table contains seven variables: _ID_, field1, field2, field3, field4, field5, and field6. The variables _ID_, field4, and field6 are numeric type; the others are character type. Among the character variables, only the value of field1 is case-insensitive and leading blanks are ignored. Values of field2, field3, and field5 are case-sensitive and leading blanks are NOT ignored. Not all variables are used in a particular observation.

Observations in an MPS-format data table are grouped into sections. Each section starts with an indicator record, followed by associated data records. Indicator records specify the names of sections and the format of the following data records. Data records contain the actual data values for a section.

The _ID_ variable maintains the order of the sections in the MPS-format data table on the CAS server. Hence it has to be an increasing index number, indicating an order of the observations that results in a valid MPS-format data table. If the _ID_ variable is omitted, you might get an error message that indicates that sections are not read in the expected order.

Order of Sections

Sections of an MPS-format data table must be specified in a fixed order, as indicated by the _ID_ variable.

Sections of linear programming problems are listed in the following order:

- NAME
- ROWS
- COLUMNS
- RHS (optional)
- RANGES (optional)
- BOUNDS (optional)
- ENDATA

Sections of quadratic programming problems are listed in the following order:

- NAME
- ROWS
- COLUMNS
- RHS (optional)
- RANGES (optional)
- BOUNDS (optional)
Sections Format: MPS-Format Data Table

The following subsections describe the format of the records for each section of the MPS-format data table. Note that each section contains two types of records: an indicator record and multiple data records. The following subsections of this documentation describe the two different types of records for each section of the MPS data table.

NAME Section

The NAME section contains only a single record identifying the name of the problem.

<table>
<thead>
<tr>
<th>Field 1</th>
<th>Field2</th>
<th>Field3</th>
<th>Field4</th>
<th>Field5</th>
<th>Field6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>NAME</td>
<td>Blank</td>
<td>Input</td>
<td>.</td>
<td>Blank or FREE or FIXED</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>model</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>name</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ROWS Section

The ROWS section contains the name and type of the rows (linear constraints or objectives). The type of each row is specified by the indicator code in field1 as follows:

- **MIN**: minimization objective
- **MAX**: maximization objective
Chapter 9: The MPS-Format Data Table

- **N**: objective
- **G**: ≥ constraint
- **L**: ≤ constraint
- **E**: = constraint

**Indicator record:**

<table>
<thead>
<tr>
<th><em>ID</em></th>
<th>Field1</th>
<th>Field2</th>
<th>Field3</th>
<th>Field4</th>
<th>Field5</th>
<th>Field6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>ROWS</td>
<td>Blank</td>
<td>Blank</td>
<td>.</td>
<td>Blank</td>
<td>.</td>
</tr>
</tbody>
</table>

**Data record:**

<table>
<thead>
<tr>
<th><em>ID</em></th>
<th>Field1</th>
<th>Field2</th>
<th>Field3</th>
<th>Field4</th>
<th>Field5</th>
<th>Field6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>Indicator code</td>
<td>Row name</td>
<td>Blank</td>
<td>.</td>
<td>Blank</td>
<td>.</td>
</tr>
</tbody>
</table>

**Notes:**

1. At least one objective row should be specified in the ROWS section. It is possible to specify multiple objective rows. However, among all the data records indicating the objective, only the first one is regarded as the objective row, while the rest are ignored. If a type-N row is taken as the objective row, minimization is assumed.

2. Duplicate entries of field2 in the ROWS section are not allowed. In other words, row name is unique. The variable field2 in the ROWS section cannot take a missing value.

---

**COLUMNS Section**

The COLUMNS section defines the column (that is, variable or decision variable) names of the problem. It also specifies the coefficients of the columns for each row.

**Indicator record:**

<table>
<thead>
<tr>
<th><em>ID</em></th>
<th>Field1</th>
<th>Field2</th>
<th>Field3</th>
<th>Field4</th>
<th>Field5</th>
<th>Field6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>COLUMNS</td>
<td>Blank</td>
<td>Blank</td>
<td>.</td>
<td>Blank</td>
<td>.</td>
</tr>
</tbody>
</table>

**Data record:**

<table>
<thead>
<tr>
<th><em>ID</em></th>
<th>Field1</th>
<th>Field2</th>
<th>Field3</th>
<th>Field4</th>
<th>Field5</th>
<th>Field6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>Blank</td>
<td>Column name (e.g., col)</td>
<td>Row name (e.g., rowi)</td>
<td>Matrix element in row rowi, column col</td>
<td>Row name (e.g., rowj)</td>
<td>Matrix element in row rowj, column col</td>
</tr>
</tbody>
</table>
Notes:

1. All elements belonging to one column must be grouped together.

2. A missing coefficient value is ignored. A data record with missing values in both field4 and field6 is ignored.

3. Duplicate entries in each pair of column and row are not allowed.

4. When a sequence of data records have an identical value in field2, you can specify the value in the first occurrence and omit the value by giving a missing value in the other records. The value in field2 of the first data record in the section cannot be missing.

Mixed Integer Linear Programs

Mixed integer linear programming (MILP) problems require you to specify which variables are constrained to be integers. Integer variables can be specified in the COLUMNS section with the use of special marker records in the following form:

```
_ID_ Field1 Field2 Field3 Field4 Field5 Field6
Number Blank Marker name ‘MARKER’ . ‘INTORG’ .
```

A marker record with field5 that contains the value ‘INTORG’ indicates the start of integer variables. In the marker record that indicates the end of integer variables, field5 must be ‘INTEND’. An alternative way to specify integer variables without using the marker records is described in the section “BOUNDS Section (Optional)” on page 261.

Notes:

1. INTORG and INTEND markers must appear in pairs in the COLUMNS section. The marker pairs can appear any number of times.

2. The marker name in field2 should be different from the preceding and following column names.

3. All variables between the INTORG and INTEND markers are assumed to be binary unless you specify a different lower bound and/or upper bound in the BOUNDS section.

RHS Section (Optional)

The RHS section specifies the right-hand-side value for the rows. Any row unspecified in this section is considered to have an RHS value of 0. Missing the entire RHS section implies that all RHS values are 0.

- Indicator record:

```
_ID_ Field1 Field2 Field3 Field4 Field5 Field6
Number RHS Blank Blank . Blank .
```
Data record:

<table>
<thead>
<tr>
<th>Field</th>
<th>Field</th>
<th>Field</th>
<th>Field</th>
<th>Field</th>
<th>Field</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID_</td>
<td>Field1</td>
<td>Field2</td>
<td>Field3</td>
<td>Field4</td>
<td>Field5</td>
</tr>
<tr>
<td>Number</td>
<td>Blank</td>
<td>RHS</td>
<td>Row</td>
<td>RHS</td>
<td>Row</td>
<td>RHS</td>
</tr>
<tr>
<td></td>
<td></td>
<td>name</td>
<td>name</td>
<td>value</td>
<td>name</td>
<td>value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(e.g.,</td>
<td>rowi)</td>
<td>for</td>
<td>(e.g.,</td>
<td>for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rowi)</td>
<td></td>
<td>rowi</td>
<td>rowj)</td>
<td>rowj</td>
</tr>
</tbody>
</table>

Notes:

1. The rows that have an RHS element defined in this section need not be specified in the same order in which the rows were specified in the ROWS section. However, a row in the RHS section should be defined in the ROWS section.

2. It is possible to specify multiple RHS vectors, which are labeled by different RHS names. The first RHS vector encountered in the RHS section is used, and all other RHS vectors are discarded. All the elements of the selected RHS vector must be specified before other RHS vectors are introduced. Within a specific RHS vector, for a given row, duplicate assignments of RHS values are not allowed.

3. An RHS value assigned to the objective row is ignored by PROC OPTLP and PROC OPTMILP, while it is taken as a constant term of the objective function by PROC OPTQP.

4. A missing value in field4 or field6 is ignored. A data record with missing values in both field4 and field6 is ignored.

5. When a sequence of data records have an identical value in field2, you can specify the value in the first occurrence and omit the value by giving a missing value in the other records. If the value in field2 of the first data record in the section is missing, it means the name of the first vector is the missing value.

RANGES Section (Optional)

The RANGES section specifies the range of the RHS value for the constraint rows. With range specification, a row can be constrained from above and below.

For a constraint row \( c \), if \( b \) is the RHS value and \( r \) is the range for this row, then the equivalent constraints are given in Table 9.1, depending on the type of row and the sign of \( r \).

<table>
<thead>
<tr>
<th>Type of Row</th>
<th>Sign of ( r )</th>
<th>Equivalent Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>±</td>
<td>( b \leq c \leq b +</td>
</tr>
<tr>
<td>L</td>
<td>±</td>
<td>( b -</td>
</tr>
<tr>
<td>E</td>
<td>+</td>
<td>( b \leq c \leq b + r )</td>
</tr>
<tr>
<td>E</td>
<td>–</td>
<td>( b + r \leq c \leq b )</td>
</tr>
</tbody>
</table>

Indicator record:

<table>
<thead>
<tr>
<th>Field</th>
<th>Field</th>
<th>Field</th>
<th>Field</th>
<th>Field</th>
<th>Field</th>
<th>Field</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID_</td>
<td>Field1</td>
<td>Field2</td>
<td>Field3</td>
<td>Field4</td>
<td>Field5</td>
</tr>
<tr>
<td>Number</td>
<td>RANGES</td>
<td>Blank</td>
<td>Blank</td>
<td>Blank</td>
<td>Blank</td>
<td>Blank</td>
</tr>
</tbody>
</table>
• Data record:

<table>
<thead>
<tr>
<th><em>ID</em></th>
<th>Field1</th>
<th>Field2</th>
<th>Field3</th>
<th>Field4</th>
<th>Field5</th>
<th>Field6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>Blank</td>
<td>Range name</td>
<td>Row name</td>
<td>Range for RHS of row</td>
<td>Row name</td>
<td>Range for RHS of row</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(e.g., rowi)</td>
<td>(e.g., rowi)</td>
<td></td>
<td>(e.g., rowj)</td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. Range assignment for an objective row (that is, MAX, MIN, or N row) is not allowed.

2. The rows that have a range element defined in this section need not be specified in the same order in which the rows were specified in the ROWS or RHS section. However, a row in the RANGES section should be defined in the ROWS section.

3. It is possible to specify multiple range vectors, which are labeled by different range names. The first range vector encountered in the RANGES section is used, and all other range vectors are discarded. All the elements in a range vector must be specified before other range vectors are introduced. Within the specific range vector, for a given range, duplicate assignments of range values are not allowed.

4. A missing value in _field4_ or _field6_ is ignored. A data record with missing values in both _field4_ and _field6_ is ignored.

5. When a sequence of data records have an identical value in _field2_, you can specify the value in the first occurrence and omit the value by giving a missing value in the other records. If the value in _field2_ of the first data record in the section is missing, it means the name of the first vector is the missing value.

---

**BOUNDS Section (Optional)**

The BOUNDS section specifies bounds for the columns.

• Indicator record:

<table>
<thead>
<tr>
<th><em>ID</em></th>
<th>Field1</th>
<th>Field2</th>
<th>Field3</th>
<th>Field4</th>
<th>Field5</th>
<th>Field6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>BOUNDS</td>
<td>Blank</td>
<td>Blank</td>
<td>.</td>
<td>Blank</td>
<td>.</td>
</tr>
</tbody>
</table>

• Data record:

<table>
<thead>
<tr>
<th><em>ID</em></th>
<th>Field1</th>
<th>Field2</th>
<th>Field3</th>
<th>Field4</th>
<th>Field5</th>
<th>Field6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>Bound type</td>
<td>Bound name</td>
<td>Column name</td>
<td>Bound for the column</td>
<td>Blank</td>
<td>Blank</td>
</tr>
</tbody>
</table>

Notes:

1. If you do not specify any bound for a column, then the upper bound is $+\infty$ for a continuous variable, and 1 for an integer variable, that is specified in the COLUMNS section. The lower bound is 0 by default.

2. General bound types include LO, UP, FX, FR, MI, and PL. Suppose the bound for a column identified in _field3_ is specified as $b$ in _field4_. Table 9.2 explains the effects of different bound types.
Table 9.2  Bound Type Rules

<table>
<thead>
<tr>
<th>Bound Type</th>
<th>Ignore $b$</th>
<th>Resultant Lower Bound</th>
<th>Resultant Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>LO</td>
<td>No</td>
<td>$b$</td>
<td>Unspecified</td>
</tr>
<tr>
<td>UP</td>
<td>No</td>
<td>Unspecified</td>
<td>$b$</td>
</tr>
<tr>
<td>FX</td>
<td>No</td>
<td>$b$</td>
<td>$b$</td>
</tr>
<tr>
<td>FR</td>
<td>Yes</td>
<td>$-\infty$</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>MI</td>
<td>Yes</td>
<td>$-\infty$</td>
<td>Unspecified</td>
</tr>
<tr>
<td>PL</td>
<td>Yes</td>
<td>Unspecified</td>
<td>$+\infty$</td>
</tr>
</tbody>
</table>

If a bound (lower or upper) is not explicitly specified, then it takes the default values according to Note 1. There is one exception: if the upper bound is specified as a negative value ($b < 0$) and the lower bound is unspecified, then the lower bound is set to $-\infty$.

Mixed integer linear programming problems can specify integer variables in the BOUNDS section. Table 9.3 shows bound types defined for MILP.

Table 9.3  Bound Type Rules

<table>
<thead>
<tr>
<th>Bound Type</th>
<th>Ignore $b$</th>
<th>Variable Type</th>
<th>Resultant Lower Bound</th>
<th>Resultant Upper Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>BV</td>
<td>Yes</td>
<td>Binary</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>LI</td>
<td>No</td>
<td>Integer</td>
<td>$b$</td>
<td>$+\infty$</td>
</tr>
<tr>
<td>UI</td>
<td>No</td>
<td>Integer</td>
<td>Unspecified</td>
<td>$b$</td>
</tr>
</tbody>
</table>

3. The columns that have bounds do not need to be specified in the same order in which the columns were specified in the COLUMNS section. However, all columns in the BOUNDS section should be defined in the COLUMNS section.

4. It is possible to specify multiple bound vectors, which are labeled by different bound names. The first bound vector encountered in the BOUNDS section is used, and all other bound vectors are discarded. All the elements of the selected bound vector must be specified before other bound vectors are introduced.

5. When data records in a sequence have an identical value in field2, you can specify the value in the first occurrence and omit the value by giving a missing value in the other records. If the value in field2 of the first data record in the section is missing, it means the name of the first vector is the missing value.

6. Within a particular BOUNDS vector, for a given column, if a bound (lower or upper) is explicitly specified by the bound type rules listed in Table 9.2, any other specification is considered to be an error.

7. If the value in field1 is LO, UP, FX, LI, or UI, then a data record with a missing value in field4 is ignored.

BRANCH Section (Optional)

Sometimes you want to specify branching priorities or directions for integer variables to improve performance. Variables with higher priorities are branched on before variables with lower priorities. The branch direction
is used to decide which branch to take first at each node. For more information, see the section “Branching Priorities” on page 182.

- **Indicator record:**
  
<table>
<thead>
<tr>
<th><em>ID</em></th>
<th>Field1</th>
<th>Field2</th>
<th>Field3</th>
<th>Field4</th>
<th>Field5</th>
<th>Field6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>BRANCH</td>
<td>Blank</td>
<td>Blank</td>
<td>.</td>
<td>Blank</td>
<td>.</td>
</tr>
</tbody>
</table>

- **Data record:**
  
<table>
<thead>
<tr>
<th><em>ID</em></th>
<th>Field1</th>
<th>Field2</th>
<th>Field3</th>
<th>Field4</th>
<th>Field5</th>
<th>Field6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>Branch</td>
<td>Blank</td>
<td>First</td>
<td>First</td>
<td>Second</td>
<td>Second</td>
</tr>
<tr>
<td></td>
<td>direction</td>
<td></td>
<td>column</td>
<td>column</td>
<td>name</td>
<td>name</td>
</tr>
</tbody>
</table>

**Notes:**

1. Valid directions include **UP** (up branch), **DN** (down branch), **RD** (rounding) and **CB** (closest bound). If field1 is blank, the solver automatically decides the direction.

2. If field4 is missing, then the name defined in field3 is ignored. Similarly, if field6 is missing, then the name defined in field5 is ignored.

3. The priority value in field4 and field6 must be nonnegative. Zero is the lowest priority and is also the default.

---

## QSECTION Section (Optional)

The QSECTION section is needed only to describe quadratic programming problems. It specifies the coefficients of the quadratic terms in the objective function.

- **Indicator record:**
  
<table>
<thead>
<tr>
<th><em>ID</em></th>
<th>Field1</th>
<th>Field2</th>
<th>Field3</th>
<th>Field4</th>
<th>Field5</th>
<th>Field6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>QSECTION or QUADOBJ</td>
<td>Blank</td>
<td>Blank</td>
<td>.</td>
<td>Blank</td>
<td>.</td>
</tr>
</tbody>
</table>

- **Data record:**
  
<table>
<thead>
<tr>
<th><em>ID</em></th>
<th>Field1</th>
<th>Field2</th>
<th>Field3</th>
<th>Field4</th>
<th>Field5</th>
<th>Field6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number</td>
<td>Blank</td>
<td>Column name</td>
<td>Column name</td>
<td>Coefficient in objective function</td>
<td>Blank</td>
<td>.</td>
</tr>
</tbody>
</table>

**Notes:**

1. The QSECTION section can appear only in PROC OPTQP and should not appear in PROC OPTLP or PROC OPTMILP. For PROC OPTLP and PROC OPTMILP, an error is reported when the submitted data table contains a QSECTION section. For PROC OPTQP, if the QSECTION section is not specified, or if there is no valid data record in the QSECTION section, then the LP solver is used to solve the resulting linear programming problem.
2. The variables field2 and field3 contain the names of the columns that form a quadratic term in the objective function. They must have been defined in the COLUMNS section. They need not refer to the same column. Zero entries should not be specified.

3. Duplicate entries of a quadratic term are not allowed. This means the combination of (field2, field3) must be unique, where \((k, j)\) and \((j, k)\) are considered to be the same combination.

4. If field4 of one data record is missing or takes a value of zero, then this data record is ignored.

---

**ENDATA Section**

The ENDATA section simply declares the end of all records. It contains only one indicator record, where field1 takes the value ENDATA, _ID_ has a value larger than in all other observations, and the values of the remaining variables are blank or missing.

---

**Details: MPS-Format Data Table**

**Converting an MPS/QPS-Format File: %MPS2SASD**

As described in the section “Overview: MPS-Format Data Table” on page 255, the MPS or QPS format is a standard file format for describing linear, integer, and quadratic programming problems. The MPS/QPS format is defined for plain text files, whereas in the SAS System it is more convenient to read data from data sets or data tables. Therefore, a facility is required to convert MPS/QPS-format text files to MPS-format data sets or data tables. The SAS macro %MPS2SASD serves this purpose.

In the MPS/QPS-format text file, a record refers to a single line of text that is divided into six fields. MPS/QPS files can be read and printed in both fixed and free format. In fixed MPS/QPS format, each field of a data record must occur in specific columns:

<table>
<thead>
<tr>
<th>Field</th>
<th>Field 1</th>
<th>Field 2</th>
<th>Field 3</th>
<th>Field 4</th>
<th>Field 5</th>
<th>Field 6</th>
</tr>
</thead>
</table>

In free format, fields of a record are separated by a space. Both fixed and free format have limitations. If users need to use row names or column names longer than 8 characters, then there is not enough space to hold them in fixed format. If users use a space as a part of a row name or column name, such as “ROW NAME”, then free-format MPS format interprets this symbol as two fields, “ROW” and “NAME”.

You can insert a comment record, denoted by an asterisk (*) in column 1, anywhere in an MPS/QPS file. Also, if a dollar sign ($) is the first character in field 3 or field 5 of any record, the information from that point to the end of the record is treated as a comment. All comments are ignored by the %MPS2SASD macro, which is described as follows.

%MPS2SASD Macro Parameters

%MPS2SASD(MPSFILE='infilename', OUTDATA=mpsdata, MAXLEN=n, FORMAT=FIXED | FREE);

FORMAT=FIXED | FREE
specifies the format of the input MPS file. This parameter is optional. The default value is the one, if any, specified by the flat file, and FIXED otherwise. You can specify the following values:

  FIXED specifies that each field of a data record occurs in specific columns.
  FREE specifies that the fields of a data record are separated by a space.

MAXLEN=n
specifies length of the variables field2, field3, and field5 in the output MPS-format data table. This parameter is optional. By default, MAXLEN=8.

MPSFILE='infilename'
specifies the path and name of the input MPS-format file. The input file is a plain text file, usually with either an “.mps” extension for linear programming problems or a “.qps” extension for quadratic programming problems. This parameter is required; there is no default value.

OUTDATA=mpsdata
specifies the name of the output MPS-format data table. If a CAS library reference is specified before the name, the output data are directly stored as a data table on the CAS server. This parameter is optional. By default, OUTDATA=MPSData.

Length of Variables

In an MPS-format data table, the variables field2, field3, and field5 hold the names of the rows and columns. The length of these character variables is limited to the maximum size of a SAS character variable. This enables you to use sufficiently long names for the rows and columns in your model.

In a data table generated by the %MPS2SASD macro, the length of the variables field2, field3, and field5 is fixed to be 8 ASCII characters by default. This length fits the fixed-format MPS/QPS file well because field 2, field 3, and field 5 are fixed at 8 characters. However, the free-format MPS/QPS files might have longer row names or longer column names. The %MPS2SASD macro provides a parameter MAXLEN=n. Using this parameter, you can set the variables field2, field3, and field5 to have a length of n characters in the output data table.

The parameter MAXLEN works only when the given MPS file is in free format. For a fixed-format MPS file, this parameter is ignored and the length of field2, field3, and field5 is 8 characters by default.
Examples: MPS-Format Data Table

Example 9.1: MPS-Format Data Table for a Product Mix Problem

Consider a simple product mix problem where a furniture company tries to find an optimal product mix of four items: a desk ($x_1$), a chair ($x_2$), a cabinet ($x_3$), and a bookcase ($x_4$). Each item is processed in a stamping department (STAMP), an assembly department (ASSEMB), and a finishing department (FINISH). The time each item requires in each department is given in the input data. Because of resource limitations, each department has an upper limit on the time available for processing. Also, because of labor constraints, the assembly department must work at least 300 hours. Finally, the marketing department tells you not to make more than 75 chairs and to make at least 50 bookcases.

This problem can be expressed as the following linear program:

$$\begin{align*}
\text{maximize} & \quad 95x_1 + 41x_2 + 84x_3 + 76x_4 \\
\text{subject to} & \quad 3x_1 + 1.5x_2 + 2x_3 + 2x_4 \leq 800 \quad \text{(STAMP)} \\
& \quad 10x_1 + 6x_2 + 8x_3 + 7x_4 \leq 1200 \quad \text{(ASSEMB)} \\
& \quad 10x_1 + 6x_2 + 8x_3 + 7x_4 \geq 300 \quad \text{(ASSEMB)} \\
& \quad 10x_1 + 8x_2 + 8x_3 + 7x_4 \leq 800 \quad \text{(FINISH)} \\
& \quad x_2 \leq 75 \\
& \quad x_4 \geq 50 \\
& \quad x_i \geq 0 \quad i = 1, 2, 3
\end{align*}$$

The following DATA step saves the problem specification as an MPS-format data table:

```plaintext
data mycas.ProdMix;
infile datalines;
input field1 $ field2 $ field3$ field4 field5 $ field6 _ID_;
datalines;
NAME . PROD_MIX . . . 1
ROWS . . . . . 2
MAX PROFIT . . . . 3
L STAMP . . . . 4
L ASSEMB . . . . 5
L FINISH . . . . 6
COLUMNS . . . . . 7
. DESK STAMP 3.0 ASSEMB 10 8
. DESK FINISH 10.0 PROFIT 95 9
. CHAIR STAMP 1.5 ASSEMB 6 10
. CHAIR FINISH 8.0 PROFIT 41 11
. CABINET STAMP 2.0 ASSEMB 8 12
. CABINET FINISH 8.0 PROFIT 84 13
. BOOKCASE STAMP 2.0 ASSEMB 7 14
. BOOKCASE FINISH 7.0 PROFIT 76 15
RHS . . . . . . 16
. TIME STAMP 800.0 ASSEMB 1200 17
. TIME FINISH 800.0 . . 18
RANGES . . . . . 19
. T1 ASSEMB 900.0 . . 20
```

The following DATA step saves the problem specification as an MPS-format data table:
**Example 9.2: Fixed-MPS-Format File**

The following file, example_fix.mps, contains the data from Example 9.1 in the form of a fixed-MPS-format file. The indicator codes MAX and MIN are not available for objective rows in fixed MPS format, so the PROFIT row is specified as type N. Minimization is assumed for type-N rows; for a maximization objective, the objective coefficients must be replaced with values of the opposite sign.

* THIS IS AN EXAMPLE FOR FIXED MPS FORMAT.

<table>
<thead>
<tr>
<th>NAME</th>
<th>PROD_MIX</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>ROWS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>PROFIT</td>
</tr>
<tr>
<td>L</td>
<td>STAMP</td>
</tr>
<tr>
<td>L</td>
<td>ASSEMB</td>
</tr>
<tr>
<td>L</td>
<td>FINISH</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>COLUMNS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DESK</td>
<td>STAMP</td>
</tr>
<tr>
<td>DESK</td>
<td>FINISH</td>
</tr>
<tr>
<td>CHAIR</td>
<td>STAMP</td>
</tr>
<tr>
<td>CHAIR</td>
<td>FINISH</td>
</tr>
<tr>
<td>CABINET</td>
<td>STAMP</td>
</tr>
<tr>
<td>CABINET</td>
<td>FINISH</td>
</tr>
<tr>
<td>BOOKCSE</td>
<td>STAMP</td>
</tr>
<tr>
<td>BOOKCSE</td>
<td>FINISH</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RHS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TIME</td>
<td>STAMP</td>
</tr>
<tr>
<td>TIME</td>
<td>FINISH</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RANGES</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>ASSEMB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BOUNDS</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>UP</td>
<td>BND</td>
</tr>
<tr>
<td>LO</td>
<td>BND</td>
</tr>
</tbody>
</table>

**Example 9.3: Free-MPS-Format File**

In free format, fields in data records other than the first record have no predefined positions. They can be written anywhere except column 1, with each field separated from the next by one or more blanks (a tab cannot be used as a field separator). However, the fields must appear in the same sequence as in the fixed format. The following file, example_free.mps, is an example. It describes the same problem as in Example 9.2.
## Example 9.4: Using the %MPS2SASD Macro

This example illustrates the use of the %MPS2SASD macro, assuming that the files example_fix.mps and example_free.mps are in your current SAS working directory.

The %MPS2SASD macro function has one required parameter, MPSFILE='infilename', which specifies the path and name of the MPS/QPS-format file. With this single parameter, the macro reads the file, converts the records, and saves the conversion to the default MPS-format data set MPSData. To save the conversion to a CAS library reference, the additional parameter OUTDATA= is required and needs to include a caslib in the name.

Running the following statements converts the fixed-format MPS file shown in Example 9.2 to the MPS-format data table ProdMix:

```sas
%mps2sasd(mpsfile='example_fix.mps', outdata=mycas.ProdMix);
proc print data=mycas.ProdMix noobs;
run;
```

Output 9.4.1 displays the MPS-format data table ProdMix.
### Output 9.4.1 The MPS-Format Data Table ProdMix

<table>
<thead>
<tr>
<th>field1</th>
<th>field2</th>
<th>field3</th>
<th>field4</th>
<th>field5</th>
<th>field6</th>
<th><em>id</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>PROD_MIX</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>STAMP</td>
<td>.</td>
<td>.</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COLUMNS</td>
<td></td>
<td>.</td>
<td>.</td>
<td>8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHAIR</td>
<td>STAMP</td>
<td>1.5</td>
<td>ASSEMB</td>
<td>6</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>CABINET</td>
<td>FINISH</td>
<td>8.0</td>
<td>PROFIT</td>
<td>-84</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>RHS</td>
<td></td>
<td>.</td>
<td>.</td>
<td>17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RANGES</td>
<td></td>
<td>.</td>
<td>.</td>
<td>20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UP</td>
<td>BND</td>
<td>CHAIR</td>
<td>75.0</td>
<td>.</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>ROWS</td>
<td></td>
<td>.</td>
<td>.</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>ASSEMB</td>
<td>.</td>
<td>.</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DESK</td>
<td>STAMP</td>
<td>3.0</td>
<td>ASSEMB</td>
<td>10</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>CHAIR</td>
<td>FINISH</td>
<td>8.0</td>
<td>PROFIT</td>
<td>-41</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>BOOKCSE</td>
<td>STAMP</td>
<td>2.0</td>
<td>ASSEMB</td>
<td>7</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>STAMP</td>
<td>800.0</td>
<td>ASSEMB</td>
<td>1200</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>T1</td>
<td>ASSEMB</td>
<td>900.0</td>
<td>.</td>
<td>21</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LO</td>
<td>BND</td>
<td>BOOKCSE</td>
<td>50.0</td>
<td>.</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>PROFIT</td>
<td>.</td>
<td>.</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>FINISH</td>
<td>.</td>
<td>.</td>
<td>7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DESK</td>
<td>FINISH</td>
<td>10.0</td>
<td>PROFIT</td>
<td>-95</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>CABINET</td>
<td>STAMP</td>
<td>2.0</td>
<td>ASSEMB</td>
<td>8</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>BOOKCSE</td>
<td>FINISH</td>
<td>7.0</td>
<td>PROFIT</td>
<td>-76</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>TIME</td>
<td>FINISH</td>
<td>800.0</td>
<td>.</td>
<td>19</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BOUNDS</td>
<td></td>
<td>.</td>
<td>.</td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENDATA</td>
<td></td>
<td>.</td>
<td>.</td>
<td>25</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that the _ID_ variable has been added automatically and its index is increasing. The order is not preserved on the CAS server, but sorting by _ID_ would put the rows in the correct order. It does not start at 1 because of the comment in the input file.

Running the following statement converts the free-format MPS file shown in Example 9.3 to the MPS-format data table ProdMix on the caslib mycas:

```sas
%mps2sasd(mpsfile='example_free.mps', outdata=mycas.ProdMix);
```

This data table is identical to the one shown in Output 9.4.1.

In the following statement, when the free-format MPS file is converted, the length of the variables field2, field3, and field5 in the data table ProdMix is explicitly set to 10 characters:

```sas
%mps2sasd(mpsfile='example_free.mps', outdata=mycas.ProdMix, maxlen=10, format=free);
```
References


# Chapter 10
## The OPTMODEL Procedure

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview: OPTMODEL Procedure</td>
<td>272</td>
</tr>
<tr>
<td>Getting Started: OPTMODEL Procedure</td>
<td>273</td>
</tr>
<tr>
<td>An Unconstrained Optimization Example</td>
<td>274</td>
</tr>
<tr>
<td>The Rosenbrock Problem</td>
<td>277</td>
</tr>
<tr>
<td>A Transportation Problem</td>
<td>278</td>
</tr>
<tr>
<td>Syntax: OPTMODEL Procedure</td>
<td>280</td>
</tr>
<tr>
<td>Functional Summary</td>
<td>282</td>
</tr>
<tr>
<td>PROC OPTMODEL Statement</td>
<td>284</td>
</tr>
<tr>
<td>Declaration Statements</td>
<td>288</td>
</tr>
<tr>
<td>Programming Statements</td>
<td>297</td>
</tr>
<tr>
<td>Details: OPTMODEL Procedure</td>
<td>339</td>
</tr>
<tr>
<td>Named Parameters</td>
<td>339</td>
</tr>
<tr>
<td>Indexing</td>
<td>340</td>
</tr>
<tr>
<td>Types</td>
<td>342</td>
</tr>
<tr>
<td>Names</td>
<td>343</td>
</tr>
<tr>
<td>Parameters</td>
<td>343</td>
</tr>
<tr>
<td>Expressions</td>
<td>344</td>
</tr>
<tr>
<td>Identifier Expressions</td>
<td>347</td>
</tr>
<tr>
<td>Function Expressions</td>
<td>347</td>
</tr>
<tr>
<td>Index Sets</td>
<td>348</td>
</tr>
<tr>
<td>Nil Values</td>
<td>349</td>
</tr>
<tr>
<td>OPTMODEL Expression Extensions</td>
<td>351</td>
</tr>
<tr>
<td>Conditions of Optimality</td>
<td>361</td>
</tr>
<tr>
<td>Data Set Input/Output</td>
<td>364</td>
</tr>
<tr>
<td>Control Flow</td>
<td>368</td>
</tr>
<tr>
<td>Formatted Output</td>
<td>368</td>
</tr>
<tr>
<td>ODS Table and Variable Names</td>
<td>371</td>
</tr>
<tr>
<td>Constraints</td>
<td>376</td>
</tr>
<tr>
<td>Suffixes</td>
<td>380</td>
</tr>
<tr>
<td>Integer Variable Suffixes</td>
<td>384</td>
</tr>
<tr>
<td>Dual Values</td>
<td>385</td>
</tr>
<tr>
<td>Reduced Costs</td>
<td>391</td>
</tr>
<tr>
<td>Presolver</td>
<td>392</td>
</tr>
<tr>
<td>Model Update</td>
<td>392</td>
</tr>
<tr>
<td>Multiple Subproblems</td>
<td>397</td>
</tr>
<tr>
<td>Multiple Solutions</td>
<td>398</td>
</tr>
</tbody>
</table>
Overview: OPTMODEL Procedure

The OPTMODEL procedure includes the powerful OPTMODEL modeling language and state-of-the-art solvers for several classes of mathematical programming problems. The problems and their solvers are listed in Table 10.1.

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solver</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint programming</td>
<td>CLP</td>
</tr>
<tr>
<td>Linear programming</td>
<td>LP</td>
</tr>
<tr>
<td>Local search optimization</td>
<td>LSO</td>
</tr>
<tr>
<td>Mixed integer linear programming</td>
<td>MILP</td>
</tr>
<tr>
<td>Network algorithms</td>
<td>Network</td>
</tr>
<tr>
<td>General nonlinear programming</td>
<td>NLP</td>
</tr>
<tr>
<td>Quadratic programming</td>
<td>QP</td>
</tr>
</tbody>
</table>

The OPTMODEL modeling language provides a modeling environment tailored to building, solving, and maintaining optimization models. This makes the process of translating the symbolic formulation of an optimization model into OPTMODEL virtually transparent since the modeling language mimics the symbolic algebra of the formulation as closely as possible. The OPTMODEL language also streamlines and simplifies
the critical process of populating optimization models with data from SAS data sets. All of this transparency produces models that are more easily inspected for completeness and correctness, more easily corrected, and more easily modified, whether through structural changes or through the substitution of new data for old.

In addition to invoking optimization solvers directly with PROC OPTMODEL as already mentioned, you can use the OPTMODEL language purely as a modeling facility. You can save optimization models built with the OPTMODEL language in SAS data sets that can be submitted to other SAS/OR optimization procedures. In general, the OPTMODEL language serves as a common point of access for many of the SAS/OR optimization capabilities, whether providing both modeling and solver access or acting as a modeling interface for other optimization procedures.

For details and examples of the problems addressed and corresponding solvers, please see the dedicated chapters in this book. This chapter aims to give you a comprehensive understanding of the OPTMODEL procedure by discussing the framework provided by the OPTMODEL modeling language. For additional examples that demonstrate the features of the OPTMODEL procedure, see SAS/OR User’s Guide: Mathematical Programming Examples.

The OPTMODEL modeling language features automatic differentiation, advanced flow control, optimization-oriented syntax (parameters, variables, arrays, constraints, objective functions), dynamic model generation, model-data separation, and transparent access to SAS data sets.

---

**Getting Started: OPTMODEL Procedure**

Optimization or mathematical programming is a search for a maximum or minimum of an *objective function* (also called a *cost function*), where search variables are restricted to particular constraints. Constraints are said to define a *feasible region* (see Figure 10.1).

**Figure 10.1** Examples of Feasible Regions

A more rigorous general formulation of such problems is as follows.

Let

\[ f : S \rightarrow \mathbb{R} \]
be a real-valued function. Find $x^*$ such that

- $x^* \in S$
- $f(x^*) \leq f(x)$, $\forall x \in S$

Note that the formulation is for the minimum of $f$ and that the maximum of $f$ is simply the negation of the minimum of $-f$.

Here, function $f$ is the objective function, and the variable in the objective function is called the optimization variable (or decision variable). $S$ is the feasible region. Typically $S$ is a subset of the Euclidean space $\mathbb{R}^n$ specified by the set of constraints, which are often a set of equalities (=) or inequalities ($\leq$, $\geq$) that every element in $S$ is required to satisfy simultaneously. For the special case where $S = \mathbb{R}^n$, the problem is an unconstrained optimization. An element $x$ of $S$ is called a feasible solution to the optimization problem, and the value $f(x)$ is called the objective value. A feasible solution $x^*$ that minimizes the objective function is called an optimal solution to the optimization problem, and the corresponding objective value is called the optimal value.

In mathematics, special notation is used to denote an optimization problem. Generally, you can write an optimization problem as follows:

\[
\begin{align*}
\text{minimize} & \quad f(x) \\
\text{subject to} & \quad x \in S
\end{align*}
\]

Normally, an empty body of constraint (the part after “subject to”) implies that the optimization is unconstrained (that is, the feasible region is the whole space $\mathbb{R}^n$). The optimal solution ($x^*$) is denoted as

\[x^* = \arg\min_{x \in S} f(x)\]

The optimal value ($f(x^*)$) is denoted as

\[f(x^*) = \min_{x \in S} f(x)\]

Optimization problems can be classified by the forms (linear, quadratic, nonlinear, and so on) of the functions in the objective and constraints. For example, a problem is said to be linearly constrained if the functions in the constraints are linear. A linear programming problem is a linearly constrained problem with a linear objective function. A nonlinear programming problem occurs where some function in the objective or constraints is nonlinear, and so on.

---

**An Unconstrained Optimization Example**

An unconstrained optimization problem formulation is simply

\[
\begin{align*}
\text{minimize} & \quad f(x)
\end{align*}
\]
An Unconstrained Optimization Example

For example, suppose you wanted to find the minimum value of this polynomial:

\[ z(x, y) = x^2 - x - 2y - xy + y^2 \]

You can compactly specify and solve the optimization problem by using the OPTMODEL modeling language. Here is the program:

```plaintext
/* invoke procedure */
proc optmodel;
  var x, y; /* declare variables */

  /* objective function */
  min z=x**2 - x - 2*y - x*y + y**2;

  /* now run the solver */
  solve;

  print x y;
quit;
```

This program produces the output in Figure 10.2.

**Figure 10.2** Optimizing a Simple Polynomial

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
<td>Minimization</td>
</tr>
<tr>
<td>Objective Function</td>
<td>z</td>
</tr>
<tr>
<td>Objective Type</td>
<td>Quadratic</td>
</tr>
<tr>
<td>Number of Variables</td>
<td>2</td>
</tr>
<tr>
<td>Bounded Above</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
<td>0</td>
</tr>
<tr>
<td>Free</td>
<td>2</td>
</tr>
<tr>
<td>Fixed</td>
<td>0</td>
</tr>
<tr>
<td>Number of Constraints</td>
<td>0</td>
</tr>
<tr>
<td>Constraint Coefficients</td>
<td>0</td>
</tr>
<tr>
<td>Hessian Diagonal Elements</td>
<td>2</td>
</tr>
<tr>
<td>Hessian Elements Below Diagonal</td>
<td>1</td>
</tr>
</tbody>
</table>
In PROC OPTMODEL you specify the mathematical formulas that describe the behavior of the optimization problem that you want to solve. In the preceding example there were two independent variables in the polynomial, \( x \) and \( y \). These are the optimization variables of the problem. In PROC OPTMODEL you declare optimization variables with the \texttt{VAR} statement. The formula that defines the quantity that you are seeking to optimize is called the objective function, or objective. The solver varies the values of the optimization variables when searching for an optimal value for the objective.

In the preceding example the objective function is named \( z \), declared with the \texttt{MIN} statement. The keyword \texttt{MIN} is an abbreviation for MINIMIZE. The expression that follows the equal sign (=) in the MIN statement defines the function to be minimized in terms of the optimization variables.

The \texttt{VAR} and \texttt{MIN} statements are just two of the many available PROC OPTMODEL declaration and programming statements. PROC OPTMODEL processes all such statements interactively, meaning that each statement is processed as soon as it is complete.

After PROC OPTMODEL has completed processing of declaration and programming statements, it processes the \texttt{SOLVE} statement, which submits the problem to a solver and prints a summary of the results. The \texttt{PRINT} statement displays the optimal values of the optimization variables \( x \) and \( y \) found by the solver.

It is worth noting that PROC OPTMODEL does not use a RUN statement but instead operates on an interactive basis throughout. You can continue to interact with PROC OPTMODEL even after invoking a solver. For example, you could modify the problem and issue another SOLVE statement (see the section “Model Update” on page 392).
The Rosenbrock Problem

You can use parameters to produce a clear formulation of a problem. Consider the Rosenbrock problem,

\[
\text{minimize } f(x_1, x_2) = \alpha (x_2 - x_1^2)^2 + (1 - x_1)^2
\]

where \( \alpha = 100 \) is a parameter (constant), \( x_1 \) and \( x_2 \) are optimization variables (whose values are to be determined), and \( f(x_1, x_2) \) is an objective function.

Here is a PROC OPTMODEL program that solves the Rosenbrock problem:

```optmodel
proc optmodel;
    number alpha = 100; /* declare parameter */
    var x {1..2};      /* declare variables */
    /* objective function */
    min f = alpha*(x[2] - x[1]**2)**2 +
           (1 - x[1])**2;
    /* now run the solver */
    solve;

    print x;
quit;
```

The PROC OPTMODEL output is shown in Figure 10.3.

**Figure 10.3** Rosenbrock Function Results

<table>
<thead>
<tr>
<th>The OPTMODEL Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Problem Summary</strong></td>
</tr>
<tr>
<td><strong>Objective Sense</strong></td>
</tr>
<tr>
<td><strong>Objective Function</strong></td>
</tr>
<tr>
<td><strong>Objective Type</strong></td>
</tr>
<tr>
<td><strong>Number of Variables</strong></td>
</tr>
<tr>
<td><strong>Bounded Above</strong></td>
</tr>
<tr>
<td><strong>Bounded Below</strong></td>
</tr>
<tr>
<td><strong>Bounded Below and Above</strong></td>
</tr>
<tr>
<td><strong>Free</strong></td>
</tr>
<tr>
<td><strong>Fixed</strong></td>
</tr>
<tr>
<td><strong>Number of Constraints</strong></td>
</tr>
</tbody>
</table>
You can easily translate the symbolic formulation of a problem into the OPTMODEL procedure. Consider the transportation problem, which is mathematically modeled as the following linear programming problem:

\[
\text{minimize} \quad \sum_{i \in O, j \in D} c_{ij} x_{ij}
\]

subject to

\[
\sum_{j \in D} x_{ij} = a_i, \quad \forall i \in O \quad \text{(SUPPLY)}
\]

\[
\sum_{i \in O} x_{ij} = b_j, \quad \forall j \in D \quad \text{(DEMAND)}
\]

\[
x_{ij} \geq 0, \quad \forall (i, j) \in O \times D
\]

where \(O\) is the set of origins, \(D\) is the set of destinations, \(c_{ij}\) is the cost to transport one unit from \(i\) to \(j\), \(a_i\) is the supply of origin \(i\), \(b_j\) is the demand of destination \(j\), and \(x_{ij}\) is the decision variable for the amount of shipment from \(i\) to \(j\).

Here is a very simple example. The cities in the set \(O\) of origins are Detroit and Pittsburgh. The cities in the set \(D\) of destinations are Boston and New York. The cost matrix, supply, and demand are shown in Table 10.2.

Table 10.2 A Transportation Problem

<table>
<thead>
<tr>
<th></th>
<th>Boston</th>
<th>New York</th>
<th>Supply</th>
</tr>
</thead>
<tbody>
<tr>
<td>Detroit</td>
<td>30</td>
<td>20</td>
<td>200</td>
</tr>
<tr>
<td>Pittsburgh</td>
<td>40</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>Demand</td>
<td>150</td>
<td>150</td>
<td></td>
</tr>
</tbody>
</table>
The problem is compactly and clearly formulated and solved by using the OPTMODEL procedure with the following statements:

```plaintext
proc optmodel;
    /* specify parameters */
    set O={'Detroit','Pittsburgh'};
    set D={'Boston','New York'};
    number c{O,D}=[30 20
        40 10];
    number a{O}=[200 100];
    number b{D}=[150 150];
    /* model description */
    var x{O,D} >= 0;
    min total_cost = sum{i in O, j in D}c[i,j]*x[i,j];
    constraint supply{i in O}: sum{j in D}x[i,j]=a[i];
    constraint demand{j in D}: sum{i in O}x[i,j]=b[j];
    /* solve and output */
    solve;
    print x;
```

The output is shown in Figure 10.4.

**Figure 10.4** Solution to the Transportation Problem

### The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objective Sense</strong></td>
</tr>
<tr>
<td><strong>Objective Function</strong></td>
</tr>
<tr>
<td><strong>Objective Type</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Variables</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded Above</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below</td>
<td>4</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
<td>0</td>
</tr>
<tr>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>Fixed</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Constraints</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear LE (&lt;=)</td>
<td>0</td>
</tr>
<tr>
<td>Linear EQ (=)</td>
<td>4</td>
</tr>
<tr>
<td>Linear GE (&gt;=)</td>
<td>0</td>
</tr>
<tr>
<td>Linear Range</td>
<td>0</td>
</tr>
</tbody>
</table>

| Constraint Coefficients | 8 |
Syntax: OPTMODEL Procedure

PROC OPTMODEL statements are divided into three categories: the PROC statement, the declaration statements, and the programming statements. The PROC statement invokes the procedure and sets initial option values. The declaration statements declare optimization model components. The programming statements read and write data, invoke the solver, and print results. In the following text, the statements are listed in the order in which they are grouped, with declaration statements first.

NOTE: Solver specific options are described in the individual chapters that correspond to the solvers.
PROC OPTMODEL options ;

Declaration Statements:
CONSTRAINT constraints ;
IMPVAR optimization expression declarations ;
MAX objective ;
MIN objective ;
NUMBER parameter declarations ;
PROBLEM problem declaration ;
SET [ < types > ] parameter declarations ;
STRING parameter declarations ;
VAR variable declarations ;

Programming Statements:
Assignment parameter = expression ;
CALL name [( expressions ) ] ;
CLOSEFILE files ;
COFOR { index-set } statement ;
CONTINUE ;
CREATE DATA SAS-data-set FROM columns ;
DO ; statements ; END ;
DO variable = specifications ; statements ; END ;
DO UNTIL ( logic ) ; statements ; END ;
DO WHILE ( logic ) ; statements ; END ;
DROP constraint ;
EXPAND name [ / options ] ;
FILE file ;
FIX variable [ = expression ] ;
FOR { index-set } statement ;
IF logic THEN statement ; [ ELSE statement ] ;
LEAVE ;
Null ;
PRINT print items ;
PROFILE [ mode ] options ;
PUT put items ;
QUIT ;
READ DATA SAS-data-set INTO columns ;
RESET OPTIONS options ;
RESTORE constraint ;
SAVE MPS SAS-data-set [ OBJECTIVE name ] [ NOOBJECTIVE ] ;
SAVE QPS SAS-data-set [ OBJECTIVE name ] [ NOOBJECTIVE ] ;
SOLVE [ WITH solver ] [ OBJECTIVE name ] [ NOOBJECTIVE ] [ RELAXINT ] [ / options ] ;
STOP ;
SUBMIT arguments [ / options ] ;
UNFIX variable [ = expression ] ;
USE PROBLEM problem ;
# Functional Summary

The statements and options available with PROC OPTMODEL are summarized by purpose in Table 10.3.

## Table 10.3 Functional Summary

<table>
<thead>
<tr>
<th>Description</th>
<th>Statement</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Declaration Statements:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Declares a constraint</td>
<td>CONSTRAINT</td>
<td></td>
</tr>
<tr>
<td>Declares optimization expressions</td>
<td>IMPVAR</td>
<td></td>
</tr>
<tr>
<td>Declares a maximization objective</td>
<td>MAX</td>
<td></td>
</tr>
<tr>
<td>Declares a minimization objective</td>
<td>MIN</td>
<td></td>
</tr>
<tr>
<td>Declares a number type parameter</td>
<td>NUMBER</td>
<td></td>
</tr>
<tr>
<td>Declares a problem</td>
<td>PROBLEM</td>
<td></td>
</tr>
<tr>
<td>Declares a set type parameter</td>
<td>SET</td>
<td></td>
</tr>
<tr>
<td>Declares a string type parameter</td>
<td>STRING</td>
<td></td>
</tr>
<tr>
<td>Declares optimization variables</td>
<td>VAR</td>
<td></td>
</tr>
<tr>
<td><strong>Programming Statements:</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Assigns a value to a variable or parameter</td>
<td>=</td>
<td></td>
</tr>
<tr>
<td>Invokes a library subroutine</td>
<td>CALL</td>
<td></td>
</tr>
<tr>
<td>Closes the opened file</td>
<td>CLOSEFILE</td>
<td></td>
</tr>
<tr>
<td>Executes the statement repeatedly with support for concurrent solver invocations</td>
<td>COFOR</td>
<td></td>
</tr>
<tr>
<td>Terminates one iteration of a loop statement</td>
<td>CONTINUE</td>
<td></td>
</tr>
<tr>
<td>Creates a new SAS data set and copies data into it from PROC OPTMODEL parameters and variables</td>
<td>CREATE DATA</td>
<td></td>
</tr>
<tr>
<td>Groups a sequence of statements together as a single statement</td>
<td>DO</td>
<td></td>
</tr>
<tr>
<td>Executes statements repeatedly</td>
<td>DO (iterative)</td>
<td></td>
</tr>
<tr>
<td>Executes statements repeatedly until some condition is satisfied</td>
<td>DO UNTIL</td>
<td></td>
</tr>
<tr>
<td>Executes statements repeatedly as long as some condition is satisfied</td>
<td>DO WHILE</td>
<td></td>
</tr>
<tr>
<td>Ignores the specified constraint</td>
<td>DROP</td>
<td></td>
</tr>
<tr>
<td>Prints the specified constraint, variable, or objective declaration expressions after expanding aggregation operators, and so on</td>
<td>EXPAND</td>
<td></td>
</tr>
<tr>
<td>Selects a file for the PUT statement</td>
<td>FILE</td>
<td></td>
</tr>
<tr>
<td>Treats a variable as fixed in value</td>
<td>FIX</td>
<td></td>
</tr>
<tr>
<td>Executes the statement repeatedly</td>
<td>FOR</td>
<td></td>
</tr>
<tr>
<td>Executes the statement conditionally</td>
<td>IF</td>
<td></td>
</tr>
<tr>
<td>Terminates the execution of the entire loop body</td>
<td>LEAVE</td>
<td></td>
</tr>
<tr>
<td>Null statement</td>
<td>;</td>
<td></td>
</tr>
<tr>
<td>Outputs string and numeric data</td>
<td>PRINT</td>
<td></td>
</tr>
</tbody>
</table>
## Functional Summary

<table>
<thead>
<tr>
<th>Description</th>
<th>Statement</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Provides timing and execution count information for statements and declarations</td>
<td>PROFILE</td>
<td></td>
</tr>
<tr>
<td>Writes text data to the current output file</td>
<td>PUT</td>
<td></td>
</tr>
<tr>
<td>Terminates the PROC OPTMODEL session</td>
<td>QUIT</td>
<td></td>
</tr>
<tr>
<td>Reads data from a SAS data set into PROC OPTMODEL parameters and variables</td>
<td>READ DATA</td>
<td></td>
</tr>
<tr>
<td>Sets PROC OPTMODEL option values or restores them to their defaults</td>
<td>RESET OPTIONS</td>
<td></td>
</tr>
<tr>
<td>Adds a constraint that was previously dropped back into the model</td>
<td>RESTORE</td>
<td></td>
</tr>
<tr>
<td>Saves the structure and coefficients for a linear programming model into a SAS data set</td>
<td>SAVE MPS</td>
<td></td>
</tr>
<tr>
<td>Saves the structure and coefficients for a quadratic programming model into a SAS data set</td>
<td>SAVE QPS</td>
<td></td>
</tr>
<tr>
<td>Invokes a PROC OPTMODEL solver</td>
<td>SOLVE</td>
<td></td>
</tr>
<tr>
<td>Halts the execution of all statements that contain it</td>
<td>STOP</td>
<td></td>
</tr>
<tr>
<td>Submits SAS code for execution</td>
<td>SUBMIT</td>
<td></td>
</tr>
<tr>
<td>Reverses the effect of FIX statement</td>
<td>UNFIX</td>
<td></td>
</tr>
<tr>
<td>Selects the current problem</td>
<td>USE PROBLEM</td>
<td></td>
</tr>
</tbody>
</table>

### PROC OPTMODEL Options:

- Specifies the accuracy for nonlinear constraints
  - `PROC OPTMODEL CDIGITS=`
- Specifies the maximum number of error messages displayed
  - `PROC OPTMODEL ERRORLIMIT=`
- Specifies the method used to approximate numeric derivatives
  - `PROC OPTMODEL FD=`
- Specifies the accuracy for the objective function
  - `PROC OPTMODEL FDIGITS=`
- Forces finite differences to be used for nonlinear equations
  - `PROC OPTMODEL FORCEFD=`
- Enables the OPTMODEL presolver for the CLP, LP, MILP, and QP solvers
  - `PROC OPTMODEL FORCEPRESOLVE=`
- Passes initial values for variables to the solver
  - `PROC OPTMODEL INITVAR/NOINITVAR`
- Specifies the tolerance for rounding the bounds on integer and binary variables
  - `PROC OPTMODEL INTFUZZ=`
- Specifies the maximum length for MPS row and column labels
  - `PROC OPTMODEL MAXLABELN=`
- Checks missing values
  - `PROC OPTMODEL MISSCHECK/NOMISSCHECK`
- Specifies the maximum number of non-error messages displayed
  - `PROC OPTMODEL MSGLIMIT=`
- Specifies the number for threads to use for threaded processing
  - `PROC OPTMODEL NTHREADS=`
- Specifies the number of digits to display
  - `PROC OPTMODEL PDIGITS=`
- Adjusts how two-dimensional array is displayed
  - `PROC OPTMODEL PMATRIX=`
- Specifies the type of presolve performed by the PROC OPTMODEL presolver
  - `PROC OPTMODEL PRESOLVER=`
Chapter 10: The OPTMODEL Procedure

### Description Statement Option

<table>
<thead>
<tr>
<th>Description</th>
<th>Statement</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies the tolerance, enabling the PROC OPTMODEL presolver to remove slightly infeasible constraints</td>
<td>PROC OPTMODEL</td>
<td>PRESTOL=</td>
</tr>
<tr>
<td>Enables or disables printing summary</td>
<td>PROC OPTMODEL</td>
<td>PRINTLEVEL=</td>
</tr>
<tr>
<td>Specifies the width to display numeric columns</td>
<td>PROC OPTMODEL</td>
<td>PWIDTH=</td>
</tr>
<tr>
<td>Specifies the CAS session name</td>
<td>PROC OPTMODEL</td>
<td>SESSREF=</td>
</tr>
<tr>
<td>Specifies the CAS session UUID</td>
<td>PROC OPTMODEL</td>
<td>SESSUUID=</td>
</tr>
<tr>
<td>Specifies the smallest difference that is permitted by the PROC OPTMODEL presolver between the upper and lower bounds of an unfixed variable</td>
<td>PROC OPTMODEL</td>
<td>VARFUZZ=</td>
</tr>
</tbody>
</table>

### PROC OPTMODEL Statement

```plaintext
PROC OPTMODEL [ options ] ;
```

The PROC OPTMODEL statement invokes the OPTMODEL procedure. You can specify options to control how the optimization model is processed and how results are displayed. You can specify the following options (these options can also be specified in the RESET OPTIONS statement).

**CDIGITS=number**

specifies the expected number of decimal digits of accuracy for nonlinear constraints. The value can be fractional. PROC OPTMODEL uses this option to choose a step length when numeric derivative approximations are required to evaluate the Jacobian of nonlinear constraints. The default value depends on your operating environment. It is assumed that constraint values are accurate to the limits of machine precision.

See the section “Automatic Differentiation” on page 401 for more information about numeric derivative approximations.

**ERRORLIMIT=number | NONE**

specifies the maximum number of error messages that can be displayed during SOLVE statement processing. Specifying a value of `number` in the range $1 \text{ to } 2^{31} - 1$ sets a specific limit. Specifying `ERRORLIMIT=NONE` removes any existing limit. The default value is 10.

**NOTE:** Some errors abort processing immediately.

**FD=FORWARD | CENTRAL**

selects the method used to approximate numeric derivatives when analytic derivatives are unavailable. Most solvers require the derivatives of the objective and constraints. You can specify the following values:

- **FORWARD** uses forward differences.
- **CENTRAL** uses central differences.
By default, FD=FORWARD. For more information about numeric derivative approximations, see the section “Automatic Differentiation” on page 401.

**FDIGITS=** *number*

specifies the expected number of decimal digits of accuracy for the objective function. The value can be fractional. PROC OPTMODEL uses the value to choose a step length when numeric derivatives are required. The default value depends on your operating environment. It is assumed that objective function values are accurate to the limits of machine precision.

For more information about numeric derivative approximations, see the section “Automatic Differentiation” on page 401.

**FORCEFD=** *NONE | OBJ | CON | ALL*

forces PROC OPTMODEL to use finite differences instead of analytic derivatives for the specified set of nonlinear expressions. This option can be useful with FCMP functions to provide more control over derivative computation. You can specify the following values:

- **ALL** restricts all derivative computations to use finite differences.
- **CON** restricts derivative computations for the nonlinear constraint expressions and any IMPVAR expressions they reference to use finite differences.
- **NONE** requests analytic derivatives where they are available.
- **OBJ** restricts derivative computations for the objective and any IMPVAR expressions it references to use finite differences.

By default, FORCEFD=NONE.

**FORCEPRESOLVE=** *number | string*

specifies whether PROC OPTMODEL can use the OPTMODEL presolver with the CLP, LP, MILP, and QP solvers. By default, the OPTMODEL presolver is disabled when PROC OPTMODEL solves linear problems or problems with predicates, or when the CLP, LP, MILP, or QP solver is specified in the SOLVE statement. Table 10.4 shows the valid values for this option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OFF</td>
<td>Restores the default behavior.</td>
</tr>
<tr>
<td>1</td>
<td>ON</td>
<td>Enables PROC OPTMODEL to use the OPTMODEL presolver when the CLP, LP, MILP, or QP solver is specified in the SOLVE statement.</td>
</tr>
</tbody>
</table>

By default, FORCEPRESOLVE=0.

**INITVAR | NOINITVAR**

selects whether or not to pass initial values for variables to the solver when the SOLVE statement is executed. INITVAR enables the current variable values to be passed. NOINITVAR causes the solver to be invoked without any specific initial values for variables. The INITVAR option is the default.

The CLP, LP, and QP solvers always ignore initial values. The NLP solvers attempt to use specified initial values. The LSO and MILP solvers use initial values only if the PRIMALIN option is specified.
INTFUZZ=number
specifies the tolerance for rounding the bounds on integer and binary variables to integer values. Bounds that differ from an integer by at most number are rounded to that integer. Otherwise, lower bounds are rounded up to the next greater integer and upper bounds are rounded down to the next lesser integer. The value of number can range between 0 and 0.5. The default value is 0.00001.

MAXLABLEN=number
specifies the maximum length for MPS row and column labels. The allowed range is 8 to 256. This option can also be used to control the length of row and column names displayed by solvers, such as those found in the LP solver iteration log. See also the description of the .label suffix in the section “Suffixes” on page 380. By default, MAXLABLEN=32.

MISSCHECK | NOMISSCHECK
enables detailed checking of missing values in expressions. MISSCHECK requests that PROC OPTMODEL produce a message each time it evaluates an arithmetic operation or function that has missing value operands (except when the operation or function specifically supports missing values). The MISSCHECK option can increase processing time. NOMISSCHECK turns off this detailed reporting. NOMISSCHECK is the default.

MSGLIMIT=number | NONE
specifies the maximum number of messages about certain issues that can be displayed during processing of a single top-level statement, such as a SOLVE or FOR statement. Specifying a value of number in the range 0 to $2^{31} - 1$ sets a specific limit. Specifying MSGLIMIT=NONE removes any existing limit. The default value is 25.

The limit is applied to notes and warnings for the following issues:

- arithmetic evaluation issues, such as division by zero
- function evaluation issues, such as invalid arguments
- problem generation and PROC OPTMODEL presolver issues
- duplicate members in set constructor and literal expressions
- string concatenation results truncated to the maximum string length
- duplicate READ DATA keys
- truncated data set column labels for the CREATE DATA statement
- misspelled keywords for an option value specified using a string expression
- unrecognized file specifications in the CLOSEFILE statement

**NOTE:** Because of complications of concurrent execution, PROC OPTMODEL might display more or fewer messages than the limit when you use a COFOR statement.

NTHREADS=number
specifies the maximum number of threads to use for threaded processing. Problem generation and many of the solvers can take advantage of threaded processing. The value of number must be an integer between 1 and 256, inclusive. The default depends on the execution environment. When the distributed computing environment is used, the default is the maximum number of threads that are allowed on each computing node. Otherwise the default depends on the SAS system options CPUCOUNT= and THREADS. For more information, see the section “Threaded and Distributed Processing” on page 407.
requests that the PRINT statement display \textit{number} significant digits for numeric columns for which no format is specified. The value can range from 1 to 9. By default, PDIGITS=5.

\textbf{PMATRIX=number}

adjusts the density evaluation of a two-dimensional array to affect how it is displayed. The value \textit{number} scales the total number of nonempty array elements and is used by the PRINT statement to evaluate whether a two-dimensional array is “sparse” or “dense.” Tables that contain a single two-dimensional array are printed in list form if they are sparse and in matrix form if they are dense. Any nonnegative value can be assigned to \textit{number}. Specifying a value for the PMATRIX= option that is less than 1 causes the list form to be used in more cases, whereas specifying a value greater than 1 causes the matrix form to be used in more cases. If the value is 0, then the list form is always used. For more information, see the section “\textit{PRINT Statement}” on page 320. By default, PMATRIX=1.

\textbf{PRESOLVER=number | string}

specifies the type of presolve that the OPTMODEL presolver performs. Table 10.5 shows the valid values of this option.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|p{7cm}|}
\hline
\textit{number} & \textit{string} & \textit{Description} \\
\hline
–1 & AUTOMATIC & Applies presolver using default setting. \\
0 & NONE & Disables presolver. \\
1 & BASIC & Performs minimal processing, only substituting fixed variables and removing empty feasible constraints. \\
2 & MODERATE & Applies a higher level of presolve processing. \\
3 & AGGRESSIVE & Applies the highest level of presolve processing. \\
\hline
\end{tabular}
\caption{Values for the PRESOLVER= Option}
\end{table}

The OPTMODEL presolver tightens variable bounds and eliminates redundant constraints. In general, this tightening improves the performance of any solver. Higher levels of presolve processing allow more tightening and substitution passes, but might take more time to execute. The AUTOMATIC option is intermediate between the MODERATE and AGGRESSIVE levels.

\textbf{NOTE:} The OPTMODEL presolver is normally bypassed when PROC OPTMODEL uses the CLP, LP, QP, MILP, or network solver and when the SAVE MPS and SAVE QPS statements execute. The \textbf{FORCEPRESOLVE=} option enables the OPTMODEL presolver to be used with the CLP, LP, QP, and MILP solvers. PROC OPTMODEL always bypasses the OPTMODEL presolver when you specify certain solver options. For more information, see the chapter for the relevant solver in this book.

\textbf{PRESTOL=number}

provides a tolerance so that slightly infeasible constraints can be eliminated by the OPTMODEL presolver. If the magnitude of the infeasibility is no greater than \textit{num}(|X| + 1), where \textit{X} is the value of the original bound, then the empty constraint is removed from the presolved problem. OPTMODEL’s presolver does not print messages about infeasible constraints and variable bounds when the infeasibility is within the PRESTOL tolerance. The value of PRESTOL can range between 0 and 0.1; the default value is 1E–12.
PRINTLEVEL=number
controls the level of listing output during a SOLVE or COFOR command. The Output Delivery System (ODS) tables printed at each level are listed in Table 10.6. Some solvers can produce additional tables; see the individual solver chapters for more information.

<table>
<thead>
<tr>
<th>number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Disables all tables</td>
</tr>
<tr>
<td>1</td>
<td>Prints Problem Summary and Solution Summary</td>
</tr>
<tr>
<td>2</td>
<td>Prints Problem Summary, Solution Summary, Methods of Derivative Computation (for NLP solvers), Solver Options, Optimization Statistics, Timing, and solver-specific ODS tables</td>
</tr>
</tbody>
</table>

For more information about the ODS tables produced by PROC OPTMODEL, see the section “ODS Table and Variable Names” on page 371.

PWIDTH=number
sets the width used by the PRINT statement to display numeric columns when no format is specified. The smallest value number can take is the value of the PDIGITS= option plus 7; the largest value number can take is 16. The default value is equal to the value of the PDIGITS= option plus 7.

SESSREF=string | “string”
specifies the name of the CAS session to use for distributed computing. The string specifies a session name defined by the SAS CAS statement. You must specify either the SESSREF= or SESSUUID= option in order to use distributed computing.

SESSUUID=“string”
specifies the universal unique identifier (UUID) of the CAS session to use for distributed computing. The string specifies the UUID text. The UUID for a CAS session can be determined through various methods, such as using the LISTSESSIONS option of the SAS CAS statement. You must specify either the SESSUUID= or SESSREF= option in order to use distributed computing.

VARFUZZ=number
specifies the smallest difference that is permitted by the OPTMODEL presolver between the upper and lower bounds of an unfixed variable. If the difference is smaller than number, then the variable is fixed to the average of the upper and lower bounds before it is presented to the solver. Any nonnegative value can be assigned to number; the default value is 0.

**Declaration Statements**

The declaration statements define the parameters, variables, constraints, and objectives that describe a PROC OPTMODEL optimization model. Declarations in the PROC OPTMODEL input are saved for later use. Unlike programming statements, declarations cannot be nested in other statements. Declaration statements are terminated by a semicolon.
Many declaration attributes, such as variable bounds, are defined using expressions. Expressions in declarations are handled symbolically and are resolved as needed. In particular, expressions are generally reevaluated when one of the parameter values they use has been changed.

**CONSTRAINT Declaration**

```plaintext
CONSTRAINT constraint [ , . . . constraint ] ;

CON constraint [ , . . . constraint ] ;
```

The constraint declaration defines one or more constraints on expressions in terms of the optimization variables. You can specify multiple constraint declaration statements.

Constraints can have an upper bound, a lower bound, or both bounds. The allowed forms are as follows:

- `[ name [ { index-set } ] : ] expression = expression [ SUFFIXES=( values ) ]` declares an equality constraint or, when an `index-set` is specified, a family of equality constraints. The solver attempts to assign values to the optimization variables to make the two expressions equal.

- `[ name [ { index-set } ] : ] expression =~ expression [ SUFFIXES=( values ) ]` declares a disequality constraint or, when an `index-set` is specified, a family of disequality constraints. The solver attempts to assign values to the optimization variables to make the two expressions unequal. The CLP solver must be used with this type of constraint.

- `[ name [ { index-set } ] : ] expression relation expression [ SUFFIXES=( values ) ]` declares an inequality constraint that has a single upper or lower bound. `index-set` declares a family of inequality constraints. `relation` is the `<=`, `<`, `>=`, or `>` operator. When `relation` is the `<=` operator, the solver tries to assign optimization variable values so that the value of the left expression is less than or equal to the value of the right expression. When `relation` is the `<` operator, the solver tries to assign optimization variable values so that the value of the left expression is less than the value of the right expression. When `relation` is the `>=` operator, the solver tries to assign optimization variable values so that the value of the left expression is greater than or equal to the value of the right expression. When `relation` is the `>` operator, the solver tries to assign optimization variable values so that the value of the left expression is greater than the value of the right expression. The CLP solver must be used when the `<` or `>` operator is specified.

- `[ name [ { index-set } ] : ] bound relation body relation bound [ SUFFIXES=( values ) ]` declares an inequality constraint that is bounded on both sides, called a range constraint. `index-set` declares a family of range constraints. `relation` is the `<=`, `<`, `>=`, or `>` operator. Both `relation` operators must match in direction. If the `<=` or `<` operator is used in the first position, then a `<=` or `<` operator must be used in the second position. If the `>=` or `>` operator is used in the first position, then a `>=` or `>` operator must be used in the second position. The first `bound` expression defines the lower bound (if the `<=` or `<` operator is used) or the upper bound (if the `>=` or `>` operator is used). The second `bound` defines the upper bound (if the `<=` or `<` operator is used) or the lower bound (if the `>=` or `>` operator is used). The solver tries to assign optimization variables so that the value of the `body` expression is in the range between the upper and lower bounds. The CLP solver must be used when the `<` or `>` operator is specified.

- `[ name [ { index-set } ] : ] predicate [ SUFFIXES=( values ) ]` declares a predicate constraint for the CLP solver. See the section “Predicates” on
Chapter 10: The OPTMODEL Procedure


**NOTE:** You can use the alternate forms from Table 10.10 for the relational operators.

name defines the name for the constraint. Use the name to reference constraint attributes, such as the bounds, elsewhere in the PROC OPTMODEL model. If no name is provided, then a default name is created of the form \_ACON\_\[n\], where \(n\) is an integer. See the section “Constraints” on page 376 for more information.

The SUFFIXES=() clause enables you to specify certain auxiliary values in the constraint declaration. (For more information, see the section “Suffixes” on page 380.) Dummy parameters that you declare in an index-set specification after the constraint name can be used in value expressions.

Here is a simple example that defines a constraint with a lower bound:

```plaintext
proc optmodel;
  var x, y;
  number low;
  con a: x+y >= low;
```

The following example adds an upper bound:

```plaintext
var x, y;
number low;
con a: low <= x+y <= low+10;
```

Indexed families of constraints can be defined by specifying an index-set after the name. Any dummy parameters that are declared in the index-set can be referenced in the expressions that define the constraint. A particular member of an indexed family can be specified by using an identifier-expression with a bracketed index list, in the same fashion as array parameters and variables. For example, the following statements create an indexed family of constraints named incr:

```plaintext
proc optmodel;
  number n;
  var x{1..n} /* require nondecreasing x values */
  con incr{i in 1..n-1}: x[i+1] >= x[i];
```

The CON statement in the example creates constraints incr[1] through incr[n-1].

Constraint expressions cannot be defined using functions that return different values each time they are called. See the section “Indexing” on page 340 for details.

**IMPVAR Declaration**

**IMPVAR** impvar-decl [ , . . . impvar-decl ]

The IMPVAR declaration specifies one or more names that refer to optimization expressions in the model. The declared name is called an implicit variable. An implicit variable is useful for structuring models so that complex expressions do not need to be repeated each time they are used. The value of an implicit variable needs to be computed only once instead of at each place where the original expression is used, which helps reduce computational overhead. Implicit variables are evaluated without intervention from the solver.

Multiple IMPVAR declarations are allowed. The names of implicit variables must be distinct from other model declarations, such as variables and constraints. Implicit variables can be used in model expressions in
the same places where ordinary variables are allowed. When the defining expression does not depend on optimization variables, you can also use implicit variables in constant expressions.

This is the syntax for an `impvar-decl`:

\[ \text{name} \{ \text{index-set} \} = \text{expression} \]

Each `impvar-decl` declares a name for an implicit variable. The name can be followed by an `index-set` specification to declare a family of implicit variables. The `expression` that the name refers to follows. Dummy parameters that are declared in the `index-set` specification can be used in the expression. The `expression` can refer to other model components, including variables, the current implicit variable, and other implicit variables.

As an example, in the following model statements the implicit variable `total_weight` is used in multiple constraints to set a limit on various product quantities, represented by locations in array `x`:

```plaintext
impvar total_weight = sum{p in PRODUCTS} Weight[p]*x[p];

con prod1_limit: Weight['Prod1'] * x['Prod1'] <= 0.3 * total_weight;
con prod2_limit: Weight['Prod2'] * x['Prod2'] <= 0.25 * total_weight;
```

**MAX and MIN Objective Declarations**

\[ \text{MAX name} \{ \text{index-set} \} = \text{expression} [\text{SUFFIXES}=(\text{values})]; \]

\[ \text{MIN name} \{ \text{index-set} \} = \text{expression} [\text{SUFFIXES}=(\text{values})]; \]

The MAX or MIN declaration specifies an objective for the solver. The `name` names the objective function for later reference. When a non-array objective declaration is read, the declaration is added to the current problem. The solver maximizes an objective that is specified with the MAX keyword and minimizes an objective that is specified with the MIN keyword. An objective is not allowed to have the same name as a parameter or variable. Multiple objectives are permitted, but only the LSO solver can process more than one objective at a time. Other solvers consider only the last objective that is added. An array objective declaration resets the list of objectives for the current problem.

The `expression` specifies the numeric function to maximize or minimize in terms of the optimization variables. Specify an `index-set` to declare a family of objectives. Dummy parameters that you declare in the `index-set` specification can be used in the `expression` and `values` that follow it. The `SUFFIXES=()` clause enables you to specify certain auxiliary values in the objective declaration. For more information, see the section “Suffixes” on page 380.

Objectives can also be used as **implicit variables**. When used in an expression, an objective name refers to the current value of the named objective function. The value of an unsuffixed objective name can depend on the value of optimization variables, so objective names cannot be used in constant expressions such as variable bounds. You can reference objective names in objective or constraint expressions. For example, the following statements declare two objective names, `q` and `l`, which are immediately referred to in the objective declaration of `z` and the declarations of the constraints:

```plaintext
proc optmodel;
  var x, y;
  min q=(x+y)**2;
  max l=x+2*y;
  min z=q+l;
  con c1: q<=4;
  con c2: l>=2;
```
Objectives cannot be defined using functions that return different values each time they are called. See the section “Indexing” on page 340 for details.

**NUMBER, STRING, and SET Parameter Declarations**

```
NUMBER parameter-decl [ , . . . parameter-decl ] ;
STRING parameter-decl [ , . . . parameter-decl ] ;
SET [ < scalar-type, . . . scalar-type > ] parameter-decl [ , . . . parameter-decl ] ;
```

Parameters provide names for constants. Parameters are declared by specifying the parameter type followed by a list of parameter names. Declarations of parameters that have NUMBER or STRING types start with a `scalar-type` specification:

```
NUMBER | NUM ;
STRING | STR ;
```

The NUM and STR keywords are abbreviations for the NUMBER and STRING keywords, respectively.

The declaration of a parameter that has the set type begins with a `set-type` specification:

```
SET [ < scalar-type, . . . scalar-type > ] ;
```

In a `set-type` declaration, the SET keyword is followed by a list of `scalar-type` items that specify the member type. A set with scalar members is specified with a single `scalar-type` item. A set with tuple members has a `scalar-type` item for each tuple element. The `scalar-type` items specify the types of the elements at each tuple position.

If the SET keyword is not followed by a list of `scalar-type` items, then the set type is determined from the type of the initialization expression. The declared type defaults to SET<NUMBER> if no initialization expression is given or if the expression type cannot be determined.

For any parameter type, the type declaration is followed by a list of `parameter-decl` items that specify the names of the parameters to declare. In a `parameter-decl` item the parameter name can be followed by an optional index specification and any necessary options, as follows:

```
name [ { index-set } ] [ parameter-options ]
```

The parameter `name` and `index-set` can be followed by a list of `parameter-options`. Dummy parameters declared in the `index-set` can be used in the `parameter-options`. The parameter options can be specified with the following forms:

```
= expression

specifies an default value that is used when a parameter value is required but no other value has been supplied. For example:
```

```
PROC OPTMODEL evaluates the expression for each parameter location the first time the parameter needs to be resolved. The expression is not used when the parameter already has a value.

\begin{verbatim}
number n init 1;
set s init {'a', 'b', 'c'};
\end{verbatim}

|= initializers\] provides a compact means to define the values for an array, in which each array location value can be individually specified by the initializers.

INIT [ initializers ] provides a compact means to define multiple default values for an array. Each array location value can be individually specified by the initializers. With this option the array values can still be updated outside the declaration.

The =expression parameter option defines a parameter value by using a formula. The formula can refer to other parameters. The parameter value is updated when the referenced parameters change. The following example shows the effects of the update:

\begin{verbatim}
proc optmodel;
number n;
set<number> s = 1..n;
number a(s);
n = 3;
a[1] = 2; /* OK */
a[7] = 19; /* error, 7 is not in s */
n = 10;
a[7] = 19; /* OK now */
\end{verbatim}

In the preceding example the value of set s is resolved for each use of array a that has an index. For the first use of a[7], the value 7 is not a member of the set s. However, the value 7 is a member of s at the second use of a[7].

The INIT expression parameter option specifies a default value for a parameter. The following example shows the usage of this option:

\begin{verbatim}
proc optmodel;
num a(i in 1..2) init i**2;
a[1] = 2;
p\_put a[*]=;
\end{verbatim}

When the value of a parameter is needed but no other value has been supplied, the default value specified by INIT expression is used, as shown in Figure 10.5.

![Figure 10.5 INIT Option: Output](image)

a[1]=2 a[2]=4

**NOTE:** Parameter values can also be read from files or specified with assignment statements. However, the value of a parameter that is assigned with the =expression or =initializers forms can be changed only by modifying the parameters used in the defining expressions. Parameter values specified by the INIT option can be reassigned freely.
Chapter 10: The OPTMODEL Procedure

Initializing Arrays
Arrays can be initialized with the = [initializers] or INIT [initializers] forms. These forms are convenient when array location values need to be individually specified. The forms behave the same way, except that the INIT [initializers] form allows the array values to be modified after the declaration. These forms of initialization are used in the following statements:

```
proc optmodel;
  number a{1..3} = [5 4 7];
  number b{1..3} INIT [5 4 7];
  put a[*]=;
  b[1] = 1;
  put b[*]=;
```

Each array location receives a different value, as shown in Figure 10.6. The displayed values for \( b \) are a combination of the default values from the declaration and the assigned value in the statements.

**Figure 10.6 Array Initialization**

```
```

Each initializer takes the following form:

```
[[ index ]] value
```

The value specifies the value of an array location and can be a numeric or string constant, a set literal, or an expression enclosed in parentheses.

In array initializers, string constants can be specified using quoted strings. When the string text follows the rules for a SAS name, the text can also be specified without quotation marks. String constants that begin with a digit, contain blanks, or contain other special characters must be specified with a quoted string.

As an example, the following statements define an array parameter that could be used to map numeric days of the week to text strings:

```
proc optmodel;
  string dn{1..5} = [Monday Tuesday Wednesday Thursday Friday];
```

The optional index in square brackets specifies the index of the array location to initialize. The index specifies one or more numeric or string subscripts. The subscripts allow the same syntactic forms as the value items. Commas can be used to separate index subscripts. For example, location \( a[1,'abc'] \) of an array \( a \) could be specified with the index \([1 abc]\). The following example initializes just the diagonal locations in a square array:

```
proc optmodel;
  number m{1..3,1..3} = [[1 1] 0.1 [2 2] 0.2 [3 3] 0.3];
```

An index does not need to specify all the subscripts of an array location. If the index begins with a comma, then only the rightmost subscripts of the index need to be specified. The preceding subscripts are supplied from the index that was used by the preceding initializer. This can simplify the initialization of arrays that are indexed by multiple subscripts. For example, you can add new entries to the matrix of the previous example by using the following statements:

```
proc optmodel;
  number m{1..3,1..3} = [[1 1] 0.1 [2 2] 0.2 [3 3] 0.3];
```
proc optmodel;
  number m{1..3,1..3} = [[1 1] 0.1 [,3] 1
    [2 2] 0.2 [,3] 2
    [3 3] 0.3];

The spacing shows the layout of the example array. The previous example was updated by initializing two more values at m[1,3] and m[2,3].

If an index is omitted, then the next location in the order of the array’s index set is initialized. If the index set has multiple index-set-items, then the rightmost indices are updated before indices to the left are updated. At the beginning of the initializer list, the rightmost index is the first member of the index set. The index set must use a range expression to avoid unpredictable results when an index value is omitted.

The initializers can be followed by commas. The use of commas has no effect on the initialization. The comma can be used to clarify layout. For example, the comma could separate rows in a matrix.

Not every array location needs to be initialized. The locations without an explicit initializer are set to missing for numeric arrays, set to an empty string for string arrays, and set to an empty set for set arrays. You can also specify a different default value in the last initializer by using an asterisk (*) in place of the index. For example:

  proc optmodel;
  /* initialize to [2 7 30 40] */
  number a{i in 1..4} = [2 7 [*] (i\*10)];

  NOTE: An array location must not be initialized more than once during the processing of the initializer list.

**PROBLEM Declaration**

```
PROBLEM  name [{ index-set }] [ FROM problem-id ] [ INCLUDE problem-items ] [ SUFFIXES=( values ) ];
```

Problems are declared with the PROBLEM declaration. Problem declarations track objectives, a set of included variables and constraints, and some status information that is associated with the variables and constraints. The problem name can optionally be followed by an index-set to create a family of problems. When a problem is first used (via the USE PROBLEM statement), the specifications from the optional FROM and INCLUDE clauses create the initial set of included variables, constraints, and the problem objectives. An empty problem is created if neither clause is specified. The clauses are applied only when the problem is first used with the USE PROBLEM statement.

The FROM clause specifies an existing problem from which to copy the set of included symbols. The problem-id is an identifier expression. The dropped and fixed status for these symbols in the specified problem is also copied.

The INCLUDE clause specifies a list of variables, constraints, and objectives to include in the problem. These items are included with default status (unfixed and undropped) which overrides the status from the FROM clause, if it exists. Each item is specified with one of the following forms:

```
identifier-expression
```

includes the specified items in the problem. The identifier-expression can be a symbol name or an array symbol with explicit index. If an array symbol is used without an index, then all array elements are included.
Chapter 10: The OPTMODEL Procedure

{ index-set } identifier-expression
includes the specified subset of items in the problem. The item specified by the identifier-expression is added to the problem for each member of the index-set. The dummy parameters from the index-set can be used in the indexing of the identifier-expression. If the identifier-expression is an array symbol without indexing, then the index-set provides the indices for the included locations.

You can use the FROM and INCLUDE clauses to designate the objectives for a problem. The objectives are copied from the problem that is designated by the FROM clause, if present. Then the INCLUDE clause, if any, is applied. The LSO solver can process multiple objectives. Other solvers use only the last objective that is specified.

The SUFFIXES=() clause enables you to specify certain auxiliary values in the problem declaration. (For more information, see the section “Suffixes” on page 380.) Dummy parameters that you declare in an index-set specification for the problem name can be used in the value expressions.

The following statements declare some problems with a variable x and different objectives to illustrate some of the ways of including model components. Note that the statements use the predeclared problem _START_ to avoid resetting the objective in prob2 when the objective z3 is declared.

```plaintext
proc optmodel;
   problem prob1;
   use problem prob1;
   var x >= 0;    /* included in prob1 */
   min z1 = (x-1)**2; /* included in prob1 */
   expand;        /* prob1 contains x, z1 */

   problem prob2 from prob1;
   use problem prob2; /* includes x, z1 */
   min z2 = (x-2)**2; /* included in prob2 */
   expand;        /* prob2 contains x, z1, z2 */
                  /* note z1 is not expanded */

   use problem _start_; /* don't modify prob2 */
   min z3 = (x-3)**2;
   problem prob3 include x z3;
   use problem prob3;
   expand;        /* prob3 contains x, z3 */

See the section “Multiple Subproblems” on page 397 for more details about problem processing.

VAR Declaration

VAR var-decl [, . . . var-decl ];

The VAR statement declares one or more optimization variables. Multiple VAR statements are permitted. A variable is not allowed to have the same name as a parameter or constraint.

Each var-decl specifies a variable name. The name can be followed by an array index-set specification and then variable options. Dummy parameters declared in the index set specification can be used in the following variable options.

Here is the syntax for a var-decl:
name | { index-set } | [ var-options ] | [ SUFFIXES=( values ) ]

For example, the following statements declare a group of 100 variables, \texttt{x[1]}-\texttt{x[100]}:

```
proc optmodel;
  var x{1..100};
```

Here are the available variable options:

- **INIT** \texttt{expression}:
  sets an initial value for the variable. The expression is used only the first time the value is required. If no initial value is specified, then 0 is used by default.

- **>=** \texttt{expression}:
  sets a lower bound for the variable value. The default lower bound is \(-\infty\), or 0 if the BINARY option is used.

- **<=** \texttt{expression}:
  sets an upper bound for the variable value. The default upper bound is \(\infty\), or 1 if the BINARY option is used.

- **INTEGER**:
  requests that the solver assign the variable an integer value.

- **BINARY**:
  requests that the solver assign the variable an integer value within default bounds of 0 to 1.

Dummy parameters that you declare in an \texttt{index-set} specification for the variable \texttt{name} can be used in the variable declaration expressions, including the SUFFIXES=() clause. The SUFFIXES=() clause enables you to specify certain auxiliary values in the declaration. For more information, see the section “Suffixes” on page 380.

For example, the following statements declare a variable that has an initial value of 0.5. The variable is bounded by 0 and 1:

```
proc optmodel;
  var x init 0.5 >= 0 <= 1;
```

The values of the bounds can be determined later by using suffixed references to the variable. For example, the upper bound for variable \texttt{x} can be referred to as \texttt{x.ub}. In addition, the bounds options can be overridden by explicit assignment to the suffixed variable name. Suffixes are described further in the section “Suffixes” on page 380. Note that the bounds for integer and binary variables are rounded to integers according to the value that you specify in the PROC OPTMODEL option INTFUZZ=.

When used in an expression, an unsuffixed variable name refers to the current value of the variable. Unsuffixed variables are not allowed in the expressions for options that define variable bounds or initial values. Such expressions have values that must be fixed during execution of the solver.

---

**Programming Statements**

PROC OPTMODEL supports several programming statements. You can perform various actions with these statements, such as reading or writing data sets, setting parameter values, generating text output, or invoking a solver.
Statements are read from the input and are executed immediately when complete. Certain statements can contain one or more substatements. The execution of substatements is held until the statements that contain them are submitted. Parameter values that are used by expressions in programming statements are resolved when the statement is executed; this resolution might cause errors to be detected. For example, the use of undefined parameters is detected during resolution of the symbolic expressions from declarations.

A statement is terminated by a semicolon. The positions at which semicolons are placed are shown explicitly in the following statement syntax descriptions.

The programming statements can be grouped into the categories shown in Table 10.7.

### Table 10.7 Types of Programming Statements in PROC OPTMODEL

<table>
<thead>
<tr>
<th>Control</th>
<th>Looping</th>
<th>General</th>
<th>Input/Output</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>DO</td>
<td>COFOR</td>
<td>Assignment</td>
<td>CLOSEFILE</td>
<td>DROP</td>
</tr>
<tr>
<td>IF</td>
<td>CONTINUE</td>
<td>CALL</td>
<td>CREATE DATA</td>
<td>EXPAND</td>
</tr>
<tr>
<td>Null (;)DO Iterative</td>
<td>PROFILE</td>
<td>FILE</td>
<td>FIX</td>
<td></td>
</tr>
<tr>
<td>QUIT</td>
<td>DO UNTIL</td>
<td>RESET OPTIONS</td>
<td>PRINT</td>
<td>RESTORE</td>
</tr>
<tr>
<td>STOP</td>
<td>DO WHILE</td>
<td>SUBMIT</td>
<td>PUT</td>
<td>SOLVE</td>
</tr>
<tr>
<td>FOR</td>
<td>LEAVE</td>
<td>SAVE MPS</td>
<td>READ DATA</td>
<td>UNFIX</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SAVE QPS</td>
<td></td>
<td>USE PROBLEM</td>
</tr>
</tbody>
</table>

### Assignment Statement

`identifier-expression = expression ;`

The assignment statement assigns a variable or parameter value. The type of the target `identifier-expression` must match the type of the right-hand-side expression.

For example, the following statements set the current value for variable `x` to 3:

```plaintext
proc optmodel;
  var x;
  x = 3;
```

**NOTE:** Parameters that were declared with the equal sign (=) initialization forms must not be reassigned a value with an assignment statement. If this occurs, PROC OPTMODEL reports an error.

### CALL Statement

`CALL name (argument-1 [, ... argument-n]) ;`

The CALL statement invokes the named library subroutine. The values that are determined for each argument expression are passed to the subroutine when the subroutine is invoked. The subroutine can update the values of PROC OPTMODEL parameters and variables when an argument is an `identifier-expression` (see the section “Identifier Expressions” on page 347). For example, the following statements set the parameter array `a` to a random permutation of 1 to 4:
**Programming Statements**

```plaintext
proc optmodel;
  number a{i in 1..4} init i;
  number seed init -1;
  call ranperm(seed, a[1], a[2], a[3], a[4]);
```

**NOTE:** The maximum length of the string value returned from an output argument is equal to the character length of the argument before the call. An undefined STRING parameter that is used as an output argument has a character length of 8.

For a list of CALL routines, see *SAS Functions and CALL Routines: Reference*. You can also call subroutines that are compiled by the FCMP procedure. For more information, see the section “FCMP Routines” on page 403.

**CLOSEFILE Statement**

```plaintext
CLOSEFILE file-specifications;
```

The CLOSEFILE statement closes files that were opened by the FILE statement. Each file is specified by a logical name, a physical filename in quotation marks, or an expression enclosed in parentheses that evaluates to a physical filename. See the section “FILE Statement” on page 316 for more information about file specifications.

The following example shows how the CLOSEFILE statement is used with a logical filename:

```plaintext
filename greet 'hello.txt';
proc optmodel;
  file greet;
  put 'Hi!';
  closefile greet;
```

Generally you must close a file with a CLOSEFILE statement before external programs can access the file. However, any open files are automatically closed when PROC OPTMODEL terminates.

**COFOR Statement**

```plaintext
COFOR { index-set } statement;
```

The COFOR statement executes its `statement` for each member of the specified `index-set`, similar to how the `FOR` statement executes. However, in a COFOR statement, PROC OPTMODEL can execute the SOLVE statement concurrently with other statements. The execution of the COFOR substatement is interleaved between loop iterations so that other iterations can be processed while an iteration waits for a SOLVE statement to complete. Multiple solvers can run concurrently. This interleaving is managed so that in many cases a FOR loop can be replaced by a COFOR loop to achieve concurrency with minimal or no other changes to the code.

The following code shows a simple example:

```plaintext
proc optmodel printlevel=0 sessref=mysess;
  var x {1..6} >= 0;
  minimize z = sum {j in 1..6} x[j];
```

cofor{i in 3..5} do;
    fix x[1]=i;
    solve;
    put i= x[1]= _solution_status_=;
end;

Figure 10.7 shows the PROC OPTMODEL output. The order of the output from different iterations can vary between runs, depending on the order in which the SOLVE statements complete. Because the SESSREF= option is used, the solvers execute in the distributed computing environment. A FOR statement could have been used instead of COFOR; the FOR statement would produce a consistent output order but only one solver would execute at a time. Note that because the solver execution in this example is trivial, the benefits from concurrency are limited.

Figure 10.7 A Simple COFOR Loop

i=4 x[1]=4 _SOLUTION_STATUS_=OPTIMAL
i=5 x[1]=5 _SOLUTION_STATUS_=INFEASIBLE
i=3 x[1]=3 _SOLUTION_STATUS_=OPTIMAL

A COFOR statement can contain other control and looping statements, including nested COFOR loops. The maximum number of threads that can be used is controlled by the PROC OPTMODEL and SAS options that are in effect when the outermost COFOR loop is entered, as described in the section “Threaded and Distributed Processing” on page 407. The outermost COFOR statement allocates threads for execution on the computer that is running PROC OPTMODEL. When you use a SESSREF= or SESSUUID= option to request distributed computing, the outermost COFOR statement also creates a distributed execution environment. Solvers within the COFOR loop can then run remotely in single-machine mode on the compute nodes (as shown in the solver output).

The COFOR statement supports simultaneous processing of several SOLVE statements. Processing proceeds through the iteration body statements as it would through a FOR loop until a SOLVE statement that uses the CLP, LP, LSO, MILP, network, NLP, or QP solver is executed. After the problem is generated, the solver starts processing in a background thread (or remote computing node in the distributed case) and the COFOR loop switches execution to another iteration of the loop, assuming enough threads and iterations are available. (Note that you need at least two threads on the computer that is running the COFOR loop to enable overlap of statement execution with solver execution.) Execution could switch to an existing iteration where the solver has completed. Alternatively, a new iteration of a COFOR loop could be started. All output from an iteration, except within a SUBMIT block, is displayed together after the iteration has completed. Output from a SUBMIT block is displayed as the block is executed.

You can change the NTHREADS= option value within a COFOR loop by using the RESET OPTIONS statement. The new value affects SOLVE statements that are executed subsequently in the same iteration but not those in other iterations. When a COFOR statement is running in distributed mode, the default value of the NTHREADS= option is the number of threads per compute node, as determined when the outermost COFOR loop starts. Otherwise the default value of the NTHREADS= option is 1. Executing SOLVE statements in the background by using a single thread usually provides the best performance on a single computer.
The order in which the solvers complete is unpredictable. So it is usually not useful for a problem that is solved within an iteration to depend on the results of SOLVE statements that are executed in other iterations of the COFOR loop. It is advisable to limit global parameter updates to operations where order is not important, such as accumulating counts, sums, or unions or writing mutually exclusive subsets of an array. It is possible to execute multiple SOLVE statements within a loop iteration, and subsequent solver invocations within an iteration can use results from prior solvers in the same iteration.

In many cases, a COFOR loop iteration solves a specialized version of a common problem structure. This requires it to modify problem attributes that are also used in other iterations, such as coefficient values or the fixed status of variables. Changes to problem attributes are not made visible to other iterations of a COFOR loop in order to avoid confusing behavior due to interleaved execution. For example, the value printed for \( x[1] \) in Figure 10.7 is the local value for the iteration, not the most recent global value. Changes to these attributes create or update a copy of the value that is local to the iteration. These attribute values along with the local dummy parameters provide a local context for the iteration.

The following problem attributes are automatically made local to the modifying iteration when they are changed within a COFOR loop:

- the current problem, selected by USE PROBLEM
- the value of variables and their suffix values
- the fixed status of variables
- the constraint suffix values
- the dropped status of constraints
- the .LABEL suffix
- NUMBER, STRING, and SET parameters that determine values that are used in the bounds or body expressions of problem declarations (CONSTRAINT, IMPVAR, MIN, MAX, or VAR)
- NUMBER, STRING, and SET parameters that determine values that are used in solver arguments within the same outermost COFOR loop
- the predeclared string parameters _SOLVER_OPTIONS_ and _solver_OPTIONS_ (for each solver)

To illustrate these rules, consider the following code, which uses the NLP solver to solve a MINLP portfolio optimization problem by selecting random subsets of the assets to optimize:

```plaintext
proc optmodel printlevel=0;
   /* assets and related parameters */
   set ASSETS;
   num return {ASSETS};
   num cov {ASSETS, ASSETS} init 0;
   read data means into ASSETS=[_n_] return;
   read data covdata into [asset1 asset2] cov cov[asset2,asset1]=cov;
   num riskLimit init 0.00025;
   num minThreshold init 0.1;
   num numTrials = 10;
   set TRIALS = 1..numTrials;
   /* number of random trials */
   set TRIALS = 1..numTrials;
```
/* declare NLP problem for fixed set of assets */
set ASSETS_THIS;
var AssetPropVar {ASSETS} >= minThreshold <= 1;
max ExpectedReturn = sum {i in ASSETS} return[i] * AssetPropVar[i];
con RiskBound:
    sum {i in ASSETS_THIS, j in ASSETS_THIS}
con TotalPortfolio:
    sum {asset in ASSETS} AssetPropVar[asset] = 1;

/* parameters to track best solution */
um infinity = constant('BIG');
um best_objective init -infinity;
set INCUMBENT;

/* iterate over trials */
num start {TRIALS};
num finish {TRIALS};
num overall_start;
overall_start = time();
call streaminit(1);
cofor {trial in TRIALS} do;
    start[trial] = time() - overall_start;
    put;
    put trial=;
    ASSETS_THIS = {i in ASSETS: rand('UNIFORM') < 0.5};
    put ASSETS_THIS=;
    for {i in ASSETS diff ASSETS_THIS}
        fix AssetPropVar[i] = 0;
    solve with NLP / logfreq=0;
    put _solution_status_=;
    if _solution_status_ ne 'INFEASIBLE' then do;
        if best_objective < ExpectedReturn then do;
            best_objective = ExpectedReturn;
            INCUMBENT = ASSETS_THIS;
        end;
    end;
    finish[trial] = time() - overall_start;
end;

put best_objective= INCUMBENT=;
create data plotdata from [trial] start finish;

proc sgplot data=plotdata;
    highlow y=trial low=start high=finish / type=bar x2axis;
yaxis reverse;
x2axis label="seconds";
run;

All the COFOR loop iterations use the same problem, _START_. However, the changes to the problem are local to the iteration that makes them. For example, the FIX statement does not affect variables in other iterations. The value of the ASSETS_THIS parameter is used by the RiskBound constraint, so the change to it is local. Because AssetPropVar is a VAR, the changes to its value are also local.
On the other hand, the values of the best_objective and INCUMBENT parameters do not affect any problem declarations. Therefore, their global values are used, enabling the code in the COFOR loop to select and save the best result. Similarly, the start and finish parameters are not used in the problem and allow the overlapping of iterations to be illustrated. Figure 10.8 from the SGPlot procedure shows how the iterations have overlapped execution times.

**Figure 10.8** Overlapped COFOR Iterations

Changes to problem attributes from completed iterations are made visible after the loop is finished. They appear in the context that contained the COFOR statement. If multiple iterations modify the same problem attribute value, then the value from the iteration that completed last is the one made visible.

The LEAVE statement can be used to terminate execution of a COFOR loop. This completes the current iteration of the COFOR loop. The currently active solvers for the COFOR loop are terminated, and the output of the incomplete iterations is discarded. The CONTINUE statement within a COFOR loop can also be used to complete the current iteration, but it has no effect on other iterations.

Using the LEAVE statement to terminate is useful, for example, when a sufficiently good solution is found for a problem. The preceding code has been modified as follows to keep generating solutions until a time limit is reached. The code sets a time limit and then executes the LEAVE statement to stop processing when the limit is exceeded. The COFOR loop uses a very large iteration range to allow it to run indefinitely.
proc optmodel printlevel=0;
set ASSETS;
num return {ASSETS};
um num cov {ASSETS, ASSETS} init 0;
read data means into ASSETS=[_n_] return;
read data covdata into [asset1 asset2] cov cov[asset2,asset1]=cov;
um riskLimit init 0.00025;
um minThreshold init 0.1;

/* declare NLP problem for fixed set of assets */
set ASSETS_THIS;
var AssetPropVar {ASSETS} >= minThreshold <= 1;
max ExpectedReturn = sum {i in ASSETS} return[i] * AssetPropVar[i];
con RiskBound:
    sum {i in ASSETS_THIS, j in ASSETS_THIS}
con TotalPortfolio:
    sum {asset in ASSETS} AssetPropVar[asset] = 1;

num infinity = constant('BIG');
um best_objective init -infinity;
set INCUMBENT;

/* run for 30 seconds */
num last_time;
last_time = time() + 30;
um n_trials init 0;
call streaminit(1);
cofor {trial in 1..1e9} do;
    put;
    put trial=;
    ASSETS_THIS = {i in ASSETS: rand('UNIFORM') < 0.5};
    put ASSETS_THIS=;
    for {i in ASSETS diff ASSETS_THIS} fix AssetPropVar[i] = 0;
solve with NLP / logfreq=0;
    put _solution_status_=;
    if _solution_status_ ne 'INFEASIBLE' then do;
        if best_objective < ExpectedReturn then do;
            best_objective = ExpectedReturn;
            INCUMBENT = ASSETS_THIS;
        end;
    end;
n_trials = n_trials + 1;
    if time() >= last_time then leave;
end;

put n_trials=;
put best_objective= INCUMBENT=;
quit;
CONTINUE Statement

CONTINUE;

The CONTINUE statement terminates the current iteration of the loop statement (iterative DO, DO UNTIL, DO WHILE, FOR, or COFOR) that immediately contains the CONTINUE statement. Execution resumes at the start of the loop after checking WHILE or UNTIL tests. The FOR, COFOR, or iterative DO loops apply new iteration values.

CREATE DATA Statement

CREATE DATA SAS-data-set FROM [[key-columns][=key-set]] columns;

The CREATE DATA statement creates a new SAS data set and copies data into it from PROC OPTMODEL parameters and variables. The CREATE DATA statement can create a data set with a single observation or a data set with observations for every location in one or more arrays. The data set is closed after the execution of the CREATE DATA statement.

NOTE: Currently, the source data are always located on the client.

The arguments to the CREATE DATA statement are as follows:

- **SAS-data-set** specifies the output data set name and options. You can specify the data set name and options directly or as the string value of an expression enclosed in parentheses.

- **key-columns** declares index values and their corresponding data set variables. The values are used to index array locations in **columns**.

- **key-set** specifies a set of index values for the **key-columns**.

- **columns** specifies data set variables as well as the PROC OPTMODEL source data for the variables.

Each **column** or **key-column** defines output data set variables and a data source for a column. For example, the following statement generates the output SAS data set **resdata** from the PROC OPTMODEL array **opt**, which is indexed by the set **indset**:

create data resdata from [solns]=indset opt;

The output data set variable **solns** contains the index elements in **indset**.

Columns

Columns can have the following forms:

- **identifier-expression [ / options]** transfers data from the PROC OPTMODEL parameter or variable specified by the **identifier-expression**. The output data set variable has the same name as the **name** part of the **identifier-expression** (see the section “Identifier Expressions” on page 347). If the **identifier-expression** refers to an array, then the index can be omitted when it matches the **key-columns**. The **options** enable formats and labels to be associated with the data
set variable. See the section “Column Options” on page 307 for more information. The following example creates a data set with the variables m and n:

```plaintext
proc optmodel;
   number m = 7, n = 5;
   create data example from m n;
```

`name = expression / options` transfers the value of a PROC OPTMODEL expression to the output data set variable `name`. The `expression` is reevaluated for each observation. If the `expression` contains any operators or function calls, then it must be enclosed in parentheses. If the `expression` is an `identifier-expression` that refers to an array, then the index can be omitted if it matches the `key-columns`. The `options` enable formats and labels to be associated with the data set variable. See the section “Column Options” on page 307 for more information. The following example creates a data set with the variable `ratio`:

```plaintext
proc optmodel;
   number m = 7, n = 5;
   create data example from ratio=(m/n);
```

`COL(name-expression) = expression / options` transfers the value of a PROC OPTMODEL expression to the output data set variable named by the string expression `name-expression`. The PROC OPTMODEL expression is reevaluated for each observation. If this expression contains any operators or function calls, then it must be enclosed in parentheses. If the PROC OPTMODEL `expression` is an `identifier-expression` that refers to an array, then the index can be omitted if it matches the `key-columns`. The `options` enable formats and labels to be associated with the data set variable. See the section “Column Options” on page 307 for more information. The following example uses the COL expression to form the variable `s5`:

```plaintext
proc optmodel;
   number m = 7, n = 5;
   create data example from col("s"||n)=(m+n);
```

```
{ index-set } < columns >
```
performs the transfers by iterating each column specified by `<columns>` for each member of the `index-set`. If there are `n` columns and `m` index set members, then `n x m` columns are generated. The dummy parameters from the index set can be used in the columns to generate distinct output data set variable names in the iterated columns, using COL expressions. The columns are expanded when the CREATE DATA statement is executed, before any output is performed. This form of `columns` cannot be nested. In other words, the following form of `columns` is NOT allowed:

```
{ index-set } < { index-set } < columns > >
```

The following example demonstrates the use of the iterated `columns` form:
proc optmodel;
  set<string> alph = {'a', 'b', 'c'};
  var x{1..3, alph} init 2;
  create data example from [i]=(1..3)
    {j in alph}<col("x"||j)=x[i,j]>
  ;

The data set created by these statements is shown in Figure 10.9.

**Figure 10.9** CREATE DATA with COL Expression

<table>
<thead>
<tr>
<th></th>
<th>xA</th>
<th>xB</th>
<th>xc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

**NOTE:** When no *key-columns* are specified, the output data set has a single observation.

The following statements incorporate several of the preceding examples to create and print a data set by using PROC OPTMODEL parameters:

```plaintext
proc optmodel;
  number m = 7, n = 5;
  create data example from m n ratio=(m/n) col("s"||n)=(m+n);
  proc print;
  run;
```

The output from the PRINT procedure is shown in Figure 10.10.

**Figure 10.10** CREATE DATA for Single Observation

<table>
<thead>
<tr>
<th></th>
<th>m</th>
<th>n</th>
<th>ratio</th>
<th>s5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>5</td>
<td>1.4</td>
<td>12</td>
</tr>
</tbody>
</table>

**Column Options**

Each *column* or *key-column* that defines a data set variable can be followed by zero or more of the following modifiers:

- **FORMAT=** *format*.
  associates a format with the current column.

- **INFORMAT=** *informat*.
  associates an informat with the current column.

- **LABEL=** *label*.
  associates a label with the current column. The label can be specified by a quoted string or an expression in parentheses.

- **LENGTH=** *length*.
  specifies a length for the current column. The length can be specified by a numeric constant or a parenthesized expression. For character columns, the length specifies a
The character length is used to determine a byte length for the column so that any character string of the specified length can be represented in the output encoding. For example, specifying LENGTH=5 with UTF-8 encoding produces a byte length of 20, whereas a single-byte encoding produces a byte length of 5. For character columns, the minimum length is 1, and the maximum length depends on the data set engine. Output to fixed CHAR variables is limited to 32,767 bytes, but VARCHAR output can use up to $2^{31} - 1$ bytes. For numeric columns, the length specifies a byte length for the output variable. The range for numeric variables depends on the operating environment and has a minimum of 2 or 3.

**TRANSCODE=YES | NO**
specifies whether character variables can be transcoded. The default value is YES. See the TRANSCODE=option of the ATTRIB statement in *SAS DATA Step Statements: Reference* for more information.

The following statements demonstrate the use of column options, including the use of multiple options for a single column:

```
proc optmodel;
   num sq{i in 1..10} = i*i;
   create data squares from [i/format=hex2./length=3] sq/format=6.2;
proc print;
run;
```

The output from the PRINT procedure is shown in Figure 10.11.

**Figure 10.11** CREATE DATA with Column Options

<table>
<thead>
<tr>
<th>Obs</th>
<th>i</th>
<th>sq</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>01</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>02</td>
<td>4.00</td>
</tr>
<tr>
<td>3</td>
<td>03</td>
<td>9.00</td>
</tr>
<tr>
<td>4</td>
<td>04</td>
<td>16.00</td>
</tr>
<tr>
<td>5</td>
<td>05</td>
<td>25.00</td>
</tr>
<tr>
<td>6</td>
<td>06</td>
<td>36.00</td>
</tr>
<tr>
<td>7</td>
<td>07</td>
<td>49.00</td>
</tr>
<tr>
<td>8</td>
<td>08</td>
<td>64.00</td>
</tr>
<tr>
<td>9</td>
<td>09</td>
<td>81.00</td>
</tr>
<tr>
<td>10</td>
<td>0A</td>
<td>100.00</td>
</tr>
</tbody>
</table>

**Key Columns**

*Key-columns* declare index values that enable multiple observations to be written from array *columns*. An observation is created for each unique index value combination. The index values supply the index for array *columns* that do not have an explicit index.

*Key-columns* define the data set variables where the index value elements are written. They can also declare local dummy parameters for use in expressions in the *columns*. *Key-columns* are syntactically similar to *columns*, but are more restricted in form. The following forms of *key-columns* are allowed:
transfers an index element value to the data set variable \textit{name}. A local dummy parameter, \textit{name}, is declared to hold the index element value. The \textit{options} enable formats and labels to be associated with the data set variable. See the section “Column Options” on page 307 for more information.

\textbf{COL(name-expression)} \[=\] \textit{index-name} \[/\] \textit{options}

transfers an index element value to the data set variable named by the string-valued \textit{name-expression}. The argument \textit{index-name} optionally declares a local dummy parameter to hold the index element value. The \textit{options} enable formats and labels to be associated with the data set variable. See the section “Column Options” on page 307 for more information.

A \textit{key-set} in the CREATE DATA statement explicitly specifies the set of index values. You can specify \textit{key-set} as an \textit{identifier expression}, an \textit{index set expression}, a \textit{set constructor expression}, or an \textit{arbitrary set expression} enclosed in parentheses. When you specify \textit{key-set} as an index set expression, the \textit{index-set} dummy parameters remain visible in \textit{columns} and override dummy parameters that are declared in the \textit{key-columns} items with the same name.

The following statements create a data set from the PROC OPTMODEL parameter \(m\), a matrix whose only nonzero entries are located at (1, 1) and (4, 1):

\begin{verbatim}
proc optmodel;
  number m{1..5, 1..3} = [[1 1] 1 [4 1] 1 [*] 0];
  set ISET = setof{i in 1..2} i**2;
  create data example from [i j] = {i in ISET, {1, 2}} m;
  proc print data=example noobs;
  run;
\end{verbatim}

The dummy parameter \(i\) in the \textit{key-set} expression takes precedence over the dummy parameter \(i\) declared in the \textit{key-columns} item. The output from these statements is shown in Figure 10.12.

\textbf{Figure 10.12} CREATE: \textit{key-set} with \textit{SETOF} Aggregation Expression

\begin{tabular}{c c c}
  i & j & m \\
  1 & 1 & 1 \\
  1 & 2 & 0 \\
  4 & 1 & 1 \\
  4 & 2 & 0 \\
\end{tabular}

If no \textit{key-set} is specified, then the set of index values is formed from the union of the index sets of the implicitly indexed columns. The number of index elements for each implicitly indexed array must match the number of \textit{key-columns}. The type of each index element (string versus numeric) must match the element of the same position in other implicit indices.

The arrays for implicitly indexed columns in a CREATE DATA statement do not need to have identical index sets. A missing value is supplied for the value of an implicitly indexed array location when the implied index value is not in the array’s index set.

In the following statements, the \textit{key-set} is unspecified. The set of index values is \{1, 2, 3\}, which is the union of the index sets of \(x\) and \(y\). These index sets are not identical, so missing values are supplied when necessary.
The results of these statements are shown in Figure 10.13.

```plaintext
class proc optmodel;
  number x{1..2} init 2;
  var y{2..3} init 3;
  create data exdata from [keycol] x y;
end;

class proc print;
  run;
end;
```

**Figure 10.13** CREATE: Unspecified key-set

<table>
<thead>
<tr>
<th>Obs</th>
<th>keycol x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 2 .</td>
</tr>
<tr>
<td>2</td>
<td>2 2 3</td>
</tr>
<tr>
<td>3</td>
<td>3 . 3</td>
</tr>
</tbody>
</table>

The types of the output data set variables match the types of the source values. The output variable type for a key-columns value matches the corresponding element type in the index value tuple. A numeric element matches a numeric data set variable, and a string element matches a character variable. For regular columns, the source expression type determines the output data set variable type. A numeric expression produces a numeric variable, and a string expression produces a character variable. Character variables are stored in VARCHAR form when the data set engine supports them. Otherwise, character variables are stored in fixed CHAR form.

Lengths of character variables in the output data set are determined automatically when the LENGTH= column option is not specified. The length is set to accommodate the longest string value output in that column.

You can use the iterated columns form to output selected rows of multiple arrays, assigning a different data set variable to each column. For example, the following statements output the last two rows of the two-dimensional array, a, along with corresponding elements of the one-dimensional array, b:

```plaintext
class proc optmodel;
  num m = 3; /* number of rows/observations */
  num n = 4; /* number of columns in a */
  num a{i in 1..m, j in 1..n} = i*j; /* compute a */
  num b{i in 1..m} = i**2; /* compute b */
  set<num> subset = 2..m; /* used to omit first row */
  create data out
    from [i]=subset {j in 1..n}<col("a"||j)=a[i,j]> b;
end;
```

The preceding statements create a data set out, which has \( m - 1 \) observations and \( n + 2 \) variables. The variables are named i, a1 through an, and b, as shown in Figure 10.14.

**Figure 10.14** CREATE DATA Set: The Iterated Column Form

<table>
<thead>
<tr>
<th>Obs</th>
<th>i a1 a2 a3 a4 b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 2 2 4 6 8 4</td>
</tr>
<tr>
<td>2</td>
<td>2 3 3 6 9 12 9</td>
</tr>
</tbody>
</table>
See the section “Data Set Input/Output” on page 364 for more examples of using the CREATE DATA statement.

**DO Statement**

```plaintext
DO ; statements ; END ;
```

The DO statement groups a sequence of statements together as a single statement. Each statement within the list is executed sequentially. The DO statement can be used for grouping with the IF, FOR, and COFOR statements.

**DO Statement, Iterative**

```plaintext
DO name = specification-1 [ , . . . specification-n ] ; statements ; END ;
```

The iterative DO statement assigns the values from the sequence of *specification* items to a previously declared parameter or variable, `name`. The specified statement sequence is executed after each assignment. This statement corresponds to the iterative DO statement of the DATA step.

Each *specification* provides either a single number or a single string value, or a sequence of such values. Each *specification* takes the following form:

```plaintext
expression [ WHILE( logic-expression ) | UNTIL( logic-expression ) ]
```

The *expression* in the *specification* provides a single value or set of values to assign to the target `name`. Multiple values can be provided for the loop by giving multiple *specification* items that are separated by commas. For example, the following statements output the values 1, 3, and 5:

```plaintext
proc optmodel;
  number i;
  do i=1,3,5;
    put i;
  end;
end;
```

In this case, the same effect can be achieved with a single range expression in place of the explicit list of values, as in the following statements:

```plaintext
proc optmodel;
  number i;
  do i=1 to 5 by 2;
    put 'value of i assigned by the DO loop = ' i;
    i=i**2;
    put 'value of i assigned in the body of the loop = ' i;
  end;
end;
```

The output of these statements is shown in Figure 10.15.

**Figure 10.15** DO Loop: Name Parameter Unaffected

```plaintext
value of i assigned by the DO loop = 1
value of i assigned in the body of the loop = 1
value of i assigned by the DO loop = 3
value of i assigned in the body of the loop = 9
value of i assigned by the DO loop = 5
value of i assigned in the body of the loop = 25
```
Unlike the DATA step, a range expression requires the limit to be specified. Additionally the BY part, if any, must follow the limit expression. Moreover, although the name parameter can be reassigned in the body of the loop, the sequence of values that is assigned by the DO loop is unaffected.

The argument expression can also be an expression that returns a set of numbers or strings. For example, the following statements produce the same sequence of values for i as the previous statements but use a set parameter value:

```plaintext
proc optmodel;
  set s = {1,3,5};
  number i;
  do i = s;
    put i;
  end;
end;
```

Each specification can include a WHILE or UNTIL clause. A WHILE or UNTIL clause applies to the expression that immediately precedes the clause. The sequence that is specified by an expression can be terminated early by a WHILE or UNTIL clause. A WHILE logic-expression is evaluated for each sequence value before the nested statements. If the logic-expression returns a false (zero or missing) value, then the current sequence is terminated immediately. An UNTIL logic-expression is evaluated for each sequence value after the nested statements. The sequence from the current specification is terminated if the logic-expression returns a true value (nonzero and nonmissing). After early termination of a sequence due to a WHILE or UNTIL expression, the DO loop execution continues with the next specification, if any.

To demonstrate use of the WHILE clause, the following statements output the values 1, 2, and 3. In this case the sequence of values from the set s is stopped when the value of i reaches 4.

```plaintext
proc optmodel;
  set s = {1,2,3,4,5};
  number i;
  do i = s while(i NE 4);
    put i;
  end;
end;
```

**DO UNTIL Statement**

```plaintext
DO UNTIL ( logic-expression ) ; statements ; END ;
```

The DO UNTIL loop executes the specified sequence of statements repeatedly until the logic-expression, evaluated after the statements, returns true (a nonmissing nonzero value).

For example, the following statements output the values 1 and 2:

```plaintext
proc optmodel;
  number i;
  i = 1;
  do until (i=3);
    put i;
    i=i+1;
  end;
end;
```

Multiple criteria can be introduced using expression operators, as in the following example:
do until (i=3 and j=7);

For a list of expression operators, see Table 10.10.

**DO WHILE Statement**

```
DO WHILE ( logic-expression ) ; statements ; END ;
```

The DO WHILE loop executes the specified sequence of statements repeatedly as long as the logic-expression, evaluated before the statements, returns true (a nonmissing nonzero value).

For example, the following statements output the values 1 and 2:

```
proc optmodel;
  number i;
  i = 1;
  do while (i<3);
    put i;
    i=i+1;
  end;
```

Multiple criteria can be introduced using expression operators, as in the following example:

```
do while (i<3 and j<7);
```

For a list of expression operators, see Table 10.10.

**DROP Statement**

```
DROP constraint-list ;
```

The DROP statement causes the solver to ignore a list of constraints, constraint arrays, or constraint array locations. The space-delimited constraint-list specifies the names of the dropped constraints. Each constraint, constraint array, or constraint array location is named by an identifier-expression. Each identifier-expression can be prefixed by an indexing set to specify the constraint indices.

The following example statements use the DROP statement:

```
proc optmodel;
  var x{1..10};
  con disp{i in 1..9}: x[i+1] >= x[i] + 0.1;

  drop cl;       /* drops the cl constraint */
  drop disp[5];  /* drops just disp[5] */
  drop disp;     /* drops all disp constraints */
```

The following line drops both the cl and disp[5] constraints:

```
drop cl disp[5];
```

Constraints can be added back to the model by using the **RESTORE** statement.
EXPAND Statement

```
EXPAND [ identifier-expression ] [ / options ] ;
```

The EXPAND statement prints the specified constraint, variable, implicit variable, or objective declaration expressions in the current problem after expanding aggregation operators, substituting the current value for parameters and indices, and resolving constant subexpressions. `identifier-expression` is the name of a variable, objective, or constraint. If the name is omitted and no `options` are specified, then all variables, the last objective, used implicit variables, and undropped constraints in the current problem are printed. The following statements show an example EXPAND statement:

```
proc optmodel;
    number n=2;
    var x{1..n};
    min z1=sum{i in 1..n}(x[i]-i)**2;
    max z2=sum{i in 1..n}(i-x[i])**3;
    con c{i in 1..n}: x[i]>=0;
    fix x[2]=3;
    expand;
```

These statements produce the output in Figure 10.16.

**Figure 10.16** EXPAND Statement Output

Var x[1]
Fix x[2] = 3
Maximize z2=(-x[1] + 1)**3 + (-x[2] + 2)**3
Constraint c[1]: x[1] >= 0
Constraint c[2]: x[2] >= 0

Specifying an `identifier-expression` restricts output to the specified declaration. A non-array name prints only the specified item. If an array name is used with a specific index, then information for the specified array location is output. Using an array name without an index restricts output to all locations in the array.

You can use the following `options` to further control the EXPAND statement output:

**SOLVE**

causes the EXPAND statement to print the variables, objectives, and constraints in the same form that would be seen by the solver if a SOLVE statement were executed. This includes any transformations by the PROC OPTMODEL presolver (see the section “Presolver” on page 392). In this form any fixed variables are replaced by their values. Unless an `identifier-expression` specifies a particular non-array item or array location, the EXPAND output is restricted to only the variables, the constraints, and the current problem objective.

The following options restrict the types of declarations output when no specific non-array item or array location is requested. By default, all types of declarations are output. Only the requested declaration types are output when one or more of the following options are used.

**CONSTRAINT** | **CON**

requests the output of undropped constraints.
FIX
requests the output of fixed variables. These variables might have been fixed by the FIX statement (or by the presolver if the SOLVE option is specified). The FIX option can also be used in combination with the name of a variable array to display just the fixed elements of the array.

IIS
restricts the display to items found in the irreducible infeasible set (IIS) after the most recent SOLVE performed by the LP solver with the IIS=ON option. The IIS option for the EXPAND statement can also be used in combination with the name of a variable or constraint array to display only the elements of the array in the IIS. For more information about IIS, see the section “Irreducible Infeasible Set” on page 519.

IMPVAR
requests the output of nonconstant implicit variables referenced in the current problem.

OBJECTIVE | OBJ
requests the output of objectives used in the current problem. This includes the current problem objective and any objectives referenced as implicit variables.

OMITTED
requests the output of variables that are referenced by problem equations but were not included in the current USE PROBLEM instance. The OPTMODEL procedure omits these variables from the generated problem.

VAR
requests the output of unfixed variables. The VAR option can also be used in combination with the name of a variable array to display just the unfixed elements of the array.

For example, you can see the effect of a FIX statement on the problem that is presented to the solver by using the SOLVE option. You can modify the previous example as follows:

```
proc optmodel;
    number n=2;
    var x{1..n};
    min z1=sum{i in 1..n}(x[i]-i)**2;
    max z2=sum{i in 1..n}(i-x[i])**3;
    con c{i in 1..n}: x[i]>=0;
    fix x[2]=3;
    expand / solve;
```

These statements produce the output in Figure 10.17.

**Figure 10.17** Expansion with Fixed Variable

<table>
<thead>
<tr>
<th>Var x[1] &gt;= 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fix x[2] = 3</td>
</tr>
<tr>
<td>Maximize z2=(-x[1] + 1)**3 - 1</td>
</tr>
</tbody>
</table>

Compare the results in Figure 10.17 to those in Figure 10.16. The constraint c[1] has been converted to a variable bound. The subexpression that uses the fixed variable has been resolved to a constant.
FILE Statement

\[ \text{FILE file-specification [ LRECL=value ] ;} \]

The FILE statement selects the current output file for the PUT statement. By default PUT output is sent to the SAS log. Use the FILE statement to manage a group of output files. The specified file is opened for output if it is not already open. The output file remains open until it is closed with the CLOSEFILE statement.

file-specification names the output file. It can use any of the following forms:

'external-file'

specifies the physical name of an external file in quotation marks. The interpretation of the filename depends on the operating environment.

file-name

specifies the logical name associated with a file by the FILENAME statement or by the operating environment. The names PRINT and LOG are reserved to refer to the SAS listing and log files, respectively.

**NOTE:** Details about the FILENAME statement can be found in *SAS Global Statements: Reference*.

( expression )

specifies an expression that evaluates to a string that contains the physical name of an external file.

The LRECL= option sets the line length of the output file. The LRECL= option is ignored if the file is already open or if the PRINT or LOG file is specified.

The LRECL= value can be specified in these forms:

integer

specifies the desired line length.

identifier-expression

specifies the name of a numeric parameter that contains the length.

( expression )

specifies a numeric expression in parentheses that returns the line length.

The LRECL= value cannot exceed the largest four-byte signed integer, which is $2^{31} - 1$.

The following example shows how to use the FILE statement to handle multiple files:

```plaintext
proc optmodel;
    file 'file.txt' lrecl=80; /* opens file.txt */
    put 'This is line 1 of file.txt.';
    file print; /* selects the listing */
    put 'This goes to the listing.';
    file 'file.txt'; /* reselects file.txt */
    put 'This is line 2 of file.txt.';
    closefile 'file.txt'; /* closes file.txt */
    file log; /* selects the SAS log */
    put 'This goes to the log.';
```
/* using expression to open and write a collection of files */
str ofile;
um i;
um l = 40;
do i = 1 to 3;
ofile = ('file' || i || '.txt');
file (ofile) lrecl=(l*i);
put ('This goes to ' || ofile);
closefile (ofile);
end;

The following statements illustrate the usefulness of using a logical name associated with a file by FILENAME statement:

    proc optmodel;
    /* assigns a logical name to file.txt */
    /* see FILENAME statement in */
    /* SAS Global Statements: Reference */
    filename myfile 'file.txt' mod;
    file myfile;
    put 'This is line 3 of file.txt.';
    closefile myfile;
    file myfile;
    put 'This is line 4 of file.txt.';
    closefile myfile;

Notice that the FILENAME statement opens the file referenced for append. Therefore, new data are appended to the end every time the logical name, myfile, is used in the FILE statement.

**FIX Statement**

    FIX variable-list [ = expression ] ;

The FIX statement causes the solver to treat a list of variables, variable arrays, or variable array locations as fixed in value. The *variable-list* consists of one or more variable names separated by spaces. Each member of the *variable-list* is fixed to the same *expression*. For example, the following statements fix the variables x and y to 3:

    proc optmodel;
    var x, y;
    num a = 2;
    fix x y=a+1;

A variable is specified with an *identifier-expression* (see the section “Identifier Expressions” on page 347). An entire variable array is fixed if the *identifier-expression* names an array without providing an index. Each *identifier-expression* can be prefixed by an *indexing set* to specify the variable indices.

You can specify a new value for the variables by using the *expression*. If a single *identifier-expression* is specified, prefixed by an indexing set, then the *expression* is evaluated for each element in the indexing set. Otherwise the *expression* is evaluated exactly once. For example, the following statements fix all locations in array x to 0 except x[10], which is fixed to 1:
Chapter 10: The OPTMODEL Procedure

```plaintext
proc optmodel;
  var x{1..10};
  fix x = 0;
  fix x[10] = 1;

The same effect can be achieved using a single statement:

  fix {i in 1..10} x[i] = if i eq 10 then 1 else 0;

If expression is omitted, the variable is fixed at its current value. For example, you can fix some variables to be their optimal values after the SOLVE statement is invoked. NOTE: The fixed value is equal to the current value for a fixed variable. The fixed value is updated if a new value is assigned to a fixed variable.

The effect of FIX can be reversed by using the UNFIX statement.

FOR Statement

FOR {index-set} statement ;

The FOR statement executes its substatement for each member of the specified index-set. The index set can declare local dummy parameters. You can reference the value of these parameters in the substatement. For example, consider the following statements:

```plaintext
 proc optmodel;
  for {i in 1..2, j in {'a', 'b'}} put i= j=;

These statements produce the output in Figure 10.18.

Figure 10.18 FOR Statement Output

i=1 j=a
i=1 j=b
i=2 j=a
i=2 j=b

As another example, the following statements set the current values for variable x to random values between 0 and 1:

```plaintext
 proc optmodel;
  var x{1..10};
  for {i in 1..10}
    x[i] = ranuni(-1);

Multiple statements can be controlled by specifying a DO statement group for the substatement.

CAUTION: Avoid modifying the parameters that are used by the FOR or COFOR statement index set from within the substatement. The set value that is used for the left-most index set item is not affected by such changes. However, the effect of parameter changes on later index set items cannot be predicted.

IF Statement

IF logic-expression THEN statement [ ELSE statement ] ;

The IF statement evaluates the logical expression and then conditionally executes the THEN or ELSE substatements. The substatement that follows the THEN keyword is executed when the logical expression
result is nonmissing and nonzero. The ELSE substatement, if any, is executed when the logical expression result is a missing value or zero. The ELSE part is optional and must immediately follow the THEN substatement. When IF statements are nested, an ELSE is always matched to the nearest incomplete unmatched IF-THEN. Multiple statements can be controlled by using DO statements with the THEN or ELSE substatements.

**NOTE:** When an IF-THEN statement is used without an ELSE substatement, substatements of the IF statement are executed when possible as they are entered. Under certain circumstances, such as when an IF statement is nested in a FOR loop, the statement is not executed during interactive input until the next statement is seen. By following the IF-THEN statement with an extra semicolon, you can cause it to be executed upon submission, since the extra semicolon is handled as a null statement.

### LEAVE Statement

**LEAVE ;**

The LEAVE statement terminates the execution of the entire loop body (iterative DO, DO UNTIL, DO WHILE, FOR, or COFOR) that immediately contains the LEAVE statement. Execution resumes at the statement that follows the loop. The following example demonstrates a simple use of the LEAVE statement:

```plaintext
proc optmodel;
  number i, j;
  do i = 1..5;
    do j = 1..4;
      if i >= 3 and j = 2 then leave;
    end;
    print i j;
  end;
end;
```

The results from these statements are displayed in Figure 10.19.

**Figure 10.19** LEAVE Statement Output

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

For values of i equal to 1 or 2, the inner loop continues uninterrupted, leaving j with a value of 4. For values of i equal to 3, 4, or 5, the inner loop terminates early, leaving j with a value of 2.
Null Statement

;

The null statement is treated as a statement in the PROC OPTMODEL syntax, but its execution has no effect. It can be used as a placeholder statement.

PRINT Statement

PRINT print-items ;

The PRINT statement outputs string and numeric data in tabular form. The statement specifies a list of arrays or other data items to print. Multiple items can be output together as data columns in the same table.

If no format is specified, the PRINT statement handles the details of formatting automatically (see the section “Formatted Output” on page 368 for details). The default format for a numerical column is the fixed-point format ($w.d$ format), which is chosen based on the values of the PDIGITS= and PWIDTH= options (see the section “PROC OPTMODEL Statement” on page 284) and on the values in the column. The PRINT statement uses scientific notation (the $Ew.$ format) when a value is too large or too small to display in fixed format. The default format for a character column is the $w.$ format, where the width is set to be the length of the longest string (ignoring trailing blanks) in the column.

print-item can be specified in the following forms:

identifier-expression [ format ]

specifies a data item to output. identifier-expression can name an array. In that case all defined array locations are output. format specifies a SAS format that overrides the default format.

( expression ) [ format ]

specifies a data value to output. format specifies a SAS format that overrides the default format.

{ index-set } identifier-expression [ format ]

specifies a data item to output under the control of an index set. The item is printed as if it were an array with the specified set of indices. This form can be used to print a subset of the locations in an array, such as a single column. If the identifier-expression names an array, then the indices of the array must match the indices of the index-set. The format argument specifies a SAS format that overrides the default format.

{ index-set } ( expression ) [ format ]

specifies a data item to output under the control of an index set. The item is printed as if it were an array with the specified set of indices. In this form the expression is evaluated for each member of the index-set to create the array values for output. format specifies a SAS format that overrides the default format.

string

specifies a string value to print.

_PAGE_

specifies a page break.
The following example demonstrates the use of several print-item forms:

```plaintext
proc optmodel;
    num x = 4.3;
    var y{j in 1..4} init j*3.68;
    print y; /* identifier-expression */
    print (x * .265) dollar6.2; /* (expression) [format] */
    print {i in 2..4} y; /* {index-set} identifier-expression */
    print {i in 1..3}(i + i*.2345692) best7.;
        /* {index-set} (expression) [format] */
    print "Line 1"; /* string */
```

The output is displayed in Figure 10.20.

**Figure 10.20 Print-item Forms**

<table>
<thead>
<tr>
<th></th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.68</td>
</tr>
<tr>
<td>2</td>
<td>7.36</td>
</tr>
<tr>
<td>3</td>
<td>11.04</td>
</tr>
<tr>
<td>4</td>
<td>14.72</td>
</tr>
</tbody>
</table>

$1.14$

<table>
<thead>
<tr>
<th></th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7.36</td>
</tr>
<tr>
<td>3</td>
<td>11.04</td>
</tr>
<tr>
<td>4</td>
<td>14.72</td>
</tr>
</tbody>
</table>

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.23457</td>
</tr>
<tr>
<td>2</td>
<td>2.46914</td>
</tr>
<tr>
<td>3</td>
<td>3.70371</td>
</tr>
</tbody>
</table>

Line 1

Adjacent print items that have similar indexing are grouped together and output in the same table. Items have similar indexing if they specify arrays that have the same number of indices and have matching index types (numeric versus string). Nonarray items are considered to have the same indexing as other nonarray items. The resulting table has a column for each array index followed by a column for each print item value. This format is called list form. For example, the following statements produce a list form table:

```plaintext
proc optmodel;
    num a{i in 1..3} = i*i;
    num b{i in 3..5} = 4*i;
    print a b;
```

These statements produce the listing output in Figure 10.21.
Chapter 10: The OPTMODEL Procedure

Figure 10.21 List Form PRINT Table

<table>
<thead>
<tr>
<th></th>
<th>a</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

The array index columns show the set of valid index values for the print items in the table. The array index column for the $i$th index is labeled $[i]$. There is a row for each combination of index values that was used. The index values are displayed in sorted ascending order.

The data columns show the array values that correspond to the index values in each row. If a particular array index is invalid or the array location is undefined, then the corresponding table entry is displayed as blank for numeric arrays and as an empty string for string arrays. If the print items are scalar, then the table has a single row and no array index columns.

If a table contains a single array print item, the array is two-dimensional (has two indices), and the array is dense enough, then the array is shown in matrix form. In this format there is a single index column that contains the row index values. The label of this column is blank. This column is followed by a column for every unique column index value for the array. The latter columns are labeled by the column value. These columns contain the array values for that particular array column. Table entries that correspond to array locations that have invalid or undefined combinations of row and column indices are blank or (for strings) printed as an empty string.

The following statements generate a simple example of matrix output:

```plaintext
proc optmodel;
    print {i in 1..6, j in i..6} (i*10+j);
```

The PRINT statement produces the output in Figure 10.22.

Figure 10.22 Matrix Form PRINT Table

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>33</td>
<td>34</td>
<td>35</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>44</td>
<td>45</td>
<td>46</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td>55</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>66</td>
<td></td>
</tr>
</tbody>
</table>

The PRINT statement prints single two-dimensional arrays in the form that uses fewer table cells (headings are ignored). Sparse arrays are normally printed in list form, and dense arrays are normally printed in matrix form. In a PROC OPTMODEL statement, the PMATRIX= option enables you to tune how the PRINT statement displays a two-dimensional array. The value of this option scales the total number of nonempty array elements, which is used to compute the tables cells needed for list form display. Specifying values for the PMATRIX= option less than 1 causes the list form to be used in more cases, while specifying values greater than 1 causes the matrix form to be used in more cases. If the value is 0, then the list form is always used. The default value of the PMATRIX= option is 1. Changing the default can be done with the RESET OPTIONS statement.
The following statements illustrate how the PMATRIX= option affects the display of the PRINT statement:

```plaintext
proc optmodel;
    num a{i in 1..6, i..i} = i;
    num b{i in 1..3, j in 1..3} = i*j;
    print a;
    print b;
    reset options pmatrix=3;
    print a;
    reset options pmatrix=0.5;
    print b;
```

The output is shown in Figure 10.23.

**Figure 10.23** PRINT Statement: Effects of PMATRIX= Option

<table>
<thead>
<tr>
<th>[1]</th>
<th>[2]</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[1]</th>
<th>[2]</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>9</td>
</tr>
</tbody>
</table>

From Figure 10.23, you can see that, by default, the PRINT statement tries to make the display compact. However, you can change the default by using the PMATRIX= option.
PROFILE Statement

PROFILE [ mode ] options ;

The PROFILE statement controls the PROC OPTMODEL profiler, which enables you to collect and display timing and execution count information for PROC OPTMODEL processing. The profiler can be very useful for finding bottlenecks during execution, such as constraints that require large amounts of time during problem generation. When the profiler is enabled, PROC OPTMODEL records the time for processing a declaration, the time for executing statements, and the number of times that statements are executed. See “Example 10.7: Sparse Modeling” on page 430 for an example use of the PROFILE statement.

The mode argument specifies the action that the PROFILE statement performs. You can use the following values for mode:

ON          enables the profiler. Data are collected until the profiler is disabled or PROC OPTMODEL terminates. The profiler is also enabled when no mode is specified in a PROFILE statement.

OFF         disables the profiler. Note that the profiler is disabled when PROC OPTMODEL begins execution.

PRINT       prints the current accumulated profiler data. Items for declarations and statements are displayed in a table in descending order of their net time. Accumulated data are printed automatically when PROC OPTMODEL terminates.

The options control how PROC OPTMODEL collects and displays profiler information. You can specify the following options:

MEMORY | NOMEMORY
specifies whether to add the memory requirements for declarations to the output for the PROFILE PRINT action. The MemSize column of the ProfileInfo ODS table contains the approximate memory requirements in bytes when the table is displayed. The memory that is used by shared values, such as strings and sets, is divided among the declarations that reference the values. NOMEMORY is the default.

PERCENT=number
restricts the output for PROFILE PRINT to items whose net times account for at least the specified percentage of total profiled time, total time \( \times \) number/100. Items that have smaller times are aggregated into a single item at the end of the table. You can set this option before the display of profile data, and it does not affect the data collection. The value of number can range from 0 to 100. The default value is 1.

RESET       discards accumulated profiler data when the PROFILE statement completes execution. Accumulated data are retained until they are explicitly reset.
allows collection of profiler data for nested statements. With the default option, STMTDEPTH=1, profiler data are collected only for top-level statements. The elapsed time for nested statement timing is included in the top-level statement timing. For example, time for a top-level FOR statement would include the execution of its substatement. Use the STMTDEPTH= option to profile the nested statements individually. The value number specifies the maximum nesting depth at which to profile statements individually. The nesting depth of a top-level statement is 1. Otherwise the nesting depth of a statement is one more than the nesting depth of the statement that encloses it, such as a DO, IF, or FOR statement.

For a PROFILE statement within a DO block or DO loop, the statement depth value is interpreted relative to the enclosing DO statement. For example, specifying STMTDEPTH=1 within a DO block causes the top-level statements of the DO block to be profiled. The STMTDEPTH= option is reset to its previous value when the enclosing DO statement completes execution.

The value of number can be an integer between 1 and 32,767. Using the ALL keyword is equivalent to specifying 32,767. Note that profiler timing can add significant overhead. Use a small number to minimize overhead.

The elapsed time that is required to process an item includes the processing of other profiled items that it depends on. For example, the processing of a constraint during problem generation might require the evaluation of parameter values. The PROFILE statement reports net time so that the total of profiled times represents actual processing time.

The equation that is used to compute net time is

\[
\text{net time} = \text{elapsed time} - \text{nested time} - \text{wait time}
\]

Elapsed time is the elapsed clock time this is required for processing an item. Nested time is the total of the elapsed times that are spent within the same thread to process other profiled items, such as substatements or declaration values. Wait time represents the time that a single thread is allocated but idle because it is waiting for other threads to perform the required processing. The total of net times can exceed the elapsed wall clock time when multiple threads are used.

**PUT Statement**

```
PUT [ put-items ] [ @ | @@ ] ;
```

The PUT statement writes text data to the current output file. The syntax of the PUT statement in PROC OPTMODEL is similar to the syntax of the PROC IML and DATA step PUT statements. The PUT statement contains a list of items that specify data for output and provide instructions for formatting the data.

The current output file is initially the SAS log. This can be overridden with the FILE statement. An output file can be closed with the CLOSEFILE statement.

Normally the PUT statement outputs the current line after processing all items. Final @ or @@ operators suppress this automatic line output and cause the current column position to be retained for use in the next PUT statement.

**put-item** can take any of the following forms.
Chapter 10: The OPTMODEL Procedure

identifier-expression [ = ] [ format ]
outputs the value of the parameter or variable that is specified by the identifier-expression. The equal sign (=) causes a name for the location to be printed before each location value. If the identifier-expression uses a suffix that can be followed by a solution index, you can use an asterisk (*) instead of the index expression to display all saved solution values. Normally each item value is printed in a default format. Any leading and trailing blanks in the formatted value are removed, and the value is followed by a blank space. When an explicit format is specified, the value is printed within the width determined by the format.

name[*] [ .suffix [ ] expression | * ] [ = ] [ format ]
outputs each defined location value for an array parameter. The array name is specified as in the identifier-expression form except that the index list is replaced by an asterisk (*). The equal sign (=) causes a name for the location to be printed before each location value along with the actual index values to be substituted for the asterisk. If the suffix can be followed by a solution index, you can use an asterisk (*) instead of the index expression to display all saved solution values. Each item value normally prints in a default format. Any leading and trailing blanks in the formatted value are removed, and the value is followed by a blank space. When an explicit format is specified, the value is printed within the width determined by the format.

( expression ) [ = ] [ format ]
outputs the value of the expression enclosed in parentheses. This produces similar results to the identifier-expression form except that the equal sign (=) uses the expression to form the name.

'quoted-string'
copies the string to the output file.

@integer | identifier-expression | ( expression ) sets the absolute column position within the current line. The literal or expression value determines the new column position.

+integer | identifier-expression( expression ) sets the relative column position within the current line. The literal or expression value determines the amount to update the column position.

/outputs the current line and moves to the first column of the next line.

_PAGE_
outputs any pending line data and moves to the top of the next page.

QUIT Statement

QUIT ;
The QUIT statement terminates the OPTMODEL execution. The statement is executed immediately, so it cannot be a nested statement. A QUIT statement is implied when a DATA or PROC statement is read.

READ DATA Statement

READ DATA SAS-data-set [ NOMISS ] INTO [ [ set-name = ] [ read-key-columns ] ] [ read-columns ] ;
The READ DATA statement reads data from a SAS data set into PROC OPTMODEL parameter and variable locations. The arguments to the READ DATA statement are as follows:
SAS-data-set specifies the input data set name and options. You can specify the data set name and options directly or as the string value of an expression enclosed in parentheses.

set-name specifies a set parameter in which to save the set of observation key values read from the input data set.

read-key-columns provide the index values for array destinations.

read-columns specify the data values to read and the destination locations.

NOTE: Currently, the data are always saved on the client. Transferring large data sets from a CAS libref can have significant network overhead and require a large amount of client memory.

The following example uses the READ DATA statement to copy data set variables j and k from the SAS data set indata into parameters of the same name. The READ= data set option specifies a password.

```
proc optmodel;
    number j, k;
    read data indata(read=secret) into j k;
```

Key Columns

If any read-key-columns are specified, then the READ DATA statement reads all observations from the input data set. If no read-key-columns are specified, then only the first observation of the data set is read. The data set is closed after reading the requested information.

Each read-key-column specifies a data set variable that supplies the column value. The values of the specified data set variables from each observation are combined into a key tuple. This combination is known as the observation key. The observation key is used to index array locations specified by the read-columns items. The observation key is expected to be unique for each observation read from the data set.

The syntax for a read-key-column is as follows:

```
[ name = ] source-name [ / trim-option ]
```

Each read-key-column specifies an element of the observation key tuple. The source-name specifies the data set variable name, either as a name or by using a COL expression (as described later for read-columns). Use the special data set variable name _N_ to refer to the observation’s iteration number, starting at 1.

A local dummy parameter can be created for each read-key-column. You can use the dummy parameter to reference the key tuple element value in subsequent read-columns items. A dummy parameter is created when the read-key-column specifies a name preceding the equal sign (=). The dummy parameter has the specified name. Also, a dummy parameter is created when the source-name is a nonliteral name and no alternate name has been specified using the equal sign. In this case, the dummy parameter has the same name as the data set variable.

You can specify a set-name to save the set of observation keys into a set parameter. If the observation key consists of a single scalar value, then the set member type must match the scalar type. Otherwise the set member type must be a tuple with element types that match the corresponding observation key element types.

The READ DATA statement initially assigns an empty set to the target set-name parameter. As observations are read, a tuple for each observation key is added to the set. A set used to index an array destination in the
read-columns can be read at the same time as the array values. Consider a data set, invdata, created by the following statements:

```plaintext
data invdata;
    input item $ invcount;
datalines;
    table 100
    sofa 250
    chair 80
;
```

The following statements read the data set invdata, which has two variables, item and invcount. The READ DATA statement constructs a set of inventory items, Items. At the same time, the parameter location invcount[item] is assigned the value of the data set variable invcount in the corresponding observation.

```plaintext
proc optmodel;
    set<string> Items;
    number invcount{Items};
    read data invdata into Items=[item] invcount;
    print invcount;
```

The output of these statements is shown in Figure 10.24.

**Figure 10.24** READ DATA Statement: Key Column

<table>
<thead>
<tr>
<th></th>
<th>invcount</th>
</tr>
</thead>
<tbody>
<tr>
<td>chair</td>
<td>80</td>
</tr>
<tr>
<td>sofa</td>
<td>250</td>
</tr>
<tr>
<td>table</td>
<td>100</td>
</tr>
</tbody>
</table>

When observations are read, the values of data set variables are copied to parameter locations. Numeric values are copied unchanged. For character values, trim-option controls how leading and trailing blanks are processed. trim-option is ignored when the value type is numeric. Specify any of the following keywords for trim-option:

- TRIM | TR: removes leading and trailing blanks from the data set value. This is the default behavior.
- LTRIM | LT: removes only leading blanks from the data set value.
- RTRIM | RT: removes only trailing blanks from the data set value.
- NOTRIM | NT: copies the data set value with no changes.

Columns
read-columns specify data set variables to read and PROC OPTMODEL parameter locations to which to assign the values. The types of the input data set variables must match the types of the parameters. Array parameters can be implicitly or explicitly indexed by the observation key values.

Normally, missing values from the data set are assigned to the parameters that are specified in the read-columns. The NOMISS keyword suppresses the assignment of missing values, leaving the corresponding
parameter locations unchanged. Note that the parameter location does not need to have a valid index in this case. This permits a single statement to read data into multiple arrays that have different index sets.

*read-columns* have the following forms:

\[
\text{identifier-expression} \ [ = \text{name} \mid \text{COL}(\text{name-expression})] \ [ / \text{trim-option} ]
\]

transfers an input data set variable to a target parameter or variable. *identifier-expression* specifies the target. If the *identifier-expression* specifies an array without an explicit index, then the observation key provides an implicit index. The name of the input data set variable can be specified with a *name* or a COL expression. Otherwise the data set variable name is given by the *name* part of the *identifier-expression*. For COL expressions, the string-valued *name-expression* is evaluated to determine the data set variable name. *trim-option* controls removal of leading and trailing blanks in the incoming data. For example, the following statements read the data set variables *column1* and *column2* from the data set *exdata* into the PROC OPTMODEL parameters *p* and *q*, respectively. The observation numbers in *exdata* are read into the set *indx*, which indexes *p* and *q*.

```plaintext
data exdata;
  input column1 column2;
datalines;
1 2
3 4
;

proc optmodel;
  number n init 2;
  set<num> indx;
  number p[indx], q[indx];
  read data exdata into
    indx=[_N_] p=column1 q=col("column"||n);
  print p q;
```

The output is shown in Figure 10.25.

**Figure 10.25** READ DATA Statement: Identifier Expressions

<table>
<thead>
<tr>
<th></th>
<th>p</th>
<th>q</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

\{ index-set \} < read-columns >

performs the transfers by iterating each column specified by *read-columns* for each member of the index set. If there are *n* columns and *m* index set members, then *n* × *m* columns are generated. The dummy parameters from the index set can be used in the columns to generate distinct input data set variable names in the iterated columns, using COL expressions. The columns are expanded when the READ DATA statement is executed, before any observations are read. This form of read-columns cannot be nested. In other words, the following form of read-columns is NOT allowed:
You can use an iterated column read-option to read multiple data set variables into the same array. For example, a data set might store an entire row of array data in a group of data set variables. The following statements demonstrate how to read a data set that contains demand data divided by day:

```plaintext
data dmnd;
    input loc $ day1 day2 day3 day4 day5;
datalines;
East 1.1 2.3 1.3 3.6 4.7
West 7.0 2.1 6.1 5.8 3.2
;

proc optmodel;
    set DOW = 1..5; /* days of week, 1=Monday, 5=Friday */
    set<string> LOCS; /* locations */
    number demand{LOCS, DOW};
    read data dmnd
        into LOCS=[loc]
            {d in DOW} < demand[loc, d]=col("day"||d) >;
    print demand;
```

These statements read a set of demand variables named DAY1–DAY5 from each observation, filling in the two-dimensional array demand. The output is shown in Figure 10.26.

**Figure 10.26 Demand Data**

<table>
<thead>
<tr>
<th>demand</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>East</td>
</tr>
<tr>
<td>1.1 2.3 1.3 3.6 4.7</td>
</tr>
<tr>
<td>West</td>
</tr>
<tr>
<td>7.0 2.1 6.1 5.8 3.2</td>
</tr>
</tbody>
</table>

**RESET OPTIONS Statement**

**RESET OPTIONS** `options` ;

**RESET OPTION** `options` ;

The RESET OPTIONS statement sets PROC OPTMODEL option values or restores them to their defaults. Options can be specified by using the same syntax as in the PROC OPTMODEL statement. The RESET OPTIONS statement provides two extensions to the option syntax. If an option normally requires a value (specified with an equal sign (=) operator), then specifying the option name alone resets it to its default value. You can also specify an expression enclosed in parentheses in place of a literal value. See the section “OPTMODEL Options” on page 400 for an example.

The RESET OPTIONS statement can be placed inside loops or conditional statements. The statement is applied each time it is executed.
RESTORE Statement

RESTORE constraint-list ;

The RESTORE statement adds a list of constraints, constraint arrays, or constraint array locations that were dropped by the DROP statement back into the solver model, or includes constraints in a problem where they were not previously present. The space-delimited constraint-list specifies the names of the constraints. Each constraint, constraint array, or constraint array location is named by an identifier-expression. An entire constraint array is restored if an identifier-expression omits the index from an array name. Each identifier-expression can be prefixed by an indexing set to specify the constraint indices.

For example, the following statements declare a constraint array and then drop it:

\[
\text{con } \text{c[i in 1..4]: } x[i] + y[i] <= 1; \\
\text{drop c;}
\]

The following statement restores the first constraint:

\[
\text{restore } c[1];
\]

The following statement restores the second and third constraints:

\[
\text{restore } \{i \text{ in 2..3} \} \text{c[i]; } /* \text{like } "\text{restore } c[2] c[3];" */
\]

If you want to restore all of the constraints, you can submit the following statement:

\[
\text{restore } c;
\]

SAVE MPS Statement

SAVE MPS SAS-data-set [(OBJECTIVE | OBJ) name] [(NOOBJECTIVE | NOOBJ)] ;

The SAVE MPS statement saves the structure and coefficients for a linear programming model into a SAS data set. This data set can be used as input data for the OPTLP or OPTMILP procedure.

NOTE: The OPTMODEL presolver (see the section “Presolver” on page 392) is automatically bypassed so that the statement saves the original model without eliminating fixed variables, tightening bounds, and so on.

The SAS-data-set argument specifies the output data set name and options. You can specify the data set name and options directly or as the string value of an expression enclosed in parentheses. The output data set uses the MPS format described in Chapter 9, “The MPS-Format Data Table.” An _ID_ column is automatically added to the data set when it is created using a CAS engine libref. The generated data set contains observations that define different parts of the linear program.

Variables, constraints, and objectives are referenced in the data set by using label text from the corresponding .label suffix value. The default text is based on the name in the model. See the section “Suffixes” on page 380 for more details. Labels are limited by default to 32 characters and are abbreviated to fit. You can change the maximum length for labels by using the MAXLABELN= option. When needed, a programmatically generated number is added to labels to avoid duplication.

If the OBJECTIVE keyword is used, the objective name becomes the current problem objective. If the NOOBJECTIVE keyword is used or the current problem does not have an objective, then the data set includes a default constant zero objective. Otherwise, the current problem objective is included in the data set.

When an integer variable has been assigned a nondefault branching priority or direction, the MPS data set includes a BRANCH section. See Chapter 9, “The MPS-Format Data Table,” for more details.
The following statements show an example of the SAVE MPS statement. The LIBNAME statement for the mycas libref allows the creation of CAS tables. The model is specified using the OPTMODEL procedure. Then it is saved as the MPS data set mycas.MPSData, as shown in Figure 10.27. Next, PROC OPTLP is used to solve the resulting linear program.

```sas
libname mycas cas sessref=mysess datalimit=all;
proc optmodel;
var x >= 0, y >= 0;
con c: x >= y;
con bx: x <= 2;
con by: y <= 1;
min obj=0.5*x-y;
save mps mycas.MPSData;
quilt;

proc optlp data=mycas.MPSData pout=mycas.PrimalOut dout=mycas.DualOut;
run;
```

Figure 10.27 The MPS Data Set Generated by SAVE MPS Statement

<table>
<thead>
<tr>
<th>FIELD1</th>
<th>FIELD2</th>
<th>FIELD3</th>
<th>FIELD4</th>
<th>FIELD5</th>
<th>FIELD6</th>
<th><em>ID</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>MPSData</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>ROWS</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>obj</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>c</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>bx</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>by</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>COLUMNS</td>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>obj</td>
<td>0.5</td>
<td>c</td>
<td>1</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>bx</td>
<td>1.0</td>
<td>.</td>
<td>9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>obj</td>
<td>-1.0</td>
<td>c</td>
<td>-1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td>by</td>
<td>1.0</td>
<td>.</td>
<td>11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RHS</td>
<td></td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>RHS. bx</td>
<td>2.0</td>
<td>.</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RHS. by</td>
<td>1.0</td>
<td>.</td>
<td>14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENDTDATA</td>
<td>.</td>
<td>.</td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SAVE QPS Statement

```
SAVE QPS SAS-data-set [( OBJECTIVE | OBJ ) name ] [( NOOBJECTIVE | NOOBJ ) ];
```

The SAVE QPS statement saves the structure and coefficients for a quadratic programming model into a SAS data set. This data set can be used as input data for the OPTQP procedure.

**NOTE:** The OPTMODEL presolver (see the section “Presolver” on page 392) is automatically bypassed so that the statement saves the original model without eliminating fixed variables, tightening bounds, and so on.

The **SAS-data-set** argument specifies the output data set name and options. You can specify the data set name and options directly or as the string value of an expression enclosed in parentheses. The output data set uses the QPS format described in Chapter 9. An _ID_ column is automatically added to the data set when it is created using a CAS engine libref. The generated data set contains observations that define different parts of the quadratic program.
Variables, constraints, and objectives are referenced in the data set by using label text from the corresponding .label suffix value. The default text is based on the name in the model. See the section “Suffixes” on page 380 for more details. Labels are limited by default to 32 characters and are abbreviated to fit. You can change the maximum length for labels by using the MAXLABELN= option. When needed, a programmatically generated number is added to labels to avoid duplication.

If the OBJECTIVE keyword is used, the objective name becomes the current problem objective. If the NOOBJECTIVE keyword is used or the current problem does not have an objective, then the data set includes a default constant zero objective. Otherwise, the current problem objective is included in the data set. The quadratic coefficients of the objective function appear in the QSECTION section of the output data set.

The following statements show an example of the SAVE QPS statement. The LIBNAME statement for the mycas libref allows the creation of CAS tables. The model is specified using the OPTMODEL procedure. Then it is saved as the QPS data set mycas.QPSData, as shown in Figure 10.28. Next, PROC OPTQP is used to solve the resulting quadratic program.

```plaintext
libname mycas cas sessref=mysess datalimit=all;
proc optmodel;
  var x{1..2} >= 0;
  min z = 2*x[1] + 3 * x[2] + x[1]**2 + 10*x[2]**2 + 2.5*x[1]*x[2];
  con c2: x[1] + 2*x[2] >= 100;
  save qps mycas.QPSData;
quit;
```

```plaintext
proc optqp data=mycas.QPSData pout=mycas.PrimalOut dout=mycas.DualOut;
run;
```

<table>
<thead>
<tr>
<th>FIELD1</th>
<th>FIELD2</th>
<th>FIELD3</th>
<th>FIELD4</th>
<th>FIELD5</th>
<th>FIELD6</th>
<th>_ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>QPSData</td>
<td>.</td>
<td>.</td>
<td>1</td>
<td>.</td>
<td>2</td>
</tr>
<tr>
<td>ROWS</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>3</td>
<td>.</td>
<td>4</td>
</tr>
<tr>
<td>N</td>
<td>z</td>
<td>.</td>
<td>.</td>
<td>5</td>
<td>.</td>
<td>6</td>
</tr>
<tr>
<td>L</td>
<td>c1</td>
<td>.</td>
<td>.</td>
<td>7</td>
<td>.</td>
<td>7</td>
</tr>
<tr>
<td>G</td>
<td>c2</td>
<td>.</td>
<td>.</td>
<td>8</td>
<td>.</td>
<td>8</td>
</tr>
<tr>
<td>COLUMNS</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>9</td>
<td>.</td>
<td>9</td>
</tr>
<tr>
<td>x[1]</td>
<td>z</td>
<td>2.0</td>
<td>c1</td>
<td>1</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>x[1]</td>
<td>c2</td>
<td>1.0</td>
<td>.</td>
<td>8</td>
<td>.</td>
<td>11</td>
</tr>
<tr>
<td>x[2]</td>
<td>z</td>
<td>3.0</td>
<td>c1</td>
<td>-1</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>x[2]</td>
<td>c2</td>
<td>2.0</td>
<td>.</td>
<td>10</td>
<td>.</td>
<td>13</td>
</tr>
<tr>
<td>RHS</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>11</td>
<td>.</td>
<td>14</td>
</tr>
<tr>
<td>.RHS.</td>
<td>c1</td>
<td>1.0</td>
<td>.</td>
<td>12</td>
<td>.</td>
<td>15</td>
</tr>
<tr>
<td>.RHS.</td>
<td>c2</td>
<td>100.0</td>
<td>.</td>
<td>13</td>
<td>.</td>
<td>16</td>
</tr>
<tr>
<td>QSECTION</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>14</td>
<td>.</td>
<td>17</td>
</tr>
<tr>
<td>x[1]</td>
<td>x[1]</td>
<td>2.0</td>
<td>.</td>
<td>15</td>
<td>.</td>
<td>18</td>
</tr>
<tr>
<td>x[1]</td>
<td>x[2]</td>
<td>2.5</td>
<td>.</td>
<td>16</td>
<td>.</td>
<td>19</td>
</tr>
<tr>
<td>ENDTAQ</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 10.28** QPS Data Set Generated by the SAVE QPS Statement
SOLVE Statement

\[
\text{SOLVE [ WITH solver ] [( OBJECTIVE | OBJ ) name ] [( NOOBJECTIVE | NOOBJ ) ] [ RELAXINT ] [ / options ];}
\]

The SOLVE statement invokes a PROC OPTMODEL solver. The current model is first resolved to the numeric form that is required by the solver. The resolved model and possibly the current values of any optimization variables are passed to the solver. The solver executes in the distributed computing environment when you specify the SESSREF= or SESSUUID= option. After the solver finishes executing, the SOLVE statement prints a short table that shows a summary of results from the solver (see the section “ODS Table and Variable Names” on page 371) and updates the _OROPTMODEL_ macro variable.

Here are the arguments to the SOLVE statement:

- **solver**
  - selects the named solver: CLP, LP, LSO, MILP, NETWORK, NLP, or QP (see corresponding chapters in this book for details). If you do not specify a WITH clause, PROC OPTMODEL chooses a solver that depends on the problem type. Table 10.8 lists the default solver for each problem type.\(^1\)

- **name**
  - specifies the objective to use. This resets the objective list for the problem. You can abbreviate the OBJECTIVE keyword as OBJ. If you do not specify this argument, then the problem objectives are unchanged. You can also specify multiple objectives for the LSO solver by using a list in parentheses of names, separated by spaces, in place of name. Within this list, you can prefix each name by using an indexing set to specify a group of objectives.

- **NOOBJECTIVE**
  - requests that the solver ignore the objectives for the problem and use a constant zero objective instead. This keyword enables the solver to process the current model as a feasibility problem. You can abbreviate the NOOBJECTIVE keyword as NOOBJ.

- **RELAXINT**
  - requests that any integral variables be relaxed to be continuous. RELAXINT can be used with linear and nonlinear problems in addition to any solver.

- **options**
  - specifies solver options. You can specify solver options directly only when you use the WITH clause. A list of the options available with the solver is provided in the individual chapters that describe each solver. Table 10.9 lists the available option types. You can use

\(^{1}\)The OPTMODEL procedure never uses the LSO or network solver as a default. If the QP solver detects nonconvexity (nonconcavity) for a minimization (maximization) problem, then PROC OPTMODEL calls the NLP solver instead.

### Table 10.8 Default Solvers and Algorithms in PROC OPTMODEL

<table>
<thead>
<tr>
<th>Problem</th>
<th>Solver</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint programming</td>
<td>CLP</td>
<td>Constraint propagation and backtracking search</td>
</tr>
<tr>
<td>Linear programming</td>
<td>LP</td>
<td>Dual simplex</td>
</tr>
<tr>
<td>Mixed integer linear programming</td>
<td>MILP</td>
<td>Branch-and-cut</td>
</tr>
<tr>
<td>General nonlinear programming</td>
<td>NLP</td>
<td>Interior point NLP</td>
</tr>
<tr>
<td>Quadratic programming</td>
<td>QP</td>
<td>Interior point QP</td>
</tr>
</tbody>
</table>

\(^{1}\)The OPTMODEL procedure never uses the LSO or network solver as a default. If the QP solver detects nonconvexity (nonconcavity) for a minimization (maximization) problem, then PROC OPTMODEL calls the NLP solver instead.
an expression in parentheses in place of a literal option value for numeric and keyword options. A string expression is matched to a keyword. OPTMODEL parameters that are changed by the solver must be specified by a parameter or array option.

![Table 10.9](image)

<table>
<thead>
<tr>
<th>Type</th>
<th>Syntax</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>`option</td>
<td>NOoption`</td>
</tr>
<tr>
<td>Keyword</td>
<td><code>option=name</code></td>
<td><code>solve with lp / ALGORITHM=PS;</code></td>
</tr>
<tr>
<td>Numeric</td>
<td><code>option=number</code></td>
<td><code>solve with nlp / OPTTOL=1E-4;</code></td>
</tr>
<tr>
<td>Parameter</td>
<td><code>option=identifier-expression</code></td>
<td><code>solve with network / links=(INCLUDE=LINKS) concomp;</code></td>
</tr>
<tr>
<td>Array</td>
<td><code>option=array-name [.suffix]</code></td>
<td><code>solve with network / links=(WEIGHT=WEIGHT) tsp;</code></td>
</tr>
</tbody>
</table>

The SOLVE statement uses the value of the predeclared _SOLVER_OPTIONS_ and _solver_ OPTIONS_ string parameters to provide default solver options. Any options that are specified by these parameters are added before options that are specified in the SOLVE statement, with options from _SOLVER_OPTIONS_ appearing first. These options are included even when the SOLVE statement does not contain a WITH clause to specify a solver; in this case, _solver_ is the name of the default solver as shown in Table 10.8.

Initially the predeclared string parameters _SOLVER_OPTIONS_ and _solver_ OPTIONS_ (for each solver) are empty strings, but you can assign them. You must use keywords or literal values to specify option values in these strings. Redundant white space is allowed. For example, the following statements set up some simple defaults:

```plaintext
_SOLVER_OPTIONS_ = "MAXTIME = 600"; /* options for all solvers */
_LP_OPTIONS_ = "PRESOLVER=AGGRESSIVE"; /* options for LP solver */
```

Optimization techniques that use initial values obtain them from the current values of the optimization variables unless the NOINITVAR option is specified. When the solver finishes executing, the current value of each optimization variable is replaced by the optimal value found by the solver. These values can then be used as the initial values for subsequent solver invocations.

**Note:** If a solver fails, any currently pending statement is stopped and processing continues with the next complete statement read from the input. For example, if a SOLVE statement that is enclosed in a DO group (see the section “DO Statement” on page 311) fails, then the subsequent statements in the group are not executed and processing resumes at the point immediately following the DO group. Neither an infeasible result, an unbounded result, nor reaching an iteration limit is considered to be a solver failure.

**Note:** The information that appears in the macro variable _OROPTMODEL_ (see the section “Macro Variable _OROPTMODEL_” on page 408) varies by solver.

**Note:** The RELAXINT keyword is applied immediately before the problem is passed to the solver, after any processing by the PROC OPTMODEL presolver. So the problem presented to the solver might not be equivalent to the one produced by setting the .RELAX suffix of all variables to a nonzero value. In particular, the bounds of integer variables are still adjusted to be integral, and PROC OPTMODEL’s presolver might use integrality to tighten bounds further.
STOP Statement

\[\text{STOP ;}\]

The STOP statement halts the execution of all statements that contain it, including DO statements and other control or looping statements. Execution continues with the next top-level source statement. The following statements demonstrate a simple use of the STOP statement:

\begin{verbatim}
proc optmodel;
   number i, j;
   do i = 1..5;
      do j = 1..4;
         if i = 3 and j = 2 then stop;
      end;
   end;
   print i j;
\end{verbatim}

The output is shown in Figure 10.29.

\begin{figure}[h]
\centering
\begin{tabular}{|c|c|}
\hline
i & j \\
3 & 2 \\
\hline
\end{tabular}
\caption{STOP Statement: Output}
\end{figure}

When the counters \(i\) and \(j\) reach 3 and 2, respectively, the STOP statement terminates both loops. Execution continues with the PRINT statement.

SUBMIT Statement

\begin{verbatim}
SUBMIT arguments [ / options ] ;
SAS statements ;
ENDSUBMIT ;
\end{verbatim}

The SUBMIT statement allows SAS code to be executed before PROC OPTMODEL processing continues. For example, you can use the SUBMIT statement to invoke other SAS procedures to perform analysis or to display results. The following statements use PROC SORT to order a list of nodes by decreasing priority; the nodes can be used for further processing:

\begin{verbatim}
proc optmodel;
   set<str> NODES;
   num priority{NODES};

   /* set up priority data... */

   /* sort nodes by descending priority */
   create data temppri from [id] priority;
   submit;
      proc sort;
         by descending priority;
      run;
   endsubmit;
\end{verbatim}
/* load nodes by priority */
str nodesByPri{i in 1..card(NODES)};
read data temppri into [n_] nodesByPri=id;

/* use the sorted list... */

The SUBMIT statement must appear as the last or only statement on a line. It is followed by lines of SAS statements, terminated by the ENDSUBMIT statement on a line of its own. The SAS statements between the SUBMIT and ENDSUBMIT statements are referred to as a SUBMIT block. The SUBMIT block is sent to the SAS language processor each time the SUBMIT statement is executed.

The SUBMIT block can include SAS global statements and procedure and invocations. Macros are not expanded until the SUBMIT block is executed. So you can change macro variables to modify the behavior of the SUBMIT block each time it is processed.

The arguments list specifies macro variables to initialize before the SUBMIT block is executed. List items are separated by spaces. Each of the arguments takes one of the following forms:

- \texttt{name}
  - copies the value of the PROC OPTMODEL parameter \texttt{name} to the macro variable that has the same name.
- \texttt{name} = \texttt{identifier-expression}
  - copies the value of the PROC OPTMODEL parameter specified by \texttt{identifier-expression} to the macro variable \texttt{name}.
- \texttt{name} = \texttt{number} | \texttt{string} | \texttt{string}'
  - copies the value of the specified \texttt{number} or \texttt{string} constant to the macro variable \texttt{name}.
- \texttt{name} = ( \texttt{expression} )
  - copies the result of evaluating \texttt{expression} to the macro variable \texttt{name}.

The following statements use a SUBMIT argument to modify the output each time the SUBMIT block is invoked:

\begin{verbatim}
for {i in 1..5} do;
  submit a=i;
    %put Value of a is \&a..;
  endsubmit;
end;
\end{verbatim}

The options in the SUBMIT statement are used to retrieve status information after a SUBMIT block is executed. Each item in the space-delimited options list has one of the following forms:

- \texttt{OK} = \texttt{identifier-expression}
  - specifies a PROC OPTMODEL numeric parameter location, \texttt{identifier-expression}, that is updated to indicate the success of the SUBMIT block execution. The location is set to 1 if execution is successful or 0 if errors are detected. PROC OPTMODEL continues execution when the SUBMIT block encounters errors only if the OK= option is specified.
- \texttt{OUT} [ = ] \texttt{output-argument}
  - specifies a single \texttt{output-argument} for retrieving macro variable values after each execution of the block.
OUT [ = ] ( output-argument )

specifies a list of space-delimited output-arguments for retrieving macro variable values after the block is executed.

Each output-argument item specifies a macro variable to copy after the block is executed. Each item takes one of the following two forms:

identifier-expression

copies the macro variable specified by the name portion of the identifier-expression into the PROC OPTMODEL parameter location specified by identifier-expression.

identifier-expression = name

copies the macro variable specified by name into the PROC OPTMODEL parameter location specified by identifier-expression.

The following statements show how to use the options in the SUBMIT statement to retrieve the result of a SUBMIT block execution:

```
proc optmodel;
   num success, syscc;
   submit / OK = success out syscc;
       data example;
           set notfound;
           j = i*i;
       run;
   endsubmit;
   print success syscc;
```

The DATA step fails, so the success parameter is set to 0 and syscc is set to the error code in the &SYSCC macro variable. The output is shown in Figure 10.30.

![Figure 10.30 SUBMIT Statement Error Handling](image)

<table>
<thead>
<tr>
<th>success</th>
<th>syscc</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1012</td>
</tr>
</tbody>
</table>

**NOTE:** The SUBMIT block runs in the same environment that the OPTMODEL procedure is running in. The SUBMIT block can change the values for options, LIBNAME librefs, FILENAME filerefs, titles, footnotes, and macro variables. The OPTMODEL procedure sees these changes but might not process them until the next top-level source statement is read.

**NOTE:** A SUBMIT block can reset the ODS environment of the OPTMODEL procedure. For example, the ODS SELECT and EXCLUDE lists could be cleared after the SUBMIT block executes.

**NOTE:** A SUBMIT statement can appear only in open code. An error message is displayed if the SUBMIT statement is read from a macro. You can avoid this limitation by placing the SUBMIT statement, SUBMIT block, and ENDSUBMIT in a separate file and by using the %INCLUDE statement to include the file in the macro.
UNFIX Statement

`UNFIX variable-list [ = expression ];`

The UNFIX statement reverses the effect of FIX statements. The solver can vary the specified variables, variable arrays, or variable array locations specified by `variable-list`. The `variable-list` consists of one or more variable names separated by spaces.

Each variable name in the `variable-list` is an identifier expression (see the section “Identifier Expressions” on page 347). The UNFIX statement affects an entire variable array if the identifier expression omits the index from an array name. Each identifier-expression can be prefixed by an indexing set to specify the variable indices.

The `expression` specifies a new initial value that is stored in each element of the `variable-list`. If a single identifier-expression is specified, prefixed by an indexing set, then the `expression` is evaluated for each element in the indexing set. Otherwise the `expression` is evaluated exactly once.

The following example demonstrates the UNFIX command:

```optmodel
proc optmodel;
  var x{1..3};
  fix x;  /* fixes entire array to 0 */
  unfix x[1];  /* x[1] can now be varied again */
  unfix x[2] = 2;  /* x[2] is given an initial value 2 */
  unfix x;  /* all x indices can now be varied */
```

After the following statements are executed, the variables x[1] and x[2] are not fixed. They each hold the value 4. The variable x[3] is fixed at a value of 2.

```optmodel
proc optmodel;
  var x{1..3} init 2;
  num a = 1;
  fix x;
  unfix {1..2} x = a+3;  /* equivalent to prior UNFIX statement */
```

USE PROBLEM Statement

`USE PROBLEM identifier-expression ;`

The USE PROBLEM programming statement makes the problem specified by the `identifier-expression` be the current problem. If the problem has not been previously used, the problem is created using the PROBLEM declaration corresponding to the name. The problem must have been previously declared.
expression. This is a valid way to formulate the objective expression. However, in many cases it is
inconvenient to specify the numeric constants directly. Direct specification of numeric constants can also hide
the structure of the problem that is being solved. The objective expression text would need to be modified
when the numeric values in the problem change. This can be very inconvenient with large models.

In PROC OPTMODEL, you can create named numeric values that behave as constants in expressions. These
named values are called _parameters_. You can write an expression by using mnemonic parameter names in
place of numeric literals. This produces a clearer formulation of the optimization problem. You can easily
modify the values of parameters, define them in terms of other parameters, or read them from a SAS data set.

The model from this same example can be reformulated in a more general polynomial form, as follows:

```plaintext
data coeff;
  input c_xx c_x c_y c_xy c_yy;
datalines;
  1 -1 -2 -1 1;
;
proc optmodel;
  var x, y;
  number c_xx, c_x, c_y, c_xy, c_yy;
  read data coeff into c_xx c_x c_y c_xy c_yy;
  min z=c_xx*x**2 + c_x*x + c_y*y + c_xy*x*y + c_yy*y**2;
solve;
```

These statements read the coefficients from a data set, _COEFF_. The _NUMBER_ statement declares the
parameters. The _READ DATA_ statement reads the parameters from the data set. You can apply this model
easily to coefficients that you have generated by various means.

---

**Indexing**

Many models have large numbers of variables or parameters that can be categorized into families of similar
purpose or behavior. Such families of items can be compactly represented in PROC OPTMODEL by using
indexing. You can use indexing to assign each item in such families to a separate value location.

PROC OPTMODEL indexing is similar to array indexing in the DATA step, but it is more flexible. Index
values can be numbers or strings, and are not required to fit into some rigid sequence. PROC OPTMODEL
indexing is based on index sets, described further in the section “Index Sets” on page 348. For example, the
following statement declares an indexed parameter:

```plaintext
number p{1..3};
```

The construct that follows the parameter name _p_, “{1..3},” is a simple index set that uses a range expression
(see “Range Expression” on page 356). The index set contains the numeric members 1, 2, and 3. The
parameter has distinct value locations for each of the index set members. The first such location is referenced
as _p[1]_, the second as _p[2]_, and the third as _p[3]_.

The following statements show an example of indexing:

```plaintext
proc optmodel;
  number p{1..3};
p[1]=5;
p[2]=7;
p[3]=9;
put p[*]=;
```
The preceding statements produce a line such as the one shown in Figure 10.31 in the log.

**Figure 10.31** Indexed Parameter Output


Index sets can also specify local dummy parameters. A dummy parameter can be used as an operand in the expressions that are controlled by the index set. For example, the assignment statements in the preceding statements could be replaced by an initialization in the parameter declaration, as follows:

```
number p{i in 1..3} init 3 + 2*i;
```

The initialization value of the parameter location \( p[1] \) is evaluated with the value of the local dummy parameter \( i \) equal to 1. So the initialization expression \( 3 + 2*i \) evaluates to 5. Similarly for location \( p[2] \), the value of \( i \) is 2 and the initialization expression evaluates to 7.

The OPTMODEL modeling language supports aggregation operators that combine values of an expression where a local dummy parameter (or parameters) ranges over the members of a set. For example, the SUM aggregation operator combines expression values by adding them together. The following statements output 21, since \( p[1] + p[2] + p[3] = 5 + 7 + 9 = 21 \):

```
proc optmodel;
  number p{i in 1..3} init 3 + 2*i;
  put (sum{i in 1..3} p[i]);
```

Aggregation operators like SUM are especially useful in objective expressions because they can combine a large number of similar expressions into a compact representation. As an example, the following statements define a trivial least squares problem:

```
proc optmodel;
  number n init 100000;
  var x{1..n};
  min z = sum{i in 1..n}(x[i] - log(i)**2;
  solve;
```

The objective function in this case is

\[
z = \sum_{i=1}^{n} (x_i - \log i)^2
\]

Effectively, the objective expression expands to the following large expression:

```
min z = (x[1] - log(1))**2
  + (x[2] - log(2))**2
  + (x[3] - log(3))**2
  + (x[4] - log(4))**2
  + ... 
  + (x[100000] - log(100000))**2;
```

Even though the problem has 100,000 variables, the aggregation operator SUM enables a compact objective expression.
**NOTE:** PROC OPTMODEL classifies as mathematically impure any function that returns a different value each time it is called. The RAND function, for example, falls into this category. PROC OPTMODEL disallows impure functions inside array index sets, objectives, and constraint expressions. The values of expressions that are specified in the declaration of a parameter are resolved in a nondeterministic order during threaded problem generation. Therefore, the values are also nondeterministic when these expressions use impure functions.

**Types**

In PROC OPTMODEL, parameters and expressions can have numeric or character values. These correspond to the elementary types named NUMBER and STRING, respectively. The NUMBER type is the same as the SAS data set numeric type. The NUMBER type includes support for missing values. The STRING type is like the SAS VARCHAR(*) character type, which allows lengths to vary as needed up to \(2^{31} - 1\) bytes. However, string values are stored independently of parameters, and value references can be shared. The NUMBER and STRING types together are called the *scalar types*. You can abbreviate the type names as NUM and STR, respectively.

PROC OPTMODEL also supports set types for parameters and expressions. Sets represent collections of values of a member type, which can be a NUMBER, a STRING, or a vector of scalars (the latter is called a *tuple* and described in the following paragraphs). Members of a set all have the same member type. Members that have the same value are stored only once. For example, PROC OPTMODEL stores the set 2, 2, 2 as the set 2.

Specify a set of numbers with SET<NUMBER>. Similarly, specify a set of strings as SET<STRING>. A set can also contain a collection of tuples, all of the same fixed length. A *tuple* is an ordered collection that contains a fixed number of elements. Each element in a tuple contains a scalar value. In PROC OPTMODEL, tuples of length 1 are equivalent to scalars. Two tuples have equal values if the elements at corresponding positions in each tuple have the same value. Within a set of tuples, the element type at a particular position in each tuple is the same for all set members. The element types are part of the set type. For example, the following statement declares parts as a set of tuples that have a string in the first element position and a number in the second element position and then initializes its elements to be <R 1>, <R 2>, <C 1>, and <C 2>.

```plaintext
set<string,number> parts = /<R 1> <R 2> <C 1> <C 2>/;
```

To create a compact model, use sets to take advantage of the structure of the problem being modeled. For example, a model might contain various values that specify attributes for each member of a group of suppliers. You could create a set that contains members that represent each supplier. You can then model the attribute values by using arrays that are indexed by members of the set.

The section “Parameters” on page 343 has more details and examples.
Names

Names are used in the OPTMODEL modeling language to refer to various entities such as parameters or variables. Names must follow the usual rules for SAS names. Names can be up to 32 characters long and are not case sensitive. They must be declared before they are used.

Avoid declarations with names that begin with an underscore (_). These names can have special uses in PROC OPTMODEL.

Parameters

In the OPTMODEL modeling language, parameters are named locations that hold constant values. Parameter declarations specify the parameter type followed by a list of parameter names to declare. For example, the following statement declares numeric parameters named \( a \) and \( b \):

\[
\text{number } a, b;
\]

Similarly, the following statements declare a set \( s \) of strings, a set \( n \) of numbers, and a set \( sn \) of tuples:

\[
\begin{align*}
\text{set } \text{<string> } s; & \\
\text{set } \text{<number> } n; & \\
\text{set } \text{<string, number> } sn;
\end{align*}
\]

You can assign values to parameters in various ways. A parameter can be assigned a value with an assignment statement. For example, the following statements assign values to the parameter \( s \), \( n \), and \( sn \) in the preceding declaration:

\[
\begin{align*}
s & = \{ \text{'a'}, \text{'b'}, \text{'c'} \}; \\
n & = \{ 1, 2, 3 \}; \\
\text{sn} & = \{ \langle \text{'a'},1 \rangle, \langle \text{'b'},2 \rangle, \langle \text{'c'},3 \rangle \};
\end{align*}
\]

Parameter values can also be assigned using a READ DATA statement (see the section “READ DATA Statement” on page 326).

A parameter declaration can provide an explicit value. To specify the value, follow the parameter name with an equal sign (=) and an expression. The value expression can be written in terms of other parameters. The declared parameter takes on a new value each time a parameter that is used in the expression changes. This automatic value update is shown in the following example:

\[
\begin{align*}
\text{proc optmodel;} & \\
\text{number } \text{pi}=4*\text{atan}(1); & \\
\text{number } r; & \\
\text{number } \text{circum}=2*\text{pi}*r; & \\
r=1; & \\
\text{put } \text{circum}; & /* \text{prints 6.2831853072 */} \\
r=2; & \\
\text{put } \text{circum}; & /* \text{prints 12.566370614 */}
\end{align*}
\]

The automatic update of parameter values makes it easy to perform “what if” analysis since, after the solver finds a solution, you can change parameters and reinvoke the solver. You can easily examine the effects of the changes on the optimal values.
If you declare a set parameter that has only the SET type specifier, then the element type is determined from
the initialization expression. If the initialization expression is omitted or if the expression is an empty set,
then the set type defaults to SET<NUMBER>. For example, the following statement implicitly declares s1
as a set of numbers:

    set s1;

The following statement declares s2 as a set of strings:

    set s2 = {'A'};

You can declare an array parameter by following the parameter name with an index set specification (see the
section “Index Sets” on page 348). For example, declare an array of 10 numbers as follows:

    number c{1..10};

Individual locations of a parameter array can be referred to with an indexing expression. For example, you
can refer to the third location of parameter c as c[3]. Array index sets cannot be specified using a function
such as RAND that returns a different value each time it is called.

Parameter names must be declared before they are used. Nonarray names become available at the end of the
parameter declaration item. Array names become available after the index set specification. The latter case
permits some forms of recursion in the optional initialization expression that can be supplied for a parameter.

You do not need to assign values to parameters before they are referenced. Most information in PROC
OPTMODEL is stored symbolically and resolved when necessary. Values are resolved in certain statements.
For example, PROC OPTMODEL resolves a parameter used in the objective during the execution of a
SOLVE statement. If no value is available during resolution, then an error is diagnosed.

Expressions

Expressions are grouped into three categories based on the types of values they can produce: logical, set, and
scalar (that is, numeric or character).

Logical expressions test for a Boolean (true or false) condition. As in the DATA step, logical operators
produce a value equal to either 0 or 1. A value of 0 represents a false condition, while a value of 1 represents
a true condition.

Logical expression operators are not allowed in certain contexts due to syntactic considerations. For example,
in the VAR statement a logical operator might indicate the start of an option. Enclose a logical expression
in parentheses to use it in such contexts. The difference is illustrated by the output (Figure 10.32) of the
following statements, where two variables, x and y, are declared with initial values. The PRINT statement
and the EXPAND statement are used to check the initial values and the variable bounds, respectively.

    proc optmodel;
    var x init 0.5 >= 0 <= 1;
    var y init (0.5 >= 0) <= 1;
    print x y;
    expand;
Contexts that expect a logical expression also accept numeric expressions. In such cases zero or missing values are interpreted as false, and all nonzero nonmissing numeric values are interpreted as true.

Set expressions return a set value. PROC OPTMODEL supports a number of operators that create and manipulate sets. See the section “OPTMODEL Expression Extensions” on page 351 for a description of the various set expressions. Index-set syntax is described in the section “Index Sets” on page 348.

Scalar expressions are similar to the expressions in the DATA step except for PROC OPTMODEL extensions. PROC OPTMODEL provides an IF expression (described in the section “IF-THEN/ELSE Expression” on page 353). String lengths are assigned dynamically, so there is generally no padding or truncation of string values.

Table 10.10 shows the expression operators from lower to higher precedence (a higher precedence is given a larger number). Operators that have higher precedence are applied in compound expressions before operators that have lower precedence. The table also gives the order of evaluation that is applied when multiple operators of the same precedence are used together. Operators available in both PROC OPTMODEL and the DATA step have compatible precedences, except that in PROC OPTMODEL the NOT operator has a lower precedence than the relational operators. This means that, for example, \( \text{NOT } 1 < 2 \) is equal to \( \text{NOT } (1 < 2) \) (which is 0), rather than \( (\text{NOT } 1) < 2 \) (which is 1).

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Associativity</th>
<th>Operator</th>
<th>Alternates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logic Expression Operators</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Left to right</td>
<td>OR</td>
<td>!</td>
</tr>
<tr>
<td>2</td>
<td>Unary</td>
<td>OR{index-set}</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>AND{index-set}</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Left to right</td>
<td>AND</td>
<td>&amp;</td>
</tr>
<tr>
<td>4</td>
<td>Unary</td>
<td>NOT</td>
<td>~ ~</td>
</tr>
<tr>
<td>5</td>
<td>Left to right</td>
<td>&lt;</td>
<td>LT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;</td>
<td>GT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;=</td>
<td>LE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>&gt;=</td>
<td>GE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>=</td>
<td>EQ</td>
</tr>
<tr>
<td></td>
<td></td>
<td>~=</td>
<td>NE ^= ^=</td>
</tr>
<tr>
<td>6</td>
<td>Left to right</td>
<td>IN</td>
<td>NOT IN</td>
</tr>
<tr>
<td>7</td>
<td>Left to right</td>
<td>WITHIN</td>
<td>NOT WITHIN</td>
</tr>
</tbody>
</table>
### Table 10.10 (continued)

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Associativity</th>
<th>Operator</th>
<th>Alternates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Expression Operators</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Left to right</td>
<td>IF l THEN s1 ELSE s2</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Left to right</td>
<td>UNION</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>DIFF</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>SYMDIFF</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Unary</td>
<td>UNION(index-set)</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>Left to right</td>
<td>INTER</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>Unary</td>
<td>INTER(index-set)</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Left to right</td>
<td>CROSS</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>Unary</td>
<td>SETOF(index-set)</td>
<td>[ \ldots ]   TO  [ \ldots ] e BY</td>
</tr>
<tr>
<td></td>
<td>Right to left</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scalar Expression Operators</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Left to right</td>
<td>IF l THEN e</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>IF l THEN e1 ELSE e2</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>Left to right</td>
<td>+ -</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>Left to right</td>
<td>SUM(index-set)</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Unary</td>
<td>PROD(index-set)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MIN(index-set)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>MAX(index-set)</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>Left to right</td>
<td>+ /</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>Unary</td>
<td>+ -</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Right to left</td>
<td>&gt;&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>&lt;&gt;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>^</td>
<td></td>
</tr>
</tbody>
</table>

*Primary expressions* are the individual operands that are combined using the expression operators. Simple primary expressions can represent constants or named parameter and variable values. More complex primary expressions can be used to call functions or construct sets.

### Table 10.11  Primary Expression Table

<table>
<thead>
<tr>
<th>Expression</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier-expression</td>
<td>Parameter/variable reference; see the section “Identifier Expressions” on page 347</td>
</tr>
<tr>
<td>name (argument-list)</td>
<td>Function call; argument-list is 0 or more expressions separated by commas</td>
</tr>
<tr>
<td>n</td>
<td>Numeric constant</td>
</tr>
<tr>
<td>. or .c</td>
<td>Missing value constant</td>
</tr>
<tr>
<td>&quot;string&quot; or ‘string’</td>
<td>String constant</td>
</tr>
<tr>
<td>{ member-list }</td>
<td>Set constructor; member-list is 0 or more scalar expressions or tuple expressions separated by commas</td>
</tr>
</tbody>
</table>
### Identifier Expressions

Use an *identifier-expression* to refer to a variable, objective, constraint, parameter or problem location in expressions or initializations. This is the syntax for *identifier-expressions*:

\[
\text{name} \left[ \left[ \text{expression-1}, \ldots, \text{expression-n} \right] \right] \left[ . \text{suffix} \left[ \left[ \text{expression} \right] \right] \right]
\]

To refer to a location in an array, follow the array *name* with a list of scalar expressions in square brackets ([ ]). The expression values are compared to the index set that was used to declare *name*. If there is more than one expression, then the values are formed into a tuple. The expression values for a valid array location must match a member of the array’s index set. For example, the following statements define a parameter array *A* that has two valid indices that match the tuples <1,2> and <3,4>:

```optmodel
proc optmodel;
    set<number, number> ISET = {<1,2>, <3,4>};
    number A(ISET);
    a[1,2] = 0; /* OK */
    a[3,2] = 0; /* invalid index */
```

The first assignment is valid with this definition of the index set, but the second fails because <3,2> is not a member of the set parameter ISET.

Specify a *suffix* to refer to auxiliary locations for variables or objectives. For more information, see the section “Suffixes” on page 380. Certain suffixes can be followed by a numeric index *expression* that selects a particular solution saved by the SOLVE statement. For more information about solution indices, see the section “Multiple Solutions” on page 398.

### Function Expressions

Most functions that can be invoked from the DATA step or the `%SYSFUNC` macro can be used in PROC OPTMODEL expressions. Certain functions are specific to the DATA step and cannot be used in PROC OPTMODEL. Functions specific to the DATA step include these:

- functions in the LAG, DIF, and DIM families
- functions that access the DATA step program data vector
functions that access symbol attributes

The CALL statement can invoke SAS library subroutines. These subroutines can read and update the values of the parameters and variables that are used as arguments. See the section “CALL Statement” on page 298 for an example.

OPTMODEL arrays can be passed to SAS library functions and subroutines using one of the following argument syntax forms:

\[
\text{OF array-name[*] [ . suffix [ [ expression | * ] ] ]}
\]
\[
\text{OF name . suffix [*]}
\]
\[
\text{OF array-name [ expression-1 [, … expression-n ] ] . suffix [*]}
\]

The \textit{array-name} is the name of an array symbol. The optional \textit{suffix} allows auxiliary values to be referenced, as described in section “Suffixes” on page 380. You can also use an \textit{identifier-expression} with a suffix that allows a solution index, specifying an asterisk (*) in place of the index \textit{expression}.

The OF array form is a compact alternative to listing the array or solution elements explicitly. The OF argument form is resolved into a sequence of arguments, one for each index in the array or for each solution index. The array elements appear in order of the array’s index set. Solutions appear in solution index order. If you use an asterisk for both the array and solution index lists, an argument appears for each combination of array and solution indices, and the solutions for each array index are placed in contiguous arguments.

As an example, the following statements use the CALL SORTN function to sort the elements of a numeric array:

\begin{verbatim}
proc optmodel;
  number original{i in 1..8} = sin(i);
  number sorted{i in 1..8} init original[i];
  call sortn(of sorted[*]);
  print original sorted;
\end{verbatim}

The output is shown in Figure 10.33. Eight arguments are passed to the SORTN routine. The original column shows the original order, and the sorted column has the sorted order.

\begin{center}
\textbf{Figure 10.33} Sorting Using an OF Array Argument
\end{center}

\begin{verbatim}
1 0.84147 -0.95892
2 0.90930 -0.75680
3 0.14112 -0.27942
4 -0.75680 0.14112
5 -0.95892 0.65699
6 -0.27942 0.84147
7 0.65699 0.90930
8 0.98936 0.98936
\end{verbatim}

\begin{center}
\textbf{Index Sets}
\end{center}

An index set represents a set of combinations of members from the component set expressions. The index set notation is used in PROC OPTMODEL to describe collections of valid array indices and to specify sets of
Nil Values

The evaluation of certain PROC OPTMODEL expressions can produce a special value, the nil value, that is distinct from any other value of the expression’s type. A nil value represents the lack of any ordinary value. You can use nil values to check for undefined parameters or to reset the effects of previous assignments or initializations.

Nil values are allowed for the following expressions:

- the operands of an equality comparison operator, = or ≠. Nil values are equal only to nil values.
- the second and third operands of an IF-THEN/ELSE expression
- the right-hand-side expression of an assignment statement. Assigning a nil value resets the effect of previous initializations or assignments of the target location.
- a parenthesized value expression for a PUT statement. Nil values are skipped on output.
• a parenthesized value expression for a PRINT statement. Nil values create an empty cell in the output.

**Note:** The initializers list from the INIT [ initializers ] clause for an array parameter declaration is evaluated at most once, except for any default element value. If an array element is assigned a nil value after the initializers are evaluated, then the default element value is used.

The _NIL_ keyword evaluates to a nil value in expressions. Nil values can also be generated when an identifier expression references an undefined parameter value. An identifier expression evaluates to a nil value instead of reporting an error when it is used as the operand of an equality comparison or as the right-hand-side expression of an assignment statement with a NUMBER, STRING, or SET parameter target. The nil value is generated when the identifier expression is used either directly or via the second or third operand of an IF-THEN/ELSE expression.

**Note:** Undefined parameter values are treated as errors when an identifier expression is used as the operand of a compound comparison (like b in (0 ~< b <= 10)) where one of the operators that are using the value is not an equality comparison.

The following code demonstrates some uses of nil values:

```
proc optmodel;
  num a{1..2};
  a[1] = 1;
  put (a[1] = _nil_)=; /* false, defined */
  put (a[2] = _nil_)=; /* true, undefined */
  put ((if a[1] then _nil_ else 2) = a[1])= /* false */;
  put ((if a[1] then _nil_ else 2) = a[2])= /* true */;
  a[1] = a[2]; /* now a[1] is undefined */
  put (a[1] = _nil_)=; /* true, undefined */

  /* array initialization list is evaluated at most once */
  num b{1..3} init [1 2 3 [*] 7];
  put b[*]; /* evaluates the initialization list */
  put b[*]; /* b[1] replaced by the default value */

  var x init (rand('UNIFORM')) >= 0 <= 10;
  put x=; /* initialized to a random value */
  x = _nil_; /* discard the current value */
  put x=; /* initialized to a new random value */
  put x.ub=; /* the declared bound */
  x.ub = 1; /* override the declared bound */
  put x.ub=; /* the overridden bound */
  x.ub = _nil_; /* reverse the override */
  put x.ub=; /* the declared bound again */

  /* nil values print as empty cells */
  print {i in 1..3, j in 1..3}(if i+j-1=3 then _nil_ else 1/(i+j-1));
```

This code produces the output in Figure 10.34.
OPTMODEL Expression Extensions

PROC OPTMODEL defines several new types of expressions for the manipulation of sets. Aggregation operators combine values of an expression that is evaluated over the members of an index set. Other operators create new sets by combining existing sets, or they test relationships between sets. PROC OPTMODEL also supports an IF expression operator that can conditionally evaluate expressions. These and other such expressions are described in this section.

AND Aggregation Expression

\[ \text{AND} \{ \text{index-set} \} \text{ logic-expression} \]

The AND aggregation operator evaluates the logical expression \text{logic-expression} jointly for each member of the index set \text{index-set}. The index set enumeration finishes early if the \text{logic-expression} evaluation produces a false value (zero or missing). The expression returns 0 if a false value is found or returns 1 otherwise. The following statements demonstrate both a true and a false result:

```
proc optmodel;
  put (and{i in 1..5} i < 10); /* returns 1 */
  put (and{i in 1..5} i NE 3); /* returns 0 */
```

CARD Function

\[ \text{CARD} \left( \text{set-expression} \right) \]

The CARD function returns the number of members of its set operand. For example, the following statements produce the output 3 since the set has 3 members:

```
proc optmodel;
  put (card(1..3));
```
CROSS Expression

\[ \text{set-expression-1 CROSS set-expression-2} \]

The CROSS expression returns the crossproduct of its set operands. The result is the set of tuples formed by concatenating the tuple value of each member of the left operand with the tuple value of each member of the right operand. Scalar set members are treated as tuples of length 1. The following statements demonstrate the CROSS operator:

```plaintext
proc optmodel;
  set s1 = 1..2;
  set<string> s2 = {'a', 'b'};
  set<number, string> s3 = s1 cross s2;
  put 's3 is ' s3;
  set<number, string, number> s4 = s3 cross 4..5;
  put 's4 is ' s4;
end;
```

This code produces the output in Figure 10.35.

![Figure 10.35 CROSS Expression Output](image)

```
s3 is {<1, 'a'>, <1, 'b'>, <2, 'a'>, <2, 'b'>}
s4 is {<1, 'a', 4>, <1, 'a', 5>, <1, 'b', 4>, <1, 'b', 5>, <2, 'a', 4>, <2, 'a', 5>, <2, 'b', 4>, <2, 'b', 5>}
```

DIFF Expression

\[ \text{set-expression-1 DIFF set-expression-2} \]

The DIFF operator returns a set that contains the set difference of the left and right operands. The result set contains values that are members of the left operand but not members of the right operand. The operands must have compatible set types. The following statements evaluate and print a set difference:

```plaintext
proc optmodel;
  put ({1,3} diff {2,3}); /* outputs {1} */
end;
```

Equality Expression

\[ \text{set-expression-1 = set-expression-2} \]
\[ \text{set-expression-1 ~ set-expression-2} \]

The equality comparison operators are extended to support the comparison of sets. Sets are considered equal if they have the same members. The following code demonstrates set comparison:

```plaintext
proc optmodel;
  put (/a b/ = /a b/); /* true */
  put (/a b/ = /a/); /* false */
  put ({}) = /a b/); /* false */
  put (1..3 = {1,3,2}); /* true, ignores order */
end;
```
IF-THEN/ELSE Expression

IF  logic-expression  THEN  expression-2  [ ELSE  expression-3 ]

The IF-THEN/ELSE expression evaluates the logical expression logic-expression and returns the result of evaluating the second or third operand expression according to the logical test result. If the logic-expression is true (nonzero and nonmissing), then the result of evaluating expression-2 is returned. If the logic-expression is false (zero or missing), then the result of evaluating expression-3 is returned. The other subexpression that is not selected is not evaluated.

An ELSE clause is matched during parsing with the nearest IF-THEN clause that does not have a matching ELSE. The ELSE clause can be omitted for numeric expressions; the resulting IF-THEN is handled as if a default ELSE 0 clause were supplied.

Use the IF-THEN/ELSE expression to handle special cases in models. For example, an inventory model based on discrete time periods might require special handling for the first or last period. In the following example the initial inventory for the first period is assumed to be fixed:

```plaintext
proc optmodel;
  number T;
  var inv{1..T}, order{1..T};
  number sell{1..T};
  number inv0;
  . . .
  /* balance inventory flow */
  con iflow{i in 1..T}:
    inv[i] = order[i] - sell[i] +
    if i=1 then inv0 else inv[i-1];
  . . .
```

The IF-THEN/ELSE expression in the example models the initial inventory for a time period i. Usually the inventory value is the inventory at the end of the previous period, but for the first time period the inventory value is given by the inv0 parameter. The iflow constraints are linear because the IF-THEN/ELSE test subexpression does not depend on variables and the other subexpressions are linear.

IF-THEN/ELSE can be used as either a set expression or a scalar expression. The type of expression depends on the subexpression between the THEN and ELSE keywords. The type used affects the parsing of the subexpression that follows the ELSE keyword because the set form has a lower operator precedence. For example, the following two expressions are equivalent because the numeric IF-THEN/ELSE has a higher precedence than the range operator (..):

```plaintext
IF logic THEN 1 ELSE 2 .. 3

(IF logic THEN 1 ELSE 2) .. 3
```

But the set form of IF-THEN/ELSE has lower precedence than the range expression operator. So the following two expressions are equivalent:

```plaintext
IF logic THEN 1 .. 2 ELSE 3 .. 4

IF logic THEN (1 .. 2) ELSE (3 .. 4)
```

The IF-THEN and IF-THEN/ELSE operators always have higher precedence than the logic operators. So, for example, the following two expressions are equivalent:
IF logic THEN numeric1 < numeric2

(IF logic THEN numeric1) < numeric2

It is best to use parentheses when in doubt about precedence.

**IN Expression**

expression IN set-expression

expression NOT IN set-expression

The IN expression returns 1 if the value of the left operand is a member of the right operand set. Otherwise, the IN expression returns 0. The NOT IN operator logically negates the returned value. Unlike the DATA step, the right operand is an arbitrary set expression. The left operand can be a tuple expression. The following example demonstrates the IN and NOT IN operators:

```
proc optmodel;
set s = 1..10;
put (5 in s); /* outputs 1 */
put (-1 not in s); /* outputs 1 */
set<num, str> t = {<1,'a'>, <2,'b'>, <2,'c'>};
put (<2, 'b'> in t); /* outputs 1 */
put (<1, 'b'> in t); /* outputs 0 */
```

**Index Set Expression**

{ index-set }

The index set expression returns the set of members of an index set. This expression is distinguished from a set constructor (see the section “Set Constructor Expression” on page 357) because it contains a list of set expressions.

The following statements use an index set with a selection expression that excludes the value 3:

```
proc optmodel;
put ({i in 1..5 : i NE 3}); /* outputs {1,2,4,5} */
```

**INTER Expression**

set-expression-1 INTER set-expression-2

The INTER operator returns a set that contains the intersection of the left and right operands. This is the set that contains values that are members of both operand sets. The operands must have compatible set types.

The following statements evaluate and print a set intersection:

```
proc optmodel;
put ({1,3} inter {2,3}); /* outputs {3} */
```
INTER Aggregation Expression

    INTER { index-set } set-expression

The INTER aggregation operator evaluates the set-expression for each member of the index set index-set. The result is the set that contains the intersection of the set of values that were returned by the set-expression for each member of the index set. An empty index set causes an expression evaluation error.

The following statements use the INTER aggregation operator to compute the value of \{1,2,3,4\} \cap \{2,3,4,5\} \cap \{3,4,5,6\}:

    proc optmodel;
    put (inter{i in 1..3} i..i+3); /* outputs \{3,4\} */

MAX Aggregation Expression

    MAX { index-set } expression

The MAX aggregation operator evaluates the scalar-valued expression expression for each member of the index set index-set. The result is the maximum of the values that are returned by the expression. Missing numeric values are handled with the SAS numeric sort order; a missing value is treated as smaller than any nonmissing numeric value. If the index set is empty and the expression is numeric, then the result is the negative number that has the largest absolute value representable on the machine. Expression evaluation produces an error if the index set is empty and the expression has a string value.

The following example produces the outputs 0.5 and *****:

    proc optmodel;
    put (max{i in 2..5} 1/i);
    put (max{i in 2..5} repeat('*',i-1));

MIN Aggregation Expression

    MIN { index-set } expression

The MIN aggregation operator evaluates the scalar-valued expression expression for each member of the index set index-set. The result is the minimum of the values that are returned by the expression. Missing numeric values are handled with the SAS numeric sort order; a missing value is treated as smaller than any nonmissing numeric value. If the index set is empty and the expression is numeric, then the result is the largest positive number representable on the machine. Expression evaluation produces an error if the index set is empty and the expression has a string value.

The following example produces the outputs 0.2 and **:

    proc optmodel;
    put (min{i in 2..5} 1/i);
    put (min{i in 2..5} repeat('*',i-1));

Nil Constant Expression

    _NIL_

The _NIL_ keyword evaluates to a nil value. For more information about nil values, see the section “Nil Values” on page 349.
OR Aggregation Expression

\[
\text{OR} \{ \text{index-set} \} \text{logic-expression}
\]

The OR aggregation operator evaluates the logical expression \text{logic-expression} for each member of the index set \text{index-set}. The index set enumeration finishes early if the \text{logic-expression} evaluation produces a true value (nonzero and nonmissing). The result is 1 if a true value is found, or 0 otherwise. The following statements demonstrate both a true and a false result:

```plaintext
proc optmodel;
  put (or{i in 1..5} i = 2); /* returns 1 */
  put (or{i in 1..5} i = 7); /* returns 0 */
```

PROD Aggregation Expression

\[
\text{PROD} \{ \text{index-set} \} \text{expression}
\]

The PROD aggregation operator evaluates the numeric expression \text{expression} for each member of the index set \text{index-set}. The result is the product of the values that are returned by the \text{expression}. This operator is analogous to the $\prod$ operator used in mathematical notation. If the index set is empty, then the result is 1.

The following example uses the PROD operator to evaluate a factorial:

```plaintext
proc optmodel;
  number n = 5;
  put (prod{i in 1..n} i); /* outputs 120 */
```

Range Expression

\[
\text{expression-1 .. expression-n} \ [\ \text{BY} \ \text{expression}]\]

The range expression returns the set of numbers from the specified arithmetic progression. The sequence proceeds from the left operand value up to the right operand limit. The increment between numbers is 1 unless a different value is specified with a BY clause. If the increment is negative, then the progression is from the left operand down to the right operand limit. The result can be an empty set.

For compatibility with the DATA step iterative DO loop construct, the keyword TO can substitute for the range (..) operator.

The limit value is not included in the resulting set unless it belongs in the arithmetic progression. For example, the following range expression does not include 30:

```plaintext
proc optmodel;
  put (10..30 by 7); /* outputs {10,17,24} */
```

The actual numbers that the range expression “f..l by i” produces are in the arithmetic sequence

\[ f, f + i, f + 2i, \ldots, f + ni \]

where

\[
n = \left\lfloor \frac{l - f}{i} + \sqrt{\epsilon} \right\rfloor
\]

and $\epsilon$ represents the relative machine precision. The limit is adjusted to avoid arithmetic roundoff errors.
PROC OPTMODEL represents the set specified by a range expression compactly when the value is stored in a parameter location, used as a set operand of an IN or NOT IN expression, used by an iterative DO loop, or used in an index set. For example, the following expression is evaluated efficiently:

$$999998.5 \text{ IN } 1..1000000000$$

**Set Constructor Expression**

$$\{ [ \text{expression-1}[..\text{expression-n}]]\}$$

The set constructor expression returns the set of the expressions in the member list. Duplicated values are added to the set only once. A warning message is produced when duplicates are detected. The constructor expression consists of zero or more subexpressions of the same scalar type or of tuple expressions that match in length and in element types.

The following statements output a three-member set and warn about the duplicated value 2:

```plaintext
proc optmodel;
  put {{1,2,3,2}}; /* outputs {1,2,3} */

proc optmodel;
  number m = 3, n = 4;
  var x{1..4} init 1;
  string y = 'c';
  put {{<'a', x[3]>, <'b', m>, <y, m/n>}};
```

**Figure 10.36** Set Constructor Expression Output

$\{\text{a}',1\text{,}'b',3\text{,}'c',0.75\}$

**Set Literal Expression**

$/ \text{members} /$

The set literal expression provides compact specification of simple set values. It is equivalent in function to the set constructor expression but minimizes typing for sets that contain numeric and string constant values. The set members are specified by members, which are literal values. As with the set constructor expression, each member must have the same type.

The following statement specifies a simple numeric set:

$/1 \text{ 2.5 } 4/$

The set contains the members 1, 2.5, and 4. A string set could be specified as follows:

$/\text{Miami 'San Francisco' Seattle 'Washington, D.C.'}/$

This set contains the strings ‘Miami’, ‘San Francisco’, ‘Seattle’, and ‘Washington, D.C.’. You can specify string values in set literals without quotation marks when the text follows the rules for a SAS name. Strings that begin with a digit or contain blanks or other special characters must be specified with quotation marks.
Specify tuple members of a set by enclosing the tuple elements within angle brackets (<elements>). The tuple elements can be specified with numeric and string literals. The following example includes the tuple elements <‘New York’, 4.5> and <‘Chicago’, -5.7>:

/<'New York' 4.5> <Chicago -5.7>/

**SETOF Aggregation Expression**

SETOF { index-set } expression

The SETOF aggregation operator evaluates the expression expression for each member of the index set index-set. The result is the set that is formed by collecting the values returned by the operand expression. The operand can be a tuple expression. For example, the following statements produce a set of tuples of numbers with their squared and cubed values:

```plaintext
proc optmodel;
put (setof{i in 1..3}<i, i*i, i**3>);
```

Figure 10.37 shows the displayed output.

**Figure 10.37** SETOF Aggregation Expression Output

{<1,1,1>,<2,4,8>,<3,9,27>}

**SLICE Expression**

SLICE ( < element-1,… element-n >, set-expression )

The SLICE expression produces a new set by selecting members in the operand set that match a pattern tuple. The pattern tuple is specified by the element list in angle brackets. Each element in the pattern tuple must specify a numeric or string expression. The expressions are used to match the values of the corresponding elements in the operand set member tuples. You can also specify an element by using an asterisk (*). The sequence of element values that correspond to asterisk positions in each matching tuple is combined into a tuple of the result set. At least one asterisk element must be specified.

The following statements demonstrate the SLICE expression:

```plaintext
proc optmodel;
put (slice(<1,*>, {<1,3>, <1,0>, <3,1>}));
put (slice(<*,2,*>, {<1,2,3>, <2,4,3>, <2,2,5>}));
```

These statements produce the output in Figure 10.38.

**Figure 10.38** SLICE Expression Output

{<3,0>
{<1,3>,<2,5>}

For the first PUT statement, <1,*> matches set members <1,3> and <1,0> but not <3,1>. The second element of each matching set tuple, corresponding to the asterisk element, becomes the value of the resulting set member. In the second PUT statement, the values of the first and third elements of the operand set member tuple are combined into a two-position tuple in the result set.
The following statements use the SLICE expression to help compute the transitive closure of a set of tuples representing a relation by using Warshall’s algorithm. In these statements the set parameter \( \text{dep} \) represents a direct dependency relation.

```optmodel
proc optmodel;
  set<str,str> dep = {<'B','A'>, <'C','B'>, <'D','C'>};
  set<str,str> cl;
  set<str> cn;
  cl = dep;
  cn = (setof{<i,j> in dep} i) inter (setof{<i,j> in dep} j);
  for {node in cn}
    cl = cl union (slice(<*,node>,cl) cross slice(<node,*>,cl));
  put cl;
```

The local dummy parameter \( \text{node} \) in the FOR statement iterates over the set \( \text{cn} \) of possible intermediate nodes that can connect relations transitively. At the end of each FOR iteration, the set parameter \( \text{cl} \) contains all tuples from the original set in addition to all transitive tuples found in the current or previous iterations.

The output in Figure 10.39 includes the indirect and direct transitive dependencies from the set \( \text{dep} \).

![Warshall's Algorithm Output](image)

A special form of \textit{index-set-item} uses the SLICE expression implicitly. See the section “More on Index Sets” on page 406 for details.

**SUM Aggregation Expression**

\[
\text{SUM} \{ \text{index-set} \} \text{ expression}
\]

The SUM aggregation operator evaluates the numeric expression \( \text{expression} \) for each member in the index set \( \text{index-set} \). The result is the sum of the values that are returned by the \( \text{expression} \). If the index set is empty, then the result is 0. This operator is analogous to the \( \sum \) operator that is used in mathematical notation. The following statements demonstrate the use of the SUM aggregation operator:

```optmodel
proc optmodel;
  put (sum {i in 1..10} i); /* outputs 55 */
```

**SYMDIFF Expression**

\[
\text{set-expression-1 SYMDIFF set-expression-2}
\]

The SYMDIFF expression returns the symmetric set difference of the left and right operands. The result set contains values that are members of either the left or right operand but are not members of both operands. The operands must have compatible set types.

The following example demonstrates a symmetric difference:

```optmodel
proc optmodel;
  put ((1,3) symdiff (2,3)); /* outputs (1,2) */
```
**Tuple Expression**

\[< \text{expression-1}, \ldots, \text{expression-n}>\]

A tuple expression represents the value of a member in a set of tuples. Each scalar subexpression inside the angle brackets represents the value of a tuple element. This form is used only with IN, SETOF, set constructor, and relational expressions.

The following statements demonstrate the tuple expression:

```optmodel
proc optmodel;
  put (<1,2,3> in setof{i in 1..2}<i,i+1,i+2>);
  put ({<1,'a'>, <2,'b'>} cross {<3,'c'>, <4,'d'>});
```

The first PUT statement checks whether the tuple \(<1, 2, 3>\) is a member of a set of tuples. The second PUT statement outputs the crossproduct of two sets of tuples that are constructed by the set constructor.

These statements produce the output in **Figure 10.40**.

**Figure 10.40** Tuple Expression Output

```
{<1,'a',3,'c'>,<1,'a',4,'d'>,<2,'b',3,'c'>,<2,'b',4,'d'>}
```

You can specify tuple expressions as operands of the relational operators. For the equality operators (\(=\) and \(\sim=\)), tuples are equal if their corresponding elements are equal. Ordering operators (\(<\), \(\leq\), \(>\), and \(\geq\)) compare tuples lexicographically. That is, the corresponding tuple elements are compared left to right, and the leftmost pair of unequal elements determines the result of the comparison.

The following code demonstrates tuple comparison:

```optmodel
proc optmodel;
  put (<1,'a'> = <1,'a'>)=; /* true, all elements equal */
  put (<1,'a'> = <1,'b'>)=; /* false, at least one element unequal */
  put (<1,'a'> ~= <1,'b'>)=; /* true, at least one element unequal */
  put (<1,2,5> < <1,3,4>)=; /* true, compared left to right, 2 < 3 */
```

These statements produce the output in **Figure 10.41**.

**Figure 10.41** Tuple Comparison Output

```
<1,'a'> = <1,'a'>=1
<1,'a'> = <1,'b'>=0
<1,'a'> ~= <1,'b'>=1
<1,2,5> < <1,3,4>=1
```

**UNION Expression**

\[
\text{set-expression-1 UNION set-expression-2}
\]

The UNION expression returns the set union of the left and right operands. The result set contains values that are members of either the left or right operand. The operands must have compatible set types. The following example performs a set union:
proc optmodel;
    put ({1,3} union {2,3}); /* outputs {1,3,2} */

UNION Aggregation Expression

    UNION { index-set } set-expression

The UNION aggregation expression evaluates the set-expression for each member of the index set index-set. The result is the set union of the values that are returned by the set-expression. If the index set is empty, then the result is an empty set.

The following statements demonstrate a UNION aggregation. The output is the value of \{1,2,3,4\} ∪ \{2,3,4,5\} ∪ \{3,4,5,6\}.

    proc optmodel;
        put (union{i in 1..3} i..i+3); /* outputs {1,2,3,4,5,6} */

WITHIN Expression

    set-expression-1 WITHIN set-expression-2
    set-expression NOT WITHIN set-expression

The WITHIN expression returns 1 if the left operand set is a subset of the right operand set and returns 0 otherwise. (That is, the operator returns true if every member of the left operand set is a member of the right operand set.) The NOT WITHIN form logically negates the result value. The following statements demonstrate the WITHIN and NOT WITHIN operators:

    proc optmodel;
        put ({1,3} within {2,3}); /* outputs 0 */
        put ({1,3} not within {2,3}); /* outputs 1 */
        put ({1,3} within {1,2,3}); /* outputs 1 */

Conditions of Optimality

Linear Programming

A standard linear program has the following formulation:

\[
\begin{align*}
\text{minimize} & \quad c^T x \\
\text{subject to} & \quad Ax \geq b \\
& \quad x \geq 0
\end{align*}
\]

where

\[x \in \mathbb{R}^n\] is the vector of decision variables
\[A \in \mathbb{R}^{m \times n}\] is the matrix of constraints
\[c \in \mathbb{R}^n\] is the vector of objective function coefficients
\[b \in \mathbb{R}^m\] is the vector of constraints right-hand sides (RHS)
This formulation is called the primal problem. The corresponding dual problem (see the section “Dual Values” on page 385) is

\[
\begin{align*}
\text{maximize} & \quad b^T y \\
\text{subject to} & \quad A^T y \leq c \\
& \quad y \geq 0
\end{align*}
\]

where \( y \in \mathbb{R}^m \) is the vector of dual variables.

The vectors \( x \) and \( y \) are optimal to the primal and dual problems, respectively, only if there exist primal slack variables \( s = Ax - b \) and dual slack variables \( w = A^T y - c \) such that the following Karush-Kuhn-Tucker (KKT) conditions are satisfied:

\[
\begin{align*}
Ax + s &= b, & x \geq 0, & s \geq 0 \\
A^T y + w &= c, & y \geq 0, & w \geq 0 \\
s^T y &= 0 \\
w^T x &= 0
\end{align*}
\]

The first line of equations defines primal feasibility, the second line of equations defines dual feasibility, and the last two equations are called the complementary slackness conditions.

**Nonlinear Programming**

To facilitate discussion of optimality conditions in nonlinear programming, you write the general form of nonlinear optimization problems by grouping the equality constraints and inequality constraints. You also write all the general nonlinear inequality constraints and bound constraints in one form as “\( \geq \)” inequality constraints. Thus, you have the following formulation:

\[
\begin{align*}
\text{minimize} & \quad f(x) \\
\text{subject to} & \quad c_i(x) = 0, \quad i \in \mathcal{E} \\
& \quad c_i(x) \geq 0, \quad i \in \mathcal{I}
\end{align*}
\]

where \( \mathcal{E} \) is the set of indices of the equality constraints, \( \mathcal{I} \) is the set of indices of the inequality constraints, and \( m = |\mathcal{E}| + |\mathcal{I}| \).

A point \( x \) is feasible if it satisfies all the constraints \( c_i(x) = 0, i \in \mathcal{E} \) and \( c_i(x) \geq 0, i \in \mathcal{I} \). The feasible region \( \mathcal{F} \) consists of all the feasible points. In unconstrained cases, the feasible region \( \mathcal{F} \) is the entire \( \mathbb{R}^n \) space.

A feasible point \( x^* \) is a local solution of the problem if there exists a neighborhood \( \mathcal{N} \) of \( x^* \) such that

\[
f(x) \geq f(x^*) \text{ for all } x \in \mathcal{N} \cap \mathcal{F}
\]

Further, a feasible point \( x^* \) is a strict local solution if strict inequality holds in the preceding case; that is,

\[
f(x) > f(x^*) \text{ for all } x \in \mathcal{N} \cap \mathcal{F}
\]

A feasible point \( x^* \) is a global solution of the problem if no point in \( \mathcal{F} \) has a smaller function value than \( f(x^*) \); that is,

\[
f(x) \geq f(x^*) \text{ for all } x \in \mathcal{F}
\]
**Unconstrained Optimization**

The following conditions hold true for unconstrained optimization problems:

- **First-order necessary conditions**: If \( x^* \) is a local solution and \( f(x) \) is continuously differentiable in some neighborhood of \( x^* \), then
  \[
  \nabla f(x^*) = 0
  \]

- **Second-order necessary conditions**: If \( x^* \) is a local solution and \( f(x) \) is twice continuously differentiable in some neighborhood of \( x^* \), then \( \nabla^2 f(x^*) \) is positive semidefinite.

- **Second-order sufficient conditions**: If \( f(x) \) is twice continuously differentiable in some neighborhood of \( x^* \), \( \nabla f(x^*) = 0 \), and \( \nabla^2 f(x^*) \) is positive definite, then \( x^* \) is a strict local solution.

**Constrained Optimization**

For constrained optimization problems, the *Lagrangian function* is defined as follows:

\[
L(x, \lambda) = f(x) - \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i c_i(x)
\]

where \( \lambda_i, i \in \mathcal{E} \cup \mathcal{I}, \) are called *Lagrange multipliers*. \( \nabla_x L(x, \lambda) \) is used to denote the gradient of the Lagrangian function with respect to \( x \), and \( \nabla^2_x L(x, \lambda) \) is used to denote the Hessian of the Lagrangian function with respect to \( x \). The active set at a feasible point \( x \) is defined as

\[
A(x) = \mathcal{E} \cup \{i \in \mathcal{I} : c_i(x) = 0\}
\]

You also need the following definition before you can state the first-order and second-order necessary conditions:

- **Linear independence constraint qualification and regular point**: A point \( x \) is said to satisfy the *linear independence constraint qualification* if the gradients of active constraints
  \[
  \nabla c_i(x), \quad i \in A(x)
  \]

are linearly independent. Such a point \( x \) is called a *regular point*.

You now state the theorems that are essential in the analysis and design of algorithms for constrained optimization:

- **First-order necessary conditions**: Suppose that \( x^* \) is a local minimum and also a regular point. If \( f(x) \) and \( c_i(x), i \in \mathcal{E} \cup \mathcal{I}, \) are continuously differentiable, there exist Lagrange multipliers \( \lambda^* \in \mathbb{R}^m \) such that the following conditions hold:
  \[
  \nabla_x L(x^*, \lambda^*) = \nabla f(x^*) - \sum_{i \in \mathcal{E} \cup \mathcal{I}} \lambda_i^* \nabla c_i(x^*) = 0
  \]
  \[
  c_i(x^*) = 0, \quad i \in \mathcal{E}
  \]
  \[
  c_i(x^*) \geq 0, \quad i \in \mathcal{I}
  \]
  \[
  \lambda_i^* \geq 0, \quad i \in \mathcal{I}
  \]
  \[
  \lambda_i^* c_i(x^*) = 0, \quad i \in \mathcal{I}
  \]

The preceding conditions are often known as the *Karush-Kuhn-Tucker conditions*, or *KKT conditions* for short.
Second-order necessary conditions: Suppose that $x^*$ is a local minimum and also a regular point. Let $\lambda^*$ be the Lagrange multipliers that satisfy the KKT conditions. If $f(x)$ and $c_i(x), i \in \mathcal{E} \cup \mathcal{I}$, are twice continuously differentiable, the following conditions hold:

$$z^T \nabla_x^2 L(x^*, \lambda^*) z \geq 0$$

for all $z \in \mathbb{R}^n$ that satisfy

$$\nabla c_i(x^*)^T z = 0, \quad i \in \mathcal{A}(x^*)$$

Second-order sufficient conditions: Suppose there exist a point $x^*$ and some Lagrange multipliers $\lambda^*$ such that the KKT conditions are satisfied. If

$$z^T \nabla_x^2 L(x^*, \lambda^*) z > 0$$

for all $z \in \mathbb{R}^n$ that satisfy

$$\nabla c_i(x^*)^T z = 0, \quad i \in \mathcal{A}(x^*)$$

then $x^*$ is a strict local solution.

Note that the set of all such $z$'s forms the null space of the matrix $[\nabla c_i(x^*)^T]_{i \in \mathcal{A}(x^*)}$. Thus, you can search for strict local solutions by numerically checking the Hessian of the Lagrangian function projected onto the null space. For a rigorous treatment of the optimality conditions, see Fletcher (1987) and Nocedal and Wright (1999).

Data Set Input/Output

You can use the CREATE DATA and READ DATA statements to exchange PROC OPTMODEL data with SAS data sets. The statements can move data into and out of PROC OPTMODEL parameters and variables. For example, the following statements use a CREATE DATA statement to save the results from an optimization into a data set:

```plaintext
proc optmodel;
  var x;
  min z = (x-5)**2;
  solve;
  create data optdata from xopt=x z;
```

These statements write a single observation into the data set OPTDATA. The data set contains two variables, $x_{opt}$ and $z$, and the values contain the optimized values of the PROC OPTMODEL variable $x$ and objective $z$, respectively. The statement "$x_{opt}=x" renames the variable $x$ to $x_{opt}$.

The group of values held by a data set variable in different observations of a data set is referred to as a column. The READ DATA and CREATE DATA statements specify a set of columns for a data set and define how data are to be transferred between the columns and PROC OPTMODEL parameters.

Columns in square brackets ([ ]) are handled specially. Such columns are called key columns. Key columns specify element values that provide an implicit index for subsequent array columns. The following example uses key columns with the CREATE DATA statement to write out variable values from an array:
In this case the optimization variable $s$ is initialized to a value of 1 and is indexed by values from the set parameters LOCS and DOW. The output data set contains an observation for each combination of values in these sets. The output data set contains three variables, location, day_of_week, and sale. The data set variables location and day_of_week save the index element values for the optimization variable $s$ that is written in each observation. The data set created is shown in Figure 10.42.

**Figure 10.42** Data Sets Created

<table>
<thead>
<tr>
<th>Data Set: SOLDATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>21</td>
</tr>
</tbody>
</table>

Note that the key columns in the preceding example do not name existing PROC OPTMODEL variables. They create new local dummy parameters, location and day_of_week, in the same manner as dummy parameters in index sets. These local parameters can be used in subsequent columns. For example, the following statements demonstrate how to use a key column value in an expression for a later column value:

```plaintext
proc optmodel;
    create data tab
        from [i]=(1..10)
            Square=(i*i) Cube=(i*i*i);
```
These statements create a data set that has 10 observations that hold squares and cubes of the numbers from 1 to 10. The key column variable here is named i and is explicitly assigned the values from 1 to 10, while the data set variables Square and Cube hold the square and cube, respectively, of the corresponding value of i.

In the preceding example the key column values are simply the numbers from 1 to 10. The value is the same as the observation number, so the variable i is redundant. You can remove the data set variable for a key column via the DROP data set option, as follows:

```plaintext
proc optmodel;
   create data tab2 (drop=i)
      from [i] = (1..10)
      Square=(i*i) Cube=(i*i*i);
```

The local parameters declared by key columns receive their values in various ways. For a READ DATA statement, the key column values come from the data set variables for the column. In a CREATE DATA statement, the values can be defined explicitly, as shown in the previous example. Otherwise, the CREATE DATA statement generates a set of values that combines the index sets of array columns that need implicit indexing. The statements that produce the output in Figure 10.42 demonstrate implicit indexing.

Use a suffix (“Suffixes” on page 380) to read or write auxiliary values, such as variable bounds or constraint duals. For example, consider the following statements:

```plaintext
data pdat;
   input p $ maxprod cost;
datalines;
   ABQ 12 0.7
   MIA 9 0.6
   CHI 14 0.5
;
proc optmodel;
   set<string> plants;
   var prod{plants} >= 0;
   number cost{plants};
   read data pdat into plants=[p] prod.ub=maxprod cost;
```

The statement “plants=[p]” in the READ DATA statement declares p as a key column and instructs PROC OPTMODEL to store the set of plant names from the data set variable p into the set parameter plants. The statement assigns the upper bound for the variable prod indexed by p to be the value of the data set variable maxprod. The cost parameter location indexed by p is also assigned to be the value of the data set variable cost.

The target variables prod and cost in the preceding example use implicit indexing. Indexing can also be performed explicitly. The following version of the READ DATA statement makes the indices explicit:

```plaintext
read data pdat into plants=[p] prod[p].ub=maxprod cost[p];
```

Explicit indexing is useful when array indices need to be transformed from the key column values in the data set. For example, the following statements reverse the order in which elements from the data set are stored in an array:

```plaintext
data abcd;
   input letter $ @@;
datalines;
a b c d
;
```
proc optmodel;
    set<num> subscripts=1..4;
    string letter{subscripts};
    read data abcd into [_N_] letter[5-_N_];
    print letter;

The output from this example appears in Figure 10.43.

**Figure 10.43** READ DATA Statement: Explicit Indexing

<table>
<thead>
<tr>
<th>letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>d</td>
</tr>
<tr>
<td>c</td>
</tr>
<tr>
<td>b</td>
</tr>
<tr>
<td>a</td>
</tr>
</tbody>
</table>

The following example demonstrates the use of explicit indexing to save sequential subsets of an array in individual data sets:

data revdata;
    input month rev @@;
    datalines;
    1 200 2 345 3 362 4 958
    5 659 6 804 7 487 8 146
    9 683 10 732 11 652 12 469
;    
    proc optmodel;
    set m = 1..3;
    var revenue{1..12};
    read data revdata into [_N_] revenue=rev;
    create data qtr1 from [month]=m revenue[month];
    create data qtr2 from [month]=m revenue[month+3];
    create data qtr3 from [month]=m revenue[month+6];
    create data qtr4 from [month]=m revenue[month+9];

Each CREATE DATA statement generates a data set that represents one quarter of the year. Each data set contains the variables month and revenue. The data set qtr2 is shown in Figure 10.44.

**Figure 10.44** CREATE DATA Statement: Explicit Indexing

<table>
<thead>
<tr>
<th></th>
<th>month</th>
<th>revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs 1</td>
<td>1</td>
<td>958</td>
</tr>
<tr>
<td>Obs 2</td>
<td>2</td>
<td>659</td>
</tr>
<tr>
<td>Obs 3</td>
<td>3</td>
<td>804</td>
</tr>
</tbody>
</table>

Data set names and options that are used in PROC OPTMODEL statements can also be generated dynamically, by using an expression in parentheses to specify the name and options. The expression must evaluate to a string value with the following form:

```
[ libref . ] member [ ( options ) ]
```
Dynamic data set names are useful for processing data sets inside looping statements such as a **FOR** statement, particularly when the loop contains a **SOLVE** statement. You can replace the multiple CREATE DATA statements in the previous example with a single statement in a loop:

```optmodel
proc optmodel;
  set m = 1..3;
  var revenue(1..12);
  read data revdata into [_N_] revenue=rev;
  for {q in 1..4}
    create data ("qtr" || q)
      from [month]=m revenue[month+(q-1)*3];
```

### Control Flow

Most of the control flow statements in PROC OPTMODEL are familiar to users of the DATA step or the IML procedure. PROC OPTMODEL supports the **IF** statement, **DO** blocks, the iterative **DO** statement, the **DO WHILE** statement, and the **DO UNTIL** statement. You can also use the **CONTINUE**, **LEAVE**, and **STOP** statements to modify control flow.

PROC OPTMODEL adds the **FOR** statement. This statement is similar in operation to an iterative **DO** loop. However, the iteration is performed over the members of an index set. This form is convenient for iteration over all the locations in an array, since the valid array indices are also defined using an index set. For example, the following statements initialize the array parameter \( A \), indexed by \( i \) and \( j \), to random values sampled from a normal distribution with mean 0 and variance 1:

```optmodel
proc optmodel;
  set R=1..10;
  set C=1..5;
  number A{R, C};
  for {i in R, j in C}
    A[i, j]=rand('NORMAL');
```

The **FOR** statement provides a convenient way to perform a statement such as the preceding assignment statement for each member of a set.

### Formatted Output

PROC OPTMODEL provides two primary means of producing formatted output. The **PUT** statement provides output of data values with detailed format control. The **PRINT** statement handles arrays and produces formatted output in tabular form.

The **PUT** statement is similar in syntax to the **PUT** statement in the DATA step and in PROC IML. The **PUT** statement can output data to the SAS log, the SAS listing, or an external file. Arguments to the **PUT** statement specify the data to output and provide instructions for formatting. The **PUT** statement provides enough control to create reports within PROC OPTMODEL. However, typically the **PUT** statement is used to produce output for debugging or to quickly check data values.
The following example demonstrates some features of the PUT statement:

```plaintext
proc optmodel;
  number a=1.7, b=2.8;
  set s={a,b};
  put a b;    /* list output */
  put a= b=;  /* named output */
  put 'Value A: ' a 8.1 @30 'Value B: ' b 8.; /* formatted */
  string str='Ratio (A/B) is:';
  put str (a/b); /* strings and expressions */
  put s=;       /* named set output */

These statements produce the output in Figure 10.45.

**Figure 10.45** PUT Statement Output

```
1.7 2.8
a=1.7 b=2.8

Value A: 1.7 Value B: 3
Ratio (A/B) is: 0.6071428571
s={1.7,2.8}
```

The first PUT statement demonstrates list output. The numeric data values are output in a default format, BEST12., with leading and trailing blanks removed. A blank space is inserted after each data value is output. The second PUT statement uses the equal sign (=) to request that the variable name be output along with the regular list output.

The third PUT statement demonstrates formatted output. It uses the @ operator to position the output in a specific column. This style of output can be used in report generation. The format specification “8.” causes the displayed value of parameter b to be rounded.

The fourth PUT statement shows the output of a string value, str. It also outputs the value of an expression enclosed in parentheses. The final PUT statement outputs a set along with its name.

The default destination for PUT statement output is the SAS log. The FILE and CLOSEFILE statements can be used to send output to the SAS listing or to an external data file. Multiple files can be open at the same time. The FILE statement selects the current destination for PUT statement output, and the CLOSEFILE statement closes the corresponding file. See the section “FILE Statement” on page 316 for more details.

The PRINT statement is designed to output numeric and string data in the form of tables. The PRINT statement handles the details of formatting automatically. However, the output format can be overridden by PROC OPTMODEL options and through Output Delivery System (ODS) facilities.

The PRINT statement can output array data in a table form that contains a row for each combination of array index values. This form uses columns to display the array index values for each row and uses other columns to display the value of each requested data item. The following statements demonstrate the table form:

```plaintext
proc optmodel;
  number square{i in 0..5} = i*i;
  number recip{i in 1..5} = 1/i;
  print square recip;
```

The PRINT statement produces the output in Figure 10.46.
The first table column, labeled “[1],” contains the index values for the parameters square and recip. The columns that are labeled square and recip contain the parameter values for each array index. For example, the last row corresponds to the index 5 and the value in the last column is 0.2, which is the value of recip[5].

Note that the first row of the table contains no value in the recip column. Parameter location recip[0] does not have a valid index, so no value is printed. The PRINT statement does not display variables that are undefined or have invalid indices. This permits arrays that have similar indexing to be printed together. The sets of defined indices in the arrays are combined to generate the set of indices shown in the table.

Also note that the PRINT statement has assigned formats and widths that differ between the square and recip columns. The PRINT statement assigns a default fixed-point format to produce the best overall output for each data column. The format that is selected depends on the PDIGITS= and PWIDTH= options.

The PDIGITS= and PWIDTH= options specify the desired significant digits and formatted width, respectively. If the range of magnitudes is large enough that no suitable format can be found, then the data item is displayed in scientific format. The table in the preceding example displays the last column with five decimal places in order to display the five significant digits that were requested by the default PDIGITS= value. The square column, on the other hand, does not need any decimal places.

The PRINT statement can also display two-dimensional arrays in matrix form. If the list following the PRINT statement contains only a single array that has two index elements, then the array is displayed in matrix form when it is sufficiently dense (otherwise the display is in table form). In this form the left-most column contains the values of the first index element. The remaining columns correspond to and are labeled by the values of the second index element. The following statements print an example of matrix form:

```
proc optmodel;
  set R=1..6;
  set C=1..4;
  number a{i in R, j in C} = 10*i+j;
  print a;
```

The PRINT statement produces the output in Figure 10.47.
Figure 10.47  PRINT Statement Output (Matrix Form)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>21</td>
<td>22</td>
<td>23</td>
<td>24</td>
</tr>
<tr>
<td>3</td>
<td>31</td>
<td>32</td>
<td>33</td>
<td>34</td>
</tr>
<tr>
<td>4</td>
<td>41</td>
<td>42</td>
<td>43</td>
<td>44</td>
</tr>
<tr>
<td>5</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
</tr>
<tr>
<td>6</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
</tr>
</tbody>
</table>

In the example the first index element ranges from 1 to 6 and corresponds to the table rows. The second index element ranges from 1 to 4 and corresponds to the table columns. Array values can be found based on the row and column values. For example, the value of parameter \( a[3,2] \) is 32. This location is found in the table in the row labeled “3” and the column labeled “2.”

ODS Table and Variable Names

PROC OPTMODEL assigns a name to each table it creates. You can use these names to reference the table when you use the Output Delivery System (ODS) to select tables and create output data sets. The names of tables common to all solvers are listed in Table 10.12. Some solvers can generate additional tables; see the individual solver chapters for more information. For more information about ODS, see SAS Output Delivery System: User’s Guide.

Table 10.12  ODS Tables Produced in PROC OPTMODEL

<table>
<thead>
<tr>
<th>ODS Table Name</th>
<th>Description</th>
<th>Statement/Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>DerivMethods</td>
<td>List of derivatives used by the solver, including the method of computation</td>
<td>SOLVE</td>
</tr>
<tr>
<td>OptStatistics</td>
<td>Solver-dependent description of the resources required for solution, including function evaluations and solver time</td>
<td>SOLVE</td>
</tr>
<tr>
<td>PrintTable</td>
<td>Specified parameter or variable values</td>
<td>PRINT</td>
</tr>
<tr>
<td>ProblemSummary</td>
<td>Description of objective, variables, and constraints</td>
<td>SOLVE</td>
</tr>
<tr>
<td>ProfileInfo</td>
<td>Detailed timing of statements and declarations</td>
<td>PROFILE</td>
</tr>
<tr>
<td>SolutionSummary</td>
<td>Overview of solution, including solver-dependent solution quality values</td>
<td>SOLVE</td>
</tr>
<tr>
<td>SolverOptions</td>
<td>List of solver options and their values</td>
<td>SOLVE</td>
</tr>
<tr>
<td>Timing</td>
<td>Detailed solution timing</td>
<td>SOLVE</td>
</tr>
</tbody>
</table>
To guarantee that ODS output data sets contain information from all executed statements, use the PERSIST= option in the ODS OUTPUT statement. For details, see SAS Output Delivery System: User’s Guide. **NOTE:** The SUBMIT statement resets ODS SELECT and EXCLUDE lists.

Table 10.13 lists the variable names of the preceding tables used in the ODS template of the OPTMODEL procedure.

**Table 10.13** Variable Names for the ODS Tables Produced in PROC OPTMODEL

<table>
<thead>
<tr>
<th>ODS Table Name</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>DerivMethods</td>
<td>Label1, cValue1, and nValue1</td>
</tr>
<tr>
<td>OptStatistics</td>
<td>Label1, cValue1, and nValue1</td>
</tr>
<tr>
<td>PrintTable (matrix form)</td>
<td>ROW, COL1 – COLn</td>
</tr>
<tr>
<td>PrintTable (table form)</td>
<td>COL1 – COLn, identifier-expression(_suffix)</td>
</tr>
<tr>
<td>ProblemSummary</td>
<td>Label1, cValue1, and nValue1</td>
</tr>
<tr>
<td>ProfileInfo</td>
<td>Item, Line, Column, Count, MemSize (optional),</td>
</tr>
<tr>
<td></td>
<td>NetTime, WaitTime, and PctTime</td>
</tr>
<tr>
<td>SolutionSummary</td>
<td>Label1, cValue1, and nValue1</td>
</tr>
<tr>
<td>SolverOptions</td>
<td>Label1, cValue1, nValue1, cValue2, and nValue2</td>
</tr>
<tr>
<td>Timing</td>
<td>Label1, cValue1, nValue1, cValue2, and nValue2</td>
</tr>
</tbody>
</table>

The PRINT statement produces an ODS table named PrintTable. The variable names that are used depend on the display format used. See the section “Formatted Output” on page 368 for details about choosing the display format.

For the PRINT statement with table format, the columns that display array indices are named COL1–COLn, where n is the number of index elements. Columns that display values from identifier expressions are named using the expression’s name and suffix. The identifier name becomes the output variable name if no suffix is used. Otherwise the variable name is formed by appending an underscore (_) and the suffix to the identifier name. Columns that display the value of expressions are named COLn, where n is the column number in the table.

For the PRINT statement with matrix format, the first column has the variable name ROW. The remaining columns are named COL1–COLn, where n is the number of distinct column indices. When an ODS table displays values from identifier expressions, a label is generated based on the expression’s name and suffix, as described for column names in the case of table format.

The PRINTLEVEL= option controls the ODS tables produced by the SOLVE and COFOR statements. When PRINTLEVEL=0, the statements produce no ODS tables. When PRINTLEVEL=1, the SOLVE statement produces the ODS tables ProblemSummary and SolutionSummary. When PRINTLEVEL=2, the SOLVE statement produces the ODS tables ProblemSummary, SolverOptions, DerivMethods, OptStatistics, and Timing.
The following statements generate several ODS tables and write each table to a SAS data set:

```sas
proc optmodel printlevel=2;
    ods output PrintTable=expt ProblemSummary=exps DerivMethods=exdm
        SolverOptions=exso SolutionSummary=exss OptStatistics=exos
        Timing=exti;
    var x{1..2} >= 0;
    min z = 2*x[1] + 3 * x[2] + x[1]**2 + 10*x[2]**2
        + 2.5*x[1]*x[2] + x[1]**3;
    con c2: x[1] + 2*x[2] >= 100;
    solve;
    print x;
```

The data set `expt` contains the PrintTable table and is shown in Figure 10.48. The variable names are COL1 and x.

![Figure 10.48 PrintTable ODS Table](image)

<table>
<thead>
<tr>
<th>Obs</th>
<th>COL1</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>10.448</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>44.776</td>
</tr>
</tbody>
</table>

The data set `exps` contains the ProblemSummary table and is shown in Figure 10.49. The variable names are Label1, cValue1, and nValue1. The rows describe the instance, and the description depends on the form of the problem. In most solvers, the rows describe the objective function, variables, and constraints. In the network solver, the rows describe the number of nodes, the number of edges, the directedness of the graph, and the type of problem solved over the graph.

![Figure 10.49 ProblemSummary ODS Table](image)

<table>
<thead>
<tr>
<th>Obs</th>
<th>Label1</th>
<th>cValue1</th>
<th>nValue1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Objective Sense</td>
<td>Minimization</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Objective Function</td>
<td>z</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Objective Type</td>
<td>Nonlinear</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Number of Variables</td>
<td>2</td>
<td>2.000000</td>
</tr>
<tr>
<td>6</td>
<td>Bounded Above</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>Bounded Below</td>
<td>2</td>
<td>2.000000</td>
</tr>
<tr>
<td>8</td>
<td>Bounded Below and Above</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>Free</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>Fixed</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Number of Constraints</td>
<td>2</td>
<td>2.000000</td>
</tr>
<tr>
<td>13</td>
<td>Linear LE (&lt;=)</td>
<td>1</td>
<td>1.000000</td>
</tr>
<tr>
<td>14</td>
<td>Linear EQ (=)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>Linear GE (&gt;=)</td>
<td>1</td>
<td>1.000000</td>
</tr>
<tr>
<td>16</td>
<td>Linear Range</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
The data set `exso` contains the SolverOptions table and is shown in Figure 10.50. The variable names are Label1, cValue1, nValue1, cValue2, and nValue2. The rows, which depend on the solver called by PROC OPTMODEL, list the values taken by each of the solver options. The presence of an asterisk (*) next to an option indicates that a value has been specified for that option.

**Figure 10.50** SolverOptions ODS Table

```
SolverOptions

<table>
<thead>
<tr>
<th>Obs</th>
<th>Label1</th>
<th>cValue1</th>
<th>nValue1</th>
<th>cValue2</th>
<th>nValue2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ALGORITHM INTERIORPOINT</td>
<td>.</td>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>FEASTOL 1E-6</td>
<td>0.000001000</td>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>HESSTYPE FULL</td>
<td>.</td>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>LOGFREQ 1</td>
<td>1.00000000</td>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>MAXITER 5000</td>
<td>5000.000000</td>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>MAXTIME 1</td>
<td>1</td>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>NTHREADS 4</td>
<td>4.00000000</td>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>OBJLIMIT 1E20</td>
<td>1E20</td>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>OPTTOL 1E-6</td>
<td>0.000001000</td>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>SOLTYP 1</td>
<td>1</td>
<td>.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>TIMETYPE REAL</td>
<td>.</td>
<td>.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The data set `exdm` contains the DerivMethods table, which displays the methods of derivative computation, and is shown in Figure 10.51. The variable names are Label1, cValue1, and nValue1. The rows, which depend on the derivatives used by the solver, specify the method used to calculate each derivative.

**Figure 10.51** DerivMethods ODS Table

```
DerivMethods

<table>
<thead>
<tr>
<th>Obs</th>
<th>Label1</th>
<th>cValue1</th>
<th>nValue1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Objective Gradient Analytic Formulas</td>
<td>.</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Objective Hessian Analytic Formulas</td>
<td>.</td>
<td></td>
</tr>
</tbody>
</table>
```

The data set `exss` contains the SolutionSummary table and is shown in Figure 10.52. The variable names are Label1, cValue1, and nValue1. The rows give an overview of the solution, including the solver chosen, the objective value, and the solution status. Depending on the values returned by the solver, the SolutionSummary table might also include some solution quality values such as optimality error and infeasibility. The values in the SolutionSummary table appear in the _OROPTMODEL_ macro variable; each solver chapter has a section that describes the solver’s contribution to this macro variable.
The data set `exos` contains the OptStatistics table, which displays the optimization statistics, and is shown in Figure 10.53. The variable names are `Label1`, `cValue1`, and `nValue1`. The rows, which depend on the solver called by PROC OPTMODEL, describe the amount of time and the function evaluations that are used by the solver and associated processing. Times are displayed in seconds of clock or CPU time according to the value of the TIMETYPE= option that is used by the solver.

Problem generation is the process of combining the model with the data into a format that solvers can use. This includes computing equation coefficients, but it does not include reading data or evaluating other programming statements. Code generation is compiles code for nonlinear equations in the model and performs other analysis that is needed prior to solver evaluations. The time required for problem generation will be negligible if the model contains only linear equations. The presolve time in this table includes the time used by the PROC OPTMODEL presolver and any presolver that is part of the solver. Solution time is the sum of the times used by the presolvers and the solver. The presolve and solution times also appear in the SolutionSummary table. The OptStatistics table includes a total time, which is the sum of times for problem generation, code generation, solution, and overhead in the SOLVE statement. Overhead includes solver setup, postprocessing, and ODS table output.
The Timing table provides an alternate breakdown of SOLVE statement timing. Times in this table are shown in seconds of clock time. The data set exti, which is shown in Figure 10.54, contains the Timing table data and statistics. The variable names are Label1, cValue1, nValue1, cValue2, and nValue2. The values present depend on the solver and on the context of the SOLVE statement.

**Figure 10.54** Timing ODS Table

<table>
<thead>
<tr>
<th>Obs</th>
<th>Label1</th>
<th>cValue1</th>
<th>nValue1</th>
<th>cValue2</th>
<th>nValue2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Problem Generation</td>
<td>0.0016999112</td>
<td>0.00</td>
<td>0.00358548</td>
<td>0.36%</td>
</tr>
<tr>
<td>2</td>
<td>OPTMODEL Presolver</td>
<td>0.0002297793</td>
<td>0.00</td>
<td>0.0048488203</td>
<td>0.48%</td>
</tr>
<tr>
<td>3</td>
<td>Solver Initialization</td>
<td>0.0378440204</td>
<td>0.04</td>
<td>0.7985874653</td>
<td>79.86%</td>
</tr>
<tr>
<td>4</td>
<td>Code Generation</td>
<td>0.0040790099</td>
<td>0.00</td>
<td>0.0860755838</td>
<td>8.61%</td>
</tr>
<tr>
<td>5</td>
<td>Solver</td>
<td>0.0047731942</td>
<td>0.00</td>
<td>0.1007243151</td>
<td>10.07%</td>
</tr>
<tr>
<td>6</td>
<td>Solver Postprocessing</td>
<td>0.0002927833</td>
<td>0.00</td>
<td>0.0061783355</td>
<td>0.62%</td>
</tr>
</tbody>
</table>

Some of the Timing table values have already been described for the OptStatistics table. Solver initialization time is overhead in the SOLVE statement before the solver starts. Solver time includes execution of the solver and its associated preprocessor, if any. Solver postprocessing time is overhead in the SOLVE statement after the solver has completed. Several table values appear only for a SOLVE statement called within a COFOR loop. These values are shown in Table 10.14.

**Table 10.14** Solver Timing Values for COFOR Loops

<table>
<thead>
<tr>
<th>Label</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Waiting for Multiple Threads</td>
<td>Time waiting for the required number of threads to become available on the client machine when NTHREADS &gt; 1</td>
</tr>
<tr>
<td>Waiting for Grid</td>
<td>Time waiting for the distributed computing environment to become ready to accept a new problem</td>
</tr>
<tr>
<td>Sending to Grid</td>
<td>Time to transmit a solver problem description to the distributed computing environment</td>
</tr>
<tr>
<td>Receiving from Grid</td>
<td>Time to receive solver results from the distributed computing environment</td>
</tr>
<tr>
<td>Waiting after Solver</td>
<td>Time between solver completion and the resumption of the SOLVE statement for processing the results</td>
</tr>
<tr>
<td>Grid Overhead</td>
<td>Time required for processing in the distributed computing environment that is not included in the preceding times</td>
</tr>
</tbody>
</table>

**Constraints**

You can add constraints to a PROC OPTMODEL model. The solver tries to satisfy the specified constraints while minimizing or maximizing the objective.

Constraints in PROC OPTMODEL have names. By using the name, you can examine various attributes of the constraint, such as the dual value that is returned by the solver (see the section “Suffixes” on page 380 for details). A constraint is not allowed to have the same name as any other model component.
PROC OPTMODEL provides a default name if none is supplied by the constraint declaration. The predefined array _ACON_ provides names for otherwise anonymous constraints. The predefined numeric parameter _NACON_ contains the number of such constraints. The constraints are assigned integer indices in sequence, so _ACON_[1] refers to the first unnamed constraint declared, while _ACON_[_NACON_] refers to the newest.

Consider the following example of a simple model that has a constraint:

```plaintext
proc optmodel;
  var x, y;
  min r = x**2 + y**2;
  con c: x+y >= 1;
  solve;
  print x y;
```

Without the constraint named c, the solver would find the point \( x = y = 0 \) that has an objective value of 0. However, the constraint makes this point infeasible. The resulting output is shown in Figure 10.55.

**Figure 10.55** Constrained Model Solution

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objective Sense</strong></td>
</tr>
<tr>
<td><strong>Objective Function</strong></td>
</tr>
<tr>
<td><strong>Objective Type</strong></td>
</tr>
<tr>
<td><strong>Number of Variables</strong></td>
</tr>
<tr>
<td><strong>Bounded Above</strong></td>
</tr>
<tr>
<td><strong>Bounded Below</strong></td>
</tr>
<tr>
<td><strong>Bounded Below and Above</strong></td>
</tr>
<tr>
<td><strong>Free</strong></td>
</tr>
<tr>
<td><strong>Fixed</strong></td>
</tr>
<tr>
<td><strong>Number of Constraints</strong></td>
</tr>
<tr>
<td><strong>Linear LE (&lt;=)</strong></td>
</tr>
<tr>
<td><strong>Linear EQ (=)</strong></td>
</tr>
<tr>
<td><strong>Linear GE (&gt;=)</strong></td>
</tr>
<tr>
<td><strong>Linear Range</strong></td>
</tr>
<tr>
<td><strong>Constraint Coefficients</strong></td>
</tr>
<tr>
<td><strong>Hessian Diagonal Elements</strong></td>
</tr>
<tr>
<td><strong>Hessian Elements Below Diagonal</strong></td>
</tr>
</tbody>
</table>
The solver has found the point where the objective function is minimized in the region $x + y \geq 1$. This is actually on the border of the region: the constraint $c$ is active (see the section “Dual Values” on page 385 for details).

In the preceding example the constraint $c$ had only a lower bound. You can specify constraints that have both upper and lower bounds. For example, replacing the constraint $c$ in the previous example would further restrict the feasible region:

```plaintext
con c: 3 >= x+y >= 1;
```

PROC OPTMODEL standardizes constraints to collect the expression terms that depend on variables and to separate the expression terms that are constant. When there is a single equality or inequality operator, the separable constant terms are moved to the right operand while the variable terms are moved to the left operand. For range constraints, the separable constant terms from the middle expression are subtracted from the lower and upper bounds. You can see the standardized constraints with the use of the EXPAND statement in the following example. Consider the following PROC OPTMODEL statements:

```plaintext
proc optmodel;
  var x{1..3};
  con b: sum{i in 1..3}(x[i] - i) = 0;
  expand b;
```

These statements produce an optimization problem with the following constraint:


The EXPAND statement produces the output in Figure 10.56.

![Figure 10.56 Expansion of a Standardized Constraint](image)
Here the i separable constant terms in the operand of the SUM operation were moved to the right-hand side of the constraint. The sum of these i values is 6.

After standardization the constraint expression that contains all the variables is called the body of the constraint. You can reference the current value of the body expression by attaching the .body suffix to the constraint name. Similarly, the upper and lower bound expressions can be referenced by using the .ub and .lb suffixes, respectively. (See the section “Suffixes” on page 380 for more information.)

As a result of standardization, the value of a body expression depends on how the corresponding constraint is entered. The following example demonstrates how using equivalent relational syntax can result in different .body values:

```latex
proc optmodel;
  var x init 1;
  con c1: x**2 <= 5;
  con c2: 5 >= x**2;
  con c3: -x**2 >= -5;
  con c4: -5 <= -x**2;
  expand;
  print c1.body c2.body c3.body c4.body;
```

The EXPAND and PRINT statements produce the output in Figure 10.57.

**Figure 10.57** Expansion and Body Values of Standardized Constraints

<table>
<thead>
<tr>
<th>Var</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint c1:</td>
<td>x**2 &lt;= 5</td>
</tr>
<tr>
<td>Constraint c2:</td>
<td>-x**2 &gt;= -5</td>
</tr>
<tr>
<td>Constraint c3:</td>
<td>-x**2 &gt;= -5</td>
</tr>
<tr>
<td>Constraint c4:</td>
<td>-x**2 &lt;= 5</td>
</tr>
</tbody>
</table>

**CAUTION:** Each constraint has an associated dual value (see “Dual Values” on page 385). As a result of standardization, the sign of a dual value depends in some instances on the way in which the corresponding constraint is entered into PROC OPTMODEL. In the case of a minimization objective with one-sided constraint \( g(x) \geq L \), avoid entering the constraint as \( L \leq g(x) \). For example, the following statements produce a value of 2:

```latex
proc optmodel;
  var x;
  min o1 = x**2;
  con c1: x >= 1;
  solve;
  print (c1.dual);
```

Replacing the constraint as follows results in a value of \(-2\):

```latex
con c1: 1 <= x;
```

In the case of a maximization objective with the one-sided constraint \( g(x) \leq U \), avoid entering the constraint as \( U \geq g(x) \).
When a constraint has variables on both sides, the sign of the dual value depends on the direction of the inequality. For example, you can enter the following constraint:

```plaintext
con c1: x**5 - y + 8 <= 5*x + y**2;
```

This is a ≤ constraint, so `c1.dual` is nonpositive. If you enter the same constraint as follows, then `c1.dual` is nonnegative:

```plaintext
con c1: 5*x + y**2 >= x**5 - y + 8;
```

It is also important to note that the signs of the dual values are negated in the case of maximization. The following statements output a value of 2:

```plaintext
proc optmodel;
var x;
min o1 = x**2;
con c1: 1 <= x <= 2;
solve;
print (c1.dual);
```

Changing the objective function as follows yields the same value of `x`, but `c1.dual` now holds the value –2:

```plaintext
max o2 = -x**2;
solve;
print (c1.dual);
```

**NOTE:** A simple bound constraint on a decision variable `x` can be entered either by using a CONSTRAINT declaration or by assigning values to `x.lb` and `x.ub`. If you require dual values for simple bound constraints, use the CONSTRAINT declaration.

Constraints that are specified using relational operators can be linear or nonlinear. PROC OPTMODEL determines the type of constraint automatically by examining the form of the body expression. Subexpressions that do not involve variables are treated as constants. Constant subexpressions that are multiplied by or added to linear subexpressions produce new linear subexpressions. For example, constraint `A` in the following statements is linear:

```plaintext
proc optmodel;
var x{1..3};
con A: 0.5*(x[1]-x[2]) + x[3] >= 0;
```

---

**Suffixes**

You can use suffixes with `identifier-expressions` to retrieve and modify various auxiliary values maintained by the solver. The values of the suffixes can come from expressions in the declaration of the name that is suffixed. For example, the following declaration of variable `v` provides the values of several suffixes of `v` at the same time:

```plaintext
var v >= 0 <= 2 init 1 suffixes=(label='X0000000' status init 'L');
```

The preceding example sets the lower and upper bounds for variable `v` (that is, `v.lb` and `v.ub`, respectively) to 0 and 2. The solver label (`v.label`) is set to the string ‘X0000000’, and the initial solver status (`v.status`) is ‘L’. The initial value of `v` is also set to 1. You can use more complex expressions in a declaration, as shown in the following example:
num ubound[1..N] init 2;
var x{i in 1..N} >= 0 <= ubound[i] suffixes(label='X'||put(i,z7.));

The preceding declaration makes the upper bound value for each element of x equal to the value of the element in the ubound array with a corresponding index. That is, for each i in 1 to N, x[i].ub is equal to ubound[i]. Similarly, the declaration provides a distinct .label value for each element of the x array.

The CONSTRAINT, MAX, MIN, PROBLEM, and VAR statements support a SUFFIXES=() clause that allows values for existing, modifiable suffixes (other than .lb and .ub) to be specified with each declaration. This is the syntax for a SUFFIXES=() clause:

```
SUFFIXES=( values )
```

Each value specifies a suffix value. The following forms are allowed:

- \texttt{name = expression}  
  provides an explicit value for the suffix \texttt{name}. In this case the suffix acts like an alias for the \texttt{expression} value. When you use this form, the suffix value cannot be modified directly, such as by an assignment statement. Suffixes that are updated by the SOLVE statement, such as .status, cannot be specified using this form.

- \texttt{name INIT expression}  
  specifies a default value when the suffix \texttt{name} is first used but no other value has been supplied.

The values of the suffixes also come from the solver or from values assigned by assignment, READ DATA, or other statements (see an example in the section “Data Set Input/Output” on page 364). Note that PROC OPTMODEL allows the .lb and .ub suffixes to be assigned after the declaration even when the declaration provides a value with an expression following \texttt{>=} or \texttt{<=}.

You must use suffixes with names of the appropriate type. For example, the .dual suffix cannot be used with the name of an objective. In particular, local dummy parameter names cannot have suffixes.

Table 10.15 shows the names of the available suffixes.

<table>
<thead>
<tr>
<th>Name Type</th>
<th>Suffix</th>
<th>Modifiable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>any</td>
<td>.name</td>
<td>No</td>
<td>Name text for any non-dummy symbol</td>
</tr>
<tr>
<td>Constraint</td>
<td>.active</td>
<td>No</td>
<td>Active status in the current problem</td>
</tr>
<tr>
<td>Constraint</td>
<td>.block</td>
<td>Yes</td>
<td>Block ID for decomposition</td>
</tr>
<tr>
<td>Constraint</td>
<td>.body</td>
<td>No</td>
<td>Current constraint body value</td>
</tr>
<tr>
<td>Constraint</td>
<td>.dual</td>
<td>No</td>
<td>Dual value from the solver</td>
</tr>
<tr>
<td>Constraint</td>
<td>.label</td>
<td>Yes</td>
<td>Label text for the solver</td>
</tr>
<tr>
<td>Constraint</td>
<td>.lb</td>
<td>Yes</td>
<td>Current lower bound</td>
</tr>
<tr>
<td>Constraint</td>
<td>.status</td>
<td>Yes</td>
<td>Status information from the solver</td>
</tr>
<tr>
<td>Constraint</td>
<td>.ub</td>
<td>Yes</td>
<td>Current upper bound</td>
</tr>
<tr>
<td>Implicit variable</td>
<td>.sol</td>
<td>No</td>
<td>Current solution value</td>
</tr>
<tr>
<td>Objective</td>
<td>.active</td>
<td>No</td>
<td>Active status in the current problem</td>
</tr>
<tr>
<td>Objective</td>
<td>.sol</td>
<td>No</td>
<td>Current objective value</td>
</tr>
<tr>
<td>Objective</td>
<td>.label</td>
<td>Yes</td>
<td>Label text for the solver</td>
</tr>
<tr>
<td>Problem</td>
<td>.active</td>
<td>No</td>
<td>Active status of the problem</td>
</tr>
</tbody>
</table>
Table 10.15 (continued)

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Suffix</th>
<th>Modifiable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem</td>
<td></td>
<td>.label</td>
<td>Yes</td>
<td>Label text for the solver</td>
</tr>
<tr>
<td>Variable</td>
<td></td>
<td>.active</td>
<td>No</td>
<td>Active status in the current problem</td>
</tr>
<tr>
<td>Variable</td>
<td></td>
<td>.direction</td>
<td>Yes</td>
<td>Branching direction for MILP</td>
</tr>
<tr>
<td>Variable</td>
<td></td>
<td>.dual</td>
<td>No</td>
<td>Alias for .rc</td>
</tr>
<tr>
<td>Variable</td>
<td></td>
<td>.fixed</td>
<td>No</td>
<td>Fixed status</td>
</tr>
<tr>
<td>Variable</td>
<td></td>
<td>.label</td>
<td>Yes</td>
<td>Label text for the solver</td>
</tr>
<tr>
<td>Variable</td>
<td></td>
<td>.lb</td>
<td>Yes</td>
<td>Lower bound</td>
</tr>
<tr>
<td>Variable</td>
<td></td>
<td>.msinit</td>
<td>No</td>
<td>Numeric value at the best starting point reported by the multistart solver</td>
</tr>
<tr>
<td>Variable</td>
<td></td>
<td>.priority</td>
<td>Yes</td>
<td>Branching priority for MILP and CLP</td>
</tr>
<tr>
<td>Variable</td>
<td></td>
<td>.rc</td>
<td>No</td>
<td>Reduced cost (LP) or gradient of the Lagrangian function</td>
</tr>
<tr>
<td>Variable</td>
<td></td>
<td>.relax</td>
<td>Yes</td>
<td>Relaxation of integrality restriction</td>
</tr>
<tr>
<td>Variable</td>
<td></td>
<td>.sol</td>
<td>No</td>
<td>Current variable value</td>
</tr>
<tr>
<td>Variable</td>
<td></td>
<td>.sol[i]</td>
<td>Yes</td>
<td>Saved solution value</td>
</tr>
<tr>
<td>Variable</td>
<td></td>
<td>.status</td>
<td>Yes</td>
<td>Status information from the solver</td>
</tr>
<tr>
<td>Variable</td>
<td></td>
<td>.ub</td>
<td>Yes</td>
<td>Upper bound</td>
</tr>
</tbody>
</table>

The .sol suffix for a variable, implicit variable, or objective can be used within a declaration to reference the current value of the symbol. It is treated as a constant in such cases. The value is independent of the current problem. When the OPTMODEL procedure processes a SOLVE statement, the value is fixed at the start of the SOLVE statement. The .sol suffix can be followed by a positive integer solution index to refer to a particular solution that the SOLVE statement returns. See the section “Multiple Solutions” on page 398 for more information about accessing multiple solutions from the solver. Each problem tracks saved solution values separately. Outside of declarations, a variable, implicit variable, or objective name with the .sol suffix and no solution index is equivalent to the unsuffixed name.

The .status suffix reports status information from the solver. Currently, only the LP solver provides status information. The .status suffix takes on the same character values that are found in the _STATUS_ variable of the PRIMALOUT and DUALOUT data sets for the OPTLP procedure, including values set by the IIS= option. See the section “Variable and Constraint Status” on page 518 and the section “Irreducible Infeasible Set” on page 519, both in Chapter 12, “The Linear Programming Solver,” for more information. For other solvers, the .status values default to a single blank character.

If you choose to modify the .status suffix for a variable or constraint, the assigned suffix value can be a single character or an empty string. The LP solver rejects invalid status characters. Blank or empty strings are treated as new row or column entries for the purpose of “warm starting” the solver.

The .active suffix reports the current activity status for names in the problem. The value is 1 if the element is active or 0 otherwise. A PROBLEM name is considered active if it is the current problem (that is, it was selected by the most recent USE PROBLEM statement). A constraint is considered active if it is included in the current problem and not dropped. An objective is considered active if it is the last objective that is added to the current problem. A variable is considered active if it is included in the current problem, independent of the fixed status.
The .fixed suffix reports the fixed status of a variable. The value is 1 if the variable is fixed using the FIX statement for the current problem or 0 otherwise. Variables that are not included in the current problem are treated as unfixed.

The .msinit suffix reports the numeric value of a variable at the best starting point, as reported by the NLP solver when the MULTISTART option is specified. If the solver does not report a best starting point, then the value is missing. The value is tracked independently for each problem to support multiple subproblems. See the section “Multistart” on page 794 in Chapter 16, “The Nonlinear Programming Solver,” for more information.

The .block suffix identifies the subproblem for constraints when used with the METHOD=USER option of the decomposition algorithm. The value must be numeric and is initially assigned a missing value. A constraint with a missing value for the .block suffix is part of the master problem. Otherwise constraints belong to the same subproblem if and only if they have the same .block suffix values. See Chapter 18, “The Decomposition Algorithm,” for more information.

The .label suffix represents the text passed to the solver to identify a variable, constraint, or objective. Some solvers can display this label in their output. The maximum text length passed to the solver is controlled by the MAXLABLEN= option. The default text is based on the name in the model, abbreviated to fit within MAXLABLEN. For example, a model variable x[1] would be labeled “x[1]”. This label text can be reassigned. The .label suffix value is also used to create MPS labels stored in the output data set for the SAVE MPS and SAVE QPS statements.

The .name suffix represents the name of a symbol as a text string. The .name suffix can be used with any declared name except for local dummy parameters. This suffix is primarily useful when applied to problem symbols (see the section “Problem Symbols” on page 399), since the .name suffix returns the name of the referenced symbol, not the problem symbol name. The name text is based on the name in the model, abbreviated to fit in 256 characters.

Suffixed names can be used wherever a parameter name is accepted, provided only the value is required. However, you are not allowed to change the value of certain suffixes. Table 10.15 marks these suffixes as not modifiable. Suffixed names that are used as a target in an assignment or READ DATA statement must be modifiable.

The following statements formulate a trivial linear programming problem. The objective value is unbounded, which is reported after the execution of the SOLVE statement. The PRINT statements illustrate the corresponding default auxiliary values. This is shown in Figure 10.58.

```plaintext
proc optmodel;
  var x, y;
  min z = x + y;
  con c: x + 2*y <= 3;
  solve;
  print x.lb x.ub x.status x.sol;
  print y.lb y.ub y.status y.sol;
  print c.lb c.ub c.body c.dual;
```

**Figure 10.58 Using a Suffix: Retrieving Auxiliary Values**

<table>
<thead>
<tr>
<th></th>
<th>LB</th>
<th>UB</th>
<th>STATUS</th>
<th>SOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>(-1.7977E+308)</td>
<td>(1.7977E+308)</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>LB</th>
<th>UB</th>
<th>STATUS</th>
<th>SOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>(-1.7977E+308)</td>
<td>(1.7977E+308)</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Next, continue to submit the following statements to change the default bounds and solve again. The output is shown in Figure 10.59.

```sas
x.lb=0;
y.lb=0;
c.lb=1;
solve;
print x.lb x.ub x.status x.sol;
print y.lb y.ub y.status y.sol;
print c.lb c.ub c.body c.dual;
```

**Figure 10.59 Using a Suffix: Modifying Auxiliary Values**

<table>
<thead>
<tr>
<th>x.LB</th>
<th>x.UB</th>
<th>x.STATUS</th>
<th>x.SOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.7977E+308</td>
<td>L</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>y.LB</th>
<th>y.UB</th>
<th>y.STATUS</th>
<th>y.SOL</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1.7977E+308</td>
<td>B</td>
<td>0.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c.LB</th>
<th>c.UB</th>
<th>c.BODY</th>
<th>c.DUAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>0.5</td>
</tr>
</tbody>
</table>

**NOTE:** Spaces are significant. The form `NAME,._TAG` is treated as a SAS format name followed by the tag name, not as a suffixed identifier. The forms `NAME,._TAG`, `NAME,_.TAG`, and `NAME,_TAG` (note the location of spaces) are interpreted as suffixed references.

### Integer Variable Suffixes

The suffixes `.relax`, `.priority`, and `.direction` are applicable to integer variables.

For an integer variable `x`, setting `x.relax` to a nonzero, nonmissing value relaxes the integrality restriction. The value of `x.relax` is read as either 1 or 0, depending on whether or not integrality is relaxed. This suffix is ignored for noninteger variables.

The value that is contained in `x.priority` sets the priority of an integer variable `x` for use with branching in the MILP solver or selection in the CLP solvers. This value can be any nonnegative, nonmissing number. The default value is 0, which indicates default branching priority. Variables with positive `.priority` values are assigned greater priority than the default. Variables with the highest `.priority` values are assigned the highest priority. Variables with the same `.priority` value are assigned the same branching priority.

The value of `x.direction` assigns a branching direction to an integer variable `x`. This value should be an integer in the range –1 to 3. A noninteger value in this range is rounded on assignment. The default value is 0. The significance of each integer is found in Table 10.16.
Table 10.16  Branching Directions

<table>
<thead>
<tr>
<th>Value</th>
<th>Direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>–1</td>
<td>Round down to nearest integer</td>
</tr>
<tr>
<td>0</td>
<td>Default</td>
</tr>
<tr>
<td>1</td>
<td>Round up to nearest integer</td>
</tr>
<tr>
<td>2</td>
<td>Round to nearest integer</td>
</tr>
<tr>
<td>3</td>
<td>Round to closest presolved bound</td>
</tr>
</tbody>
</table>

Suppose the solver branches next on an integer variable x whose last LP relaxation solution is 3.3. Suppose also that after passing through the presolver, the lower bound of x is 0 and the upper bound of x is 10. If the value in x.direction is –1 or 2, then the solver sets x to 3 for the next iteration. If the value in x.direction is 1, then the solver sets x to 4. If the value in x.direction is 3, then the solver sets x to 0.

The MPS data set created by the SAVE MPS statement ("SAVE MPS Statement" on page 331) includes a BRANCH section if any nondefault .priority or .direction values have been specified for integer variables.

Dual Values

A dual value is associated with each constraint. To access the dual value of a constraint, use the constraint name followed by the suffix .dual.

For linear programming problems, the dual value associated with a constraint is also known as the dual price (also called the shadow price). The shadow price is usually interpreted economically as the rate at which the optimal value changes with respect to a change in some right-hand side that represents a resource supply or demand requirement.

For nonlinear programming problems, the dual values correspond to the values of the optimal Lagrange multipliers. For more details about duality in nonlinear programming, see Bazaraa, Sherali, and Shetty (1993).

From the dual value associated with the constraint, you can also tell whether the constraint is active or not. A constraint is said to be active (tight at a point) if it holds with equality at that point. It can be informative to identify active constraints at the optimal point and check their corresponding dual values. Relaxing the active constraints might improve the objective value.

Background on Duality in Mathematical Programming

For a minimization problem, there exists an associated problem with the following property: any feasible solution to the associated problem provides a lower bound for the original problem, and conversely any feasible solution to the original problem provides an upper bound for the associated problem. The original and the associated problems are referred to as the primal and the dual problem, respectively. More specifically, consider the primal problem,

\[
\begin{align*}
\text{minimize } & \quad f(x) \\
\text{subject to } & \quad c_i(x) = 0, \quad i \in \mathcal{E} \\
& \quad c_i(x) \leq 0, \quad i \in \mathcal{L} \\
& \quad c_i(x) \geq 0, \quad i \in \mathcal{G}
\end{align*}
\]
where \( E, \mathcal{L}, \) and \( \mathcal{G} \) denote the sets of equality, \( \leq \) inequality, and \( \geq \) inequality constraints, respectively. Variables \( x \in \mathbb{R}^n \) are called the primal variables. The Lagrangian function of the primal problem is defined as

\[
L(x, \lambda, \mu, v) = f(x) - \sum_{i \in E} \lambda_i c_i(x) - \sum_{i \in \mathcal{L}} \mu_i c_i(x) - \sum_{i \in \mathcal{G}} v_i c_i(x)
\]

where \( \lambda_i \in \mathbb{R}, \mu_i \leq 0, \) and \( v_i \geq 0 \). By convention, the Lagrange multipliers for inequality constraints have to be nonnegative. Hence \( \lambda, -\mu, \) and \( v \) correspond to the Lagrange multipliers in the preceding Lagrangian function. It can be seen that the Lagrangian function is a linear combination of the objective function and constraints of the primal problem.

The Lagrangian function plays a fundamental role in nonlinear programming. It is used to define the optimality conditions that characterize a local minimum of the primal problem. It is also used to formulate the dual problem of the preceding primal problem. To this end, consider the following dual function:

\[
d(\lambda, \mu, v) = \inf_x L(x, \lambda, \mu, v)
\]

The dual problem is defined as

\[
\begin{align*}
\text{maximize} & \quad d(\lambda, \mu, v) \\
\text{subject to} & \quad \mu \leq 0 \\
& \quad v \geq 0.
\end{align*}
\]

The variables \( \lambda, \mu, \) and \( v \) are called the dual variables. Note that the dual variables associated with the equality constraints (\( \lambda \)) are free, whereas those associated with \( \leq \) inequality constraints (\( \mu \)) have to be nonpositive and those associated with \( \geq \) inequality constraints (\( v \)) have to be nonnegative.

The relation between the primal and the dual problems provides a nice connection between the optimal solutions of the problems. Suppose \( x^* \) is an optimal solution of the primal problem and \( (\lambda^*, \mu^*, v^*) \) is an optimal solution of the dual problem. The difference between the objective values of the primal and dual problems, \( \delta = f(x^*) - d(\lambda^*, \mu^*, v^*) \geq 0 \), is called the duality gap. For some restricted class of convex nonlinear programming problems, both the primal and the dual problems have an optimal solution and the optimal objective values are equal—that is, the duality gap \( \delta = 0 \). In such cases, the optimal values of the dual variables correspond to the optimal Lagrange multipliers of the primal problem with the correct signs.

A maximization problem is treated analogously to a minimization problem. For the maximization problem

\[
\begin{align*}
\text{maximize} & \quad f(x) \\
\text{subject to} & \quad c_i(x) = 0, \quad i \in E \\
& \quad c_i(x) \leq 0, \quad i \in \mathcal{L} \\
& \quad c_i(x) \geq 0, \quad i \in \mathcal{G},
\end{align*}
\]

the dual problem is

\[
\begin{align*}
\text{minimize} & \quad d(\lambda, \mu, v) \\
\text{subject to} & \quad \mu \geq 0 \\
& \quad v \leq 0,
\end{align*}
\]

where the dual function is defined as \( d(\lambda, \mu, v) = \sup_x L(x, \lambda, \mu, v) \) and the Lagrangian function \( L(x, \lambda, \mu, v) \) is defined the same as earlier. In this case, \( \lambda, \mu, \) and \( -v \) correspond to the Lagrange multipliers in \( L(x, \lambda, \mu, v) \).
**Minimization Problems**

For inequality constraints in minimization problems, a positive optimal dual value indicates that the associated \( \geq \) inequality constraint is active at the solution, and a negative optimal dual value indicates that the associated \( \leq \) inequality constraint is active at the solution. In PROC OPTMODEL, the optimal dual value for a range constraint (a constraint with both upper and lower bounds) is the sum of the dual values associated with the upper and lower inequalities. Since only one of the two inequalities can be active, the sign of the optimal dual value, if nonzero, identifies which one is active.

For equality constraints in minimization problems, the optimal dual values are unrestricted in sign. A positive optimal dual value for an equality constraint implies that, starting close enough to the primal solution, the same optimum could be found if the equality constraint were replaced with a \( \geq \) inequality constraint. A negative optimal dual value for an equality constraint implies that the same optimum could be found if the equality constraint were replaced with a \( \leq \) inequality constraint.

The following is an example where simple linear programming is considered:

```plaintext
proc optmodel;
var x, y;
min z = 6*x + 7*y;
con 4*x + y >= 5,
    -x - 3*y <= -4,
    x + y <= 4;
solve;
print x y;
expand _ACON_;
print _ACON_.dual _ACON_.body;
```

The PRINT statements generate the output shown in Figure 10.60.

**Figure 10.60** Dual Values in Minimization Problem: Display

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
<tr>
<td>Number of Variables</td>
</tr>
<tr>
<td>Bounded Above</td>
</tr>
<tr>
<td>Bounded Below</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
</tr>
<tr>
<td>Free</td>
</tr>
<tr>
<td>Fixed</td>
</tr>
<tr>
<td>Number of Constraints</td>
</tr>
<tr>
<td>Linear LE (&lt;=)</td>
</tr>
<tr>
<td>Linear EQ (=)</td>
</tr>
<tr>
<td>Linear GE (&gt;=)</td>
</tr>
<tr>
<td>Linear Range</td>
</tr>
<tr>
<td>Constraint Coefficients</td>
</tr>
</tbody>
</table>
It can be seen that the first and second constraints are active, with dual values 1 and –2. Continue to submit
the following statements. Notice how the objective value is changed in Figure 10.61.

```plaintext
_ACON_[1].lb = _ACON_[1].lb - 1;
solve;
_ACON_[2].ub = _ACON_[2].ub + 1;
solve;
```
Figure 10.61 Dual Values in Minimization Problem: Interpretation

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objective Sense</strong></td>
</tr>
<tr>
<td><strong>Objective Function</strong></td>
</tr>
<tr>
<td><strong>Objective Type</strong></td>
</tr>
<tr>
<td><strong>Number of Variables</strong></td>
</tr>
<tr>
<td><strong>Bounded Above</strong></td>
</tr>
<tr>
<td><strong>Bounded Below</strong></td>
</tr>
<tr>
<td><strong>Bounded Below and Above</strong></td>
</tr>
<tr>
<td><strong>Free</strong></td>
</tr>
<tr>
<td><strong>Fixed</strong></td>
</tr>
<tr>
<td><strong>Number of Constraints</strong></td>
</tr>
<tr>
<td><strong>Linear LE ($\leq$)</strong></td>
</tr>
<tr>
<td><strong>Linear EQ ($=$)</strong></td>
</tr>
<tr>
<td><strong>Linear GE ($\geq$)</strong></td>
</tr>
<tr>
<td><strong>Linear Range</strong></td>
</tr>
<tr>
<td><strong>Constraint Coefficients</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solver</strong></td>
</tr>
<tr>
<td><strong>Algorithm</strong></td>
</tr>
<tr>
<td><strong>Objective Function</strong></td>
</tr>
<tr>
<td><strong>Solution Status</strong></td>
</tr>
<tr>
<td><strong>Objective Value</strong></td>
</tr>
<tr>
<td><strong>Primal Infeasibility</strong></td>
</tr>
<tr>
<td><strong>Dual Infeasibility</strong></td>
</tr>
<tr>
<td><strong>Bound Infeasibility</strong></td>
</tr>
<tr>
<td><strong>Iterations</strong></td>
</tr>
<tr>
<td><strong>Presolve Time</strong></td>
</tr>
<tr>
<td><strong>Solution Time</strong></td>
</tr>
</tbody>
</table>
The change is just as the dual values imply. After the first constraint is relaxed by one unit, the objective value is improved by one unit. For the second constraint, the relaxation and improvement are one unit and two units, respectively.

**NOTE:** The signs of dual values produced by PROC OPTMODEL depend, in some instances, on the way in which the corresponding constraints are entered. See the section “Constraints” on page 376 for details.

## Maximization Problems

For inequality constraints in maximization problems, a positive optimal dual value indicates that the associated \( \leq \) inequality constraint is active at the solution, and a negative optimal dual value indicates that the associated \( \geq \) inequality constraint is active at the solution. The optimal dual value for a range constraint is the sum of
Reduced Costs

In linear programming problems, each variable has a corresponding reduced cost. To access the reduced cost of a variable, add the suffix `.rc` or `.dual` to the variable name. These two suffixes are interchangeable.

The reduced cost of a variable is the rate at which the objective value changes when the value of that variable changes. At optimality, basic variables have a reduced cost of zero; a nonbasic variable with zero reduced cost indicates the existence of multiple optimal solutions.

In nonlinear programming problems, the reduced cost interpretation does not apply. The `.dual` and `.rc` variable suffixes represent the gradient of the Lagrangian function, computed using the values returned by the solver.

The following example illustrates the use of the `.rc` suffix:

```plaintext
proc optmodel;
   var x >= 0, y >= 0, z >= 0;
   max cost = 4*x + 3*y - 5*z;
   con
      -x + y + 5*z <= 15,
      3*x - 2*y - z <= 12,
      2*x + 4*y + 2*z <= 16;
   solve;
   print x y z;
   print x.rc y.rc z.rc;
```

The PRINT statements generate the output shown in Figure 10.62.

**Figure 10.62** Reduced Cost in Maximization Problem: Display

```
x  y  z
5 1.5 0
```

```
x.RC  y.RC  z.RC
0 0 -6.5
```

In this example, `x` and `y` are basic variables, while `z` is nonbasic. The reduced cost of `z` is –6.5, which implies that increasing `z` from 0 to 1 decreases the optimal value from 24.5 to 18.
**Presolver**

PROC OPTMODEL includes a simple presolver that processes linear constraints to produce tighter bounds on variables. The presolver can reduce the number of variables and constraints that are presented to the solver. These changes can result in reduced solution times.

Linear constraints that involve only a single variable are converted into variable bounds. The presolver then eliminates redundant linear constraints for which variable bounds force the constraint to always be satisfied. Tightly bounded variables where upper and lower bounds are within the range specified by the VARFUZZ= option (see the section “PROC OPTMODEL Statement” on page 284) are automatically fixed to the average of the bounds. The presolver also eliminates variables that are fixed by the user or by the presolver.

The presolver can infer tighter variable bounds from linear constraints when all variables in the constraint or all but one variable have known bounds. For example, when given the following PROC OPTMODEL declarations, the presolver can use the inferred bounds $x \leq 7$ and $y \leq 4$ to remove the constraint $c2$:

```plaintext
proc optmodel;
  var x >= 3;
  var y >= 0;
  con c: x + y <= 7;
  con c2: x + 2*y <= 15; /* redundant */
```

The presolver makes multiple passes and rechecks linear constraints after bounds are tightened for the referenced variables. The number of passes is controlled by the PRESOLVER= option. After the passes are finished, the presolver attempts to fix the value of all variables that are not used in the updated objective and constraints. The current value of such a variable is used if the value lies between the variable’s upper and lower bounds. Otherwise, the value is adjusted to the nearer bound. The value of an integer variable is rounded before being checked against its bounds.

The presolver uses the tightened variable bounds to infer the range of values for nonlinear constraint bodies. In some cases, the presolver can determine that a nonlinear constraint is infeasible. The presolver can also remove a constraint that does not restrict the feasible region of the problem.

In some cases the solver might perform better without the presolve transformations, so almost all such transformations are unavailable when the option PRESOLVER=BASIC is specified. However, the presolver still eliminates variables that have values that have been fixed by the FIX statement. To disable the OPTMODEL presolver entirely, use PRESOLVER=None. The solver assigns values to any unused, unfixed variables when the option PRESOLVER=None is specified.

**Model Update**

The PROC OPTMODEL modeling language provides several means of modifying a model after it is first specified. You can change the parameter values of the model. You can add new model components. The FIX and UNFIX statements can fix variables to specified values or rescind previously fixed values. The DROP and RESTORE statements can deactivate and reactivate constraints. See also the section “Multiple Subproblems” on page 397 for information on how to maintain multiple models.
To illustrate how these statements work, reconsider the following example from the section “Constraints” on page 376:

```plaintext
proc optmodel;
  var x, y;
  min r = x**2 + y**2;
  con c: x+y >= 1;
  solve;
  print x y;
```

As described previously, the solver finds the optimal point $x = y = 0.5$ with $r = 0.5$. You can see the effect of the constraint $c$ on the solution by temporarily removing it. You can add the following statements:

```plaintext
drop c;
solve;
print x y;
```

This change produces the output in Figure 10.63.

**Figure 10.63** Solution with Dropped Constraint

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
<tr>
<td>Number of Variables</td>
</tr>
<tr>
<td>Bounded Above</td>
</tr>
<tr>
<td>Bounded Below</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
</tr>
<tr>
<td>Free</td>
</tr>
<tr>
<td>Fixed</td>
</tr>
<tr>
<td>Number of Constraints</td>
</tr>
<tr>
<td>Constraint Coefficients</td>
</tr>
<tr>
<td>Hessian Diagonal Elements</td>
</tr>
<tr>
<td>Hessian Elements Below Diagonal</td>
</tr>
</tbody>
</table>
The optimal point is $x = y = 0$, as expected.

You can restore the constraint $c$ with the RESTORE statement, and you can also investigate the effect of forcing the value of variable $x$ to 0.3. This requires the following statements:

```plaintext
restore c;
fix x=0.3;
solve;
print x y c.dual;
```

This produces the output in Figure 10.64.
The variable $x$ still has the value that was defined in the FIX statement. The objective value has increased by 0.08 from its constrained optimum 0.5 (see Figure 10.55). The constraint $c$ is active, as confirmed by the positive dual value.
You can return to the original optimization problem by allowing the solver to vary variable \( x \) with the UNFIX statement, as follows:

```plaintext
unfix x;
solve;
print x y c.dual;
```

This produces the output in Figure 10.65. The model was returned to its original conditions.

**Figure 10.65** Solution with Original Model

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objective Sense</strong></td>
</tr>
<tr>
<td><strong>Objective Function</strong></td>
</tr>
<tr>
<td><strong>Objective Type</strong></td>
</tr>
<tr>
<td><strong>Number of Variables</strong></td>
</tr>
<tr>
<td><strong>Bounded Above</strong></td>
</tr>
<tr>
<td><strong>Bounded Below</strong></td>
</tr>
<tr>
<td><strong>Bounded Below and Above</strong></td>
</tr>
<tr>
<td><strong>Free</strong></td>
</tr>
<tr>
<td><strong>Fixed</strong></td>
</tr>
<tr>
<td><strong>Number of Constraints</strong></td>
</tr>
<tr>
<td><strong>Linear LE ((&lt;=))</strong></td>
</tr>
<tr>
<td><strong>Linear EQ (=)</strong></td>
</tr>
<tr>
<td><strong>Linear GE ((&gt;=))</strong></td>
</tr>
<tr>
<td><strong>Linear Range</strong></td>
</tr>
<tr>
<td><strong>Constraint Coefficients</strong></td>
</tr>
<tr>
<td><strong>Hessian Diagonal Elements</strong></td>
</tr>
<tr>
<td><strong>Hessian Elements Below Diagonal</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solver</strong></td>
</tr>
<tr>
<td><strong>Algorithm</strong></td>
</tr>
<tr>
<td><strong>Objective Function</strong></td>
</tr>
<tr>
<td><strong>Solution Status</strong></td>
</tr>
<tr>
<td><strong>Objective Value</strong></td>
</tr>
<tr>
<td><strong>Primal Infeasibility</strong></td>
</tr>
<tr>
<td><strong>Dual Infeasibility</strong></td>
</tr>
<tr>
<td><strong>Bound Infeasibility</strong></td>
</tr>
<tr>
<td><strong>Duality Gap</strong></td>
</tr>
<tr>
<td><strong>Complementarity</strong></td>
</tr>
<tr>
<td><strong>Iterations</strong></td>
</tr>
<tr>
<td><strong>Presolve Time</strong></td>
</tr>
<tr>
<td><strong>Solution Time</strong></td>
</tr>
</tbody>
</table>
Multiple Subproblems

The OPTMODEL procedure enables multiple models to be manipulated easily by using named problems to switch the active model components. Problems keep track of an objective, a set of included variables and constraints, and some status information that is associated with the variables and constraints. Other data, such as parameter values, bounds, and the current value of variables, are shared by all problems.

Problems are declared with the PROBLEM declaration. You can easily switch between problems by using the USE PROBLEM statement. The USE PROBLEM statement makes the specified problem become the current problem. The various statements that generate problem data, such as SOLVE, EXPAND, and SAVE MPS, always operate using the model components included in the current problem.

A problem declaration can specify the problem’s objectives by copying them from the problem that is named in a FROM clause or by including objective symbols. The objectives can be updated while the problem is current by declaring a new non-array objective or by executing programming statements that specify new objectives.

Variables can also be included when the problem is current by declaring them or by using the FIX or UNFIX statement. Similarly, constraints can be included when the problem is current by declaring them or by using the RESTORE or DROP statement. There is no way to exclude a variable or constraint item after it has been included in a problem, although the variable or constraint can be fixed or dropped.

Variables that are declared but not included in a problem are treated as constants when a problem is generated, while constraints that are declared but not included are ignored. The solver does not update the values and status for these model components.

A problem also tracks certain other status information that is associated with its included symbols, and this information can be changed without affecting other problems. This information includes the fixed status for variables, and the dropped status for constraints. The following additional data that are tracked by the problem are available through variable and constraint suffixes:

- var.STATUS (including IIS status)
- var.SOL[i] (for each solution i)
- var.MSINIT
- var.RC
- var.DUAL (alias of var.RC)
- var.FIXED
- con.STATUS (including IIS status)
- con.DUAL
- con.BLOCK
The initial problem when OPTMODEL starts is predeclared with the name _START_. This problem can be reinstated again (after other USE PROBLEM statements) with the statement

```plaintext
use problem _start_;
```

See “Example 10.5: Multiple Subproblems” on page 422 for example statements that use multiple subproblems.

### Multiple Solutions

When a solver finishes, it reports zero or more solutions for the optimization variables of the current problem. Each solution assigns a value to each of the variables in the problem. The SOLVE statement saves these solutions with the current problem. The first reported solution, if any, is also copied into the optimization variables. The number of solutions is available in the predeclared numeric parameter _NSOL_.

**NOTE:** The network solver does not require optimization variables and has its own conventions for returning multiple solutions.

You can access the solutions that are saved with the problem by adding a solution index to the .sol suffix of the variable name. For example, x.sol[2] would reference the second solution saved for variable x in the current problem. Both the variable name and the suffix can be indexed. For example, assign[3,7].sol[1] refers to the first solution for the array variable element assign[3,7]. The solution index must be an integer in the range 1 to _NSOL_.

The following example illustrates the processing of multiple solutions from the CLP solver:

```plaintext
proc optmodel printlevel=0;
  var x{1..2} integer >= 1 <= 3;
  con c: alldiff(x);
  solve with clp / allsolns;
  print _NSOL_;
  print {j in 1..2, i in 1.._NSOL_} x[j].sol[i];
  create data solout from 
  [sol]={i in 1.._NSOL_}
  {j in 1..2} <col("x"||j)=x[j].sol[i]> ;
```

This program produces the output in Figure 10.66. It also creates a data set, solout, which has each solution in a separate observation.

**Figure 10.66** Processing Multiple Solutions

<table>
<thead>
<tr>
<th><em>NSOL</em></th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>x.SOL</td>
<td>1 2 3 4 5 6</td>
</tr>
<tr>
<td></td>
<td>1 1 1 2 2 3 3</td>
</tr>
<tr>
<td></td>
<td>2 2 3 1 3 1 2</td>
</tr>
</tbody>
</table>
Problem Symbols

The OPTMODEL procedure declares a number of symbols that are aliases for model components in the current problem. These symbols allow the model components to be accessed uniformly. These symbols are described in Table 10.17.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Indexing</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>NVAR</em></td>
<td></td>
<td>Number of variables</td>
</tr>
<tr>
<td><em>VAR</em></td>
<td>1..<em>NVAR</em></td>
<td>Variable array</td>
</tr>
<tr>
<td><em>NCON</em></td>
<td></td>
<td>Number of constraints</td>
</tr>
<tr>
<td><em>CON</em></td>
<td>1..<em>NCON</em></td>
<td>Constraint array</td>
</tr>
<tr>
<td><em>S_NVAR</em></td>
<td></td>
<td>Number of presolved variables</td>
</tr>
<tr>
<td><em>S_VAR</em></td>
<td>1..<em>S_VAR</em></td>
<td>Presolved variable array</td>
</tr>
<tr>
<td><em>S_NCON</em></td>
<td></td>
<td>Number of presolved constraints</td>
</tr>
<tr>
<td><em>S_CON</em></td>
<td>1..<em>S_CON</em></td>
<td>Presolved constraint array</td>
</tr>
<tr>
<td><em>OBJ</em></td>
<td></td>
<td>Current objective</td>
</tr>
<tr>
<td><em>PROBLEM</em></td>
<td></td>
<td>Current problem</td>
</tr>
</tbody>
</table>

If the table specifies indexing, then the corresponding symbol is accessed as an array. For example, if the problem includes two variables, x and y, then the value of _NVAR_ is 2 and the current variable values can be accessed as _var_[1] and _var_[2]. The problem variables prefixed with _S are restricted to model components in the problem after processing by the OPTMODEL presolver.

The following statements define a simple linear programming model and then use the problem symbols to print out some of the problem results. The .name suffix is used in the PRINT statements to display the actual variable and constraint names. Any of the suffixes that apply to a model component can be applied to the corresponding generic symbol.

```latex
proc optmodel printlevel=0;
  var x1 >= 0, x2 >= 0, x3 >= 0, x4 >= 0, x5 >= 0;

  minimize z = x1 + x2 + x3 + x4;

  con a1: x1 + x2 + x3 <= 4;
  con a2: x4 + x5 <= 6;
  con a3: x1 + x4 >= 5;
  con a4: x2 + x5 >= 2;
  con a5: x3 >= 3;

  solve with lp;

  print _var_.name _var_ _var_.rc _var_.status;
  print _con_.name _con_.lb _con_.body _con_.ub _con_.dual _con_.status;
```

The PRINT statement output is shown in Figure 10.67.
OPTMODEL Options

All PROC OPTMODEL options can be specified in the PROC statement (see the section “PROC OPTMODEL Statement” on page 284 for more information). However, it is sometimes necessary to change options after other PROC OPTMODEL statements have been executed. For example, if an optimization technique had trouble with convergence, then it might be useful to vary the PRESOLVER= option value. This can be done with the RESET OPTIONS statement.

The RESET OPTIONS statement accepts options in the same form used by the PROC OPTMODEL statement. The RESET OPTIONS statement is also able to reset option values and to change options programmatically. For example, the following statements print the value of parameter $n$ at various precisions:

```latex
proc optmodel;
  number n = 1/7;
  for {i in 1..9 by 4}
    do;
      reset options pdigits=(i);
      print n;
    end;
  reset options pdigits; /* reset to default */
```

The output generated is shown in Figure 10.68. The RESET OPTIONS statement in the DO loop sets the PDIGITS option to the value of i. The final RESET OPTIONS statement restores the default option value, because the value was omitted.

**Figure 10.68 Changing the PDIGITS Option Value**

```
  n
  0.1

  n
  0.14286

  n
  0.142857143
```
Automatic Differentiation

PROC OPTMODEL automatically generates statements to evaluate the derivatives for most objective expressions and nonlinear constraints. PROC OPTMODEL generates analytic derivatives for objective and constraint expressions written in terms of the procedure’s mathematical operators and most standard SAS library functions.

**Note:** Some functions, such as ABS, FLOOR, and SIGN, and some operators, such as IF-THEN, &gt;&lt; (maximum operator), and &gt;&gt; (minimum operator), must be used carefully in modeling expressions because functions that include such components are not continuously differentiable or even continuous.

Expressions that reference user-defined functions or some SAS library functions might require numerical approximation of derivatives. PROC OPTMODEL uses either forward-difference approximation or central-difference approximation as specified by the FD= option (see the section “PROC OPTMODEL Statement” on page 284).

**Note:** The numerical gradient approximations are significantly slower than automatically generated derivatives when the number of optimization variables is large.

Forward-Difference Approximations

The FD=FORWARD option requests the use of forward-difference derivative approximations. For a function $f$ of $n$ variables, the first-order derivatives are approximated by

$$
    g_i = \frac{\partial f}{\partial x_i} = \frac{f(x + e_i h_i) - f(x)}{h_i}
$$

Notice that up to $n$ additional function calls are needed here. The step lengths $h_i$, $i = 1, \ldots, n$, are based on the assumed function precision, DIGITS:

$$
    h_i = 10^{-\text{DIGITS}/2}(1 + |x_i|)
$$

You can use the FDIGITS= option to specify the function precision, DIGITS, for the objective function. For constraints, use the CDIGITS= option.

The second-order derivatives are approximated by using up to $n(n + 3)/2$ extra function calls (Dennis and Schnabel 1983, pp. 80, 104):

$$
    \frac{\partial^2 f}{\partial x_i^2} = \frac{f(x + h_i e_i) - 2f(x) + f(x - h_i e_i)}{h_i^2}
$$

$$
    \frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i) - f(x + h_j e_j) + f(x)}{h_i h_j}
$$
Notice that the diagonal of the Hessian uses a central-difference approximation (Abramowitz and Stegun 1972, p. 884). The step lengths are

\[ h_i = 10^{-DIGITS/3} (1 + |x_i|) \]

**Central-Difference Approximations**

The FD=CENTRAL option requests the use of central-difference derivative approximations. Generally, central-difference approximations are more accurate than forward-difference approximations, but they require more function evaluations. For a function \( f \) of \( n \) variables, the first-order derivatives are approximated by

\[ g_i = \frac{\partial f}{\partial x_i} = \frac{f(x + e_i h_i) - f(x - e_i h_i)}{2h_i} \]

Notice that up to \( 2n \) additional function calls are needed here. The step lengths \( h_i, i = 1, \ldots, n \), are based on the assumed function precision, \( DIGITS \):

\[ h_i = 10^{-DIGITS/3} (1 + |x_i|) \]

You can use the FDIGITS= option to specify the function precision, \( DIGITS \), for the objective function. For constraints, use the CDIGITS= option.

The second-order derivatives are approximated by using up to \( 2n(n + 1) \) extra function calls (Abramowitz and Stegun 1972, p. 884):

\[
\frac{\partial^2 f}{\partial x_i^2} = \frac{-f(x + 2h_i e_i) + 16f(x + h_i e_i) - 30f(x) + 16f(x - h_i e_i) - f(x - 2h_i e_i)}{12h_i^2} \\
\frac{\partial^2 f}{\partial x_i \partial x_j} = \frac{f(x + h_i e_i + h_j e_j) - f(x + h_i e_i - h_j e_j) - f(x - h_i e_i + h_j e_j) + f(x - h_i e_i - h_j e_j)}{4h_i h_j}
\]

The step lengths are

\[ h_i = 10^{-DIGITS/3} (1 + |x_i|) \]
Conversions

Numeric values are implicitly converted to strings when needed for function arguments or operands to the string concatenation operator (||). A warning message is generated when the conversion is applied to a function argument. The conversion uses BEST12. format. Unlike the DATA step, the conversion trims blanks.

Implicit conversion of strings to numbers is not permitted. Use the INPUT function to explicitly perform such conversions.

FCMP Routines

The OPTMODEL procedure can call functions and subroutines that are compiled by the FCMP procedure. You can use FCMP functions wherever a function expression is allowed in PROC OPTMODEL. Use the CALL statement to call FCMP subroutines. The following example defines a function in the FCMP procedure and calls it within PROC OPTMODEL:

```latex
proc fcmp outlib=work.funcs.test;
  /* arithmetic geometric mean */
  function agm(a0, b0);
    a = a0; b = b0;
    if a<=0 or b<=0 then return(0);
    do until( a - b < a/1e12 );
      a1 = 0.5*a + 0.5*b;
      b1 = sqrt(a*b);
      a = a1; b = b1;
    end;
    return(a);
  endsub;
run;

/* libraries must be specified with the CMPLIB option */
option cmplib=work.funcs;

proc optmodel;
  print (agm(1,2));

  /* find x where agm(1,x) == 23 */
  var x init 1;
  num c = 23;
  min z = (agm(1,x)-c)^2;
  solve;
  print x;
```

FCMP subroutines can return data by updating OPTMODEL numeric and string parameters, which are passed as arguments in a CALL statement. These arguments are declared using the OUTARGS statement in the PROC FCMP subroutine definition. The OPTMODEL argument must be specified with an identifier expression. The following code shows a simple example of output arguments. The maximum length of output strings from OUTARGS arguments is restricted to the argument length before the call, as described in the section “CALL Statement” on page 298.
You can pass OPTMODEL arrays to FCMP functions and subroutines that accept matrix arguments. The array must match the type and dimensions of the FCMP argument declaration. The argument in the OPTMODEL CALL statement must be specified using the following syntax:

\[
\text{array-name} [ . \text{suffix} ]
\]

The following code passes a constant matrix to an FCMP function. The array `coeff` contains the coefficients of a polynomial, which in this case defines a simple quadratic formula, \(x^2 - 2x + 1\).

```plaintext
proc fcmp outlib=work.funcs.test;
    function evalpoly(x, coeff[*]);
        z = 0;
        do i = dim1(coeff) to 1 by -1;
            z = z * x + coeff[i];
        end;
        return (z);
    endsub;
run;

option cmplib=work.funcs;

proc optmodel;
    num coeff{1..3} = [1, -2, 1];
    var x;
    min z=evalpoly(x, coeff);
    solve;
    print x;
run;
```

An array that is used as a matrix argument must be indexed like an FCMP matrix. In other words, the array index set must be specified as the crossproduct of one or more range expressions (such as 1..N) where the
lower bound and step size are 1. Set parameters that are used for indexing must contain a crossproduct of ranges, but the element order is not important. The following code shows some examples of suitable and unsuitable array declarations:

```sas
proc fcmp outlib=work.funcs.test;
    subroutine mattest(x[*]);
        put x[1]=;
    endsub;
    subroutine mattest2(x[*,*]);
        put x[1,1]=;
    endsub;
run;

option cmplib=work.funcs;

proc optmodel;
    /* the following arrays can be used as matrices */
    num N init 3;
    num mat1{1..N} init 0;
    call mattest(mat1); /* OK */
    set S1 = 1..5;
    num mat2{S1} init 0;
    call mattest(mat2); /* OK */
    set S2 = {S1,S1};
    num mat3{S2} init 0;
    call mattest2(mat3); /* OK */
    num mat4{S1 cross S1} init 0;
    call mattest2(mat4); /* OK */
    num L init 1;
    num mat5{L..N} init 0;
    call mattest(mat5); /* OK */
    set S3 init S1;
    num mat6{S3} init 0;
    call mattest(mat6); /* OK */

    /* some errors are detected at execution time */
    S3 = 2..5;
    call mattest(mat6); /* ERROR: lower bound not 1 */
    S3 = {1, 3, 4, 5};
    call mattest(mat6); /* ERROR: missing index 2 */
    L = 0;
    call mattest(mat5); /* ERROR: lower bound not 1 */

    /* the following arrays cannot be used as matrices */
    num arr1{1..10 by 3}; /* step size is not 1 */
    call mattest(arr1); /* ERROR */
    num arr2{i in 1..N, j in 1..N: j >= i}; /* selection expression used */
    call mattest2(arr2); /* ERROR */
    num arr3{i in 1..N, j in 1..i}; /* index dependency on 'i' */
    call mattest2(arr3); /* ERROR */
```

Not all PROC FCMP functionality is compatible with PROC OPTMODEL; in particular, the following FCMP functions are not supported and should not be called within your FCMP function definitions: READ_ARRAY, WRITE_ARRAY, RUN_MACRO, and RUN_SASFILE. In many cases, OPTMODEL capabilities can replace
these functions. Matrix arguments can be used in place of the READ_ARRAY function by using the READ
DATA statement to load the matrix in PROC OPTMODEL. Similarly, you can replace the WRITE_ARRAY
function in an FCMP subroutine by copying the matrix to an output argument and using the OPTMODEL
procedure to write the matrix. You can use the SUBMIT statement in place of the RUN_MACRO and
RUN_SASFILE functions.

The SAS CMPLIB= system option specifies where to look for previously compiled functions and subroutines.
For more information about the CMPLIB= system option, see SAS System Options: Reference. FCMP
functions can be used in distributed mode with the NLP multistart or LSO solver. The needed PROC FCMP
compiled routines are automatically packaged and distributed. For more information about the multistart

PROC OPTMODEL uses derivatives values that are provided by FCMP when they are available. FCMP can-
not provide derivatives with respect to array arguments, so PROC OPTMODEL must use finite differences to
compute these derivatives. Also, if the CMPOPT= SAS system option specifies the FUNCDIFFERENCING
value, then PROC OPTMODEL uses its own finite differencing for FCMP functions.

More on Index Sets

Dummy parameters behave like parameters but are assigned values only when an index set is evaluated. You
can reference the declared dummy parameters from index set expressions that follow the index set item. You
can also reference the dummy parameters in the expression or statement controlled by the index set. As the
members of the set expression of an index set item are enumerated, the element values of the members are
assigned to the local dummy parameters.

The number of names in a dummy parameter declaration must match the element length of the corresponding
set expression in the index set item. A single name is allowed when the set member type is scalar (numeric or
string). If the set members are tuples that have \( n > 1 \) elements, then \( n \) names are required between the angle
brackets (\(< >\)) that precede the IN keyword.

Multiple index set items in an index set are nominally processed in a left-to-right order. That is, a set
expression from an index set item is evaluated as though the index set items that precede it have already been
evaluated. The left-hand index set items can assign values to local dummy parameters that are used by the set
expressions that follow them. After each member from the set expression is enumerated, any index set items
to the right are reevaluated as needed. The actual order in which index set items are evaluated can vary, if
necessary, to allow more efficient enumeration. PROC OPTMODEL generates the same set of values in any
case, although possibly in a different order than strict left-to-right evaluation.

You can view the element combinations that are generated from an index set as tuples. This is especially true
for index set expressions (see the section “Index Set Expression” on page 354). However, in most cases no
tuple set is actually formed, and the element values are assigned only to local dummy parameters.

You can specify a selection expression following a colon (:). The index set generates only those combinations
of values for which the selection expression is true. For example, the following statements produce a set of
upper triangular indices:

```plaintext
    proc optmodel;
    put (setof {i in 1..3, j in 1..3 : j >= i} <i, j>);
```

These statements produce the output in Figure 10.70.
You can use the left-to-right evaluation of index set items to express the previous set more compactly. The following statements produce the same output as the previous statements:

```plaintext
proc optmodel;
  put ({i in 1..3, i..3});
```

In this example, the first time the second index set item is evaluated, the value of the dummy parameter i is 1, so the item produces the set {1,2,3}. At the second evaluation the value of i is 2, so the second item produces the set {2,3}. At the final evaluation the value of i is 3, so the second item produces the set {3}.

In many cases it is useful to combine the SLICE operator with index sets. A special form of index set item uses the SLICE operator implicitly. Normally an index set item that is applied to a set of tuples of length greater than one must be of the form

```
< name-1 [, . . . , name-n ] > IN set-expression
```

In the special form, one or more of the name elements are replaced by expressions. The expressions select tuple elements by using the SLICE operator. An expression that consists of a single name must be enclosed in parentheses to distinguish it from a dummy parameter. The remaining names are the dummy parameters for the index set item that is applied to the SLICE result. The following example demonstrates the use of implicit set slicing:

```plaintext
proc optmodel;
  number N = 3;
  set<num,str> S = {<1,'a'>,<2,'b'>,<3,'a'>,<4,'b'>};
  put ({i in 1..N, <(i),j> in S});
  put ({i in 1..N, j in slice(<i,*>, S)});
```

The two PUT statements in this example are equivalent.

---

**Threaded and Distributed Processing**

The OPTMODEL procedure can take advantage of the multiple CPUs that are available in many computers. PROC OPTMODEL automatically uses multithreaded execution to divide problem generation among the multiple CPUs of the computer that is running the procedure. Hessian and Jacobian matrix evaluation is automatically parallelized across threads of execution on multiple CPUs. The COFOR statement enables solvers to concurrently execute in background threads on multiple CPUs, overlapping with PROC OPTMODEL statement processing. Parallel execution can decrease the amount of clock time required to perform a task, although the total CPU time required might increase.

Optimization solvers execute in the distributed computing environment when you specify the SESSREF= or SESSUUID= option. Distributed computing provides the most benefit when you can structure a computational process to include a large number of nontrivial subprocesses that can be executed independently. Solvers can run on the distributed computing environment by using one or more threads of a compute node. The COFOR statement allows concurrent processing by multiple solvers. Alternatively, some solvers can distribute their processing across multiple compute nodes.
When you do not request distributed computing, threaded processing is performed on the client. The NTHREADS= option controls the number of threads that are used for processing. When you do not specify the NTHREADS= option, threading in the OPTMODEL procedure is controlled by the following SAS system options:

\[
\text{CPUCOUNT=} \text{number | ACTUAL}
\]

specifies the maximum number of CPUs that can be used.

\[
\text{THREADS | NOTHREADS}
\]

enables or disables the use of threading.

Good performance is usually obtained with the default option settings (THREADS and CPUCOUNT=ACTUAL). See the option descriptions in \textit{SAS System Options: Reference} for more details.

When distributed computing is used, the NTHREADS= option is ignored by problem generation because it is performed on the client. In this case, the maximum number of threads that are used for problem generation is based on the SAS CPUCOUNT= and THREADS options.

The NTHREADS= option and the SAS system options set the maximum number of threads. The number of threads that PROC OPTMODEL actually uses depends on the characteristics of the problem that is being solved and the requirements of the distributed computing environment. In particular, threading is not used when the problem is simple enough that threading offers no advantage.

---

### Macro Variable _OROPTMODEL_

The OPTMODEL procedure creates a macro variable named _OROPTMODEL_. You can inspect the execution of the most recently invoked solver from the value of the macro variable. The macro variable is defined at the start of the procedure and updated after each SOLVE statement is executed. The OPTMODEL procedure also updates the macro variable when an error is detected.

The _OROPTMODEL_ value is a string that consists of several “KEYWORD=value” items in sequence, separated by blanks; for example:

```
STATUS=OK ALGORITHM=DS SOLUTION_STATUS=OPTIMAL OBJECTIVE=119302.04331
PRIMAL_INFEASIBILITY=3.552714E-13 DUAL_INFEASIBILITY=2.273737E-13
BOUND_INFEASIBILITY=0 ITERATIONS=82 PRESOLVE_TIME=0.02 SOLUTION_TIME=0.05
```

The information contained in _OROPTMODEL_ varies according to which solver was last called. For lists of keywords and possible values, see the individual solver chapters.

If a value has not been computed, then the corresponding element is not included in the value of the macro variable. When PROC OPTMODEL starts, for example, the macro variable value is set to “STATUS=OK” because no SOLVE statement has been executed. If the STATUS= indicates an error, then the other values from the solver might not be available, depending on when the error occurred.

---

### Solver Status Parameters

In addition to creating the macro variable _OROPTMODEL_, the OPTMODEL procedure creates several predeclared parameters to provide simple access to solver status values. These parameters are declared as follows:
The value of _STATUS_ is equal to the STATUS= component of the _OROPTMODEL_ macro variable. The value of _STATUS_ is initially “OK”. The value is updated during the SOLVE statement and after statement execution errors.

The value of _SOLUTION_STATUS_ is equal to the SOLUTION_STATUS= component of the _OROPTMODEL_ macro variable. The value is initially an empty string. The value is updated during the SOLVE statement.

You can use the remaining status parameters to access all the components of the _OROPTMODEL_ macro variable. The following statements demonstrate these parameters:

```optmodel
proc optmodel printlevel=0;
   var x init 1 >= 0.001;
   min z=sin(x)/x;
   solve;
   for {k in /STATUS SOLUTION_STATUS ALGORITHM/}
      put _OROPTMODEL_STR_[k]=;
   for {k in /OBJECTIVE ITERATIONS/}
      put _OROPTMODEL_NUM_[k]=;
```

These statements produce the output in Figure 10.71.

**Figure 10.71** Solver Status Parameters

```
_OROPTMODEL_STR_[STATUS]=OK
_OROPTMODEL_STR_[SOLUTION_STATUS]=OPTIMAL
_OROPTMODEL_STR_[ALGORITHM]=IP
_OROPTMODEL_NUM_[OBJECTIVE]=-0.217233628
_OROPTMODEL_NUM_[ITERATIONS]=3
```

The _OROPTMODEL_STR_ array contains the same character component values that are found in the _OROPTMODEL_ macro variable. Specify the component name as the array index. For example, the indices “STATUS” and “SOLUTION_STATUS” select array elements that hold the STATUS= and SOLUTION_STATUS= component values, respectively. The set _OROPTMODEL_STR_KEY_ contains the component indices that you can use with _OROPTMODEL_STR_. The OPTMODEL procedure updates the _OROPTMODEL_STR_ and _OROPTMODEL_STR_KEY_ parameters during the execution of the SOLVE statement and after any execution errors occur.

The _OROPTMODEL_NUM_ array contains the numeric component values that are displayed in the _OROPTMODEL_ macro variable. For example, the index “OBJECTIVE” selects the array element that holds the objective value when a solution is available. The set _OROPTMODEL_NUM_KEY_ contains the component indices that you can use with _OROPTMODEL_NUM_. The OPTMODEL procedure updates the _OROPTMODEL_NUM_ and _OROPTMODEL_NUM_KEY_ parameters during the execution of the SOLVE statement and after any execution errors occur.
Macro and Statement Evaluation Order

PROC OPTMODEL reads a complete statement, such as a DO statement, before executing any code in it. But macro language statements are processed as the code is read. So you must be careful when using the _OROPTMODEL_ macro variable in code that involves SOLVE statements nested in loops or DO statements. The following statements demonstrate one example of this behavior:

```plaintext
proc optmodel;
  var x, y;
  min z=x**2 + (x*y-1)**2;
  for {n in 1..3} do;
    fix x=n;
    solve;
    %put Line 1 &_OROPTMODEL_;
    put 'Line 2 ' (symget("_OROPTMODEL_"));
  end;
quit;
```

In the preceding statements the %PUT statement is executed once, before any SOLVE statements are executed. It displays PROC OPTMODEL's initial setting of the macro variable. But the PUT statement is executed after each SOLVE statement and indicates the expected solution status.

Rewriting PROC NLP Models for PROC OPTMODEL

This section describes techniques for converting PROC NLP models to PROC OPTMODEL models. Example 10.8 also demonstrates how to rewrite a PROC NLP model for use with PROC OPTMODEL.

To illustrate the basics, consider the following first version of the PROC NLP model for the example “Simple Pooling Problem” in Chapter 6, “The NLP Procedure” (SAS/OR User’s Guide: Mathematical Programming Legacy Procedures):

```plaintext
proc nlp all;
   parms amountx amounty amounta amountb amountc
       pooltox pooltoy ctox ctoy pools = 1;
   bounds 0 <= amountx amounty amounta amountb amountc,
       amountx <= 100,
       amounty <= 200,
       0 <= pooltox pooltoy ctox ctoy,
       1 <= pools <= 3;
   lincon amounta + amountb = pooltox + pooltoy,
       pooltox + ctox = amountx,
       pooltoy + ctoy = amounty,
       ctox + ctoy = amountc;
   nlincon nlc1-nlc2 >= 0.,
       nlc3 = 0.;
   max f;
   costa = 6; costb = 16; costc = 10;
   costx = 9; costy = 15;
   f = costx * amountx + costy * amounty
       - costa * amounta - costb * amountb - costc * amountc;
   nlc1 = 2.5 * amountx - pools * pooltox - 2. * ctox;
   nlc2 = 1.5 * amounty - pools * pooltoy - 2. * ctoy;
   nlc3 = 3 * amounta + amountb - pools * (amounta + amountb);
run;
```
These statements define a model that has bounds, linear constraints, nonlinear constraints, and a simple objective function. The following statements are a straightforward conversion of the PROC NLP statements to PROC OPTMODEL form:

```plaintext
proc optmodel;
  var amountx init 1 >= 0 <= 100,
      amounty init 1 >= 0 <= 200;
  var amounta init 1 >= 0,
      amountb init 1 >= 0,
      amountc init 1 >= 0;
  var pooltox init 1 >= 0,
      pooltoy init 1 >= 0;
  var ctox init 1 >= 0,
      ctoy init 1 >= 0;
  var pools init 1 >=1 <= 3;
  con amounta + amountb = pooltox + pooltoy, 
      pooltox + ctox = amountx, 
      pooltoy + ctoy = amounty, 
      ctox + ctoy = amountc;
  number costa, costb, costc, costx, costy;
  costa = 6; costb = 16; costc = 10;
  costx = 9; costy = 15;
  max f = costx * amountx + costy * amounty 
       - costa * amounta - costb * amountb - costc * amountc;
  con nlc1: 2.5 * amountx - pools * pooltox - 2. * ctox >= 0, 
          nlc2: 1.5 * amounty - pools * pooltoy - 2. * ctoy >= 0, 
          nlc3: 3 * amounta + amountb - pools * (amounta + amountb) 
                = 0;
  solve;
  print amountx amounty amounta amountb amountc 
       pooltox pooltoy ctox ctoy pools;
```

The PROC OPTMODEL variable declarations are split into individual declarations because PROC OPTMODEL does not permit name lists in its declarations. In the OPTMODEL procedure, you specify variable bounds as part of the variable declaration instead of in a separate BOUNDS statement. The PROC NLP statements are as follows:

```plaintext
parms amountx amounty amounta amountb amountc 
          pooltox pooltoy ctox ctoy pools = 1;
bounds 0 <= amountx amounty amounta amountb amountc, 
       amountx <= 100, 
       amounty <= 200, 
       0 <= pooltox pooltoy ctox ctoy, 
       1 <= pools <= 3;
```

The following PROC OPTMODEL statements are equivalent to the preceding PROC NLP statements:

```plaintext
var amountx init 1 >= 0 <= 100,
    amounty init 1 >= 0 <= 200;
var amounta init 1 >= 0,
    amountb init 1 >= 0,
    amountc init 1 >= 0;
var pooltox init 1 >= 0,
    pooltoy init 1 >= 0;
```
Chapter 10: The OPTMODEL Procedure

var ctox init 1 >= 0,
    ctoy init 1 >= 0;
var pools init 1 >= 1 <= 3;

The linear constraints are declared in the PROC NLP model by using the following statement:

\begin{align*}
\text{lincon amounta + amountb} & = \text{pooltox + pooltoy}, \\
\text{pooltox + ctox} & = \text{amountx}, \\
\text{pooltoy + ctoy} & = \text{amounty}, \\
\text{ctox + ctoy} & = \text{amountc};
\end{align*}

The following linear constraint declarations in the PROC OPTMODEL model are quite similar to the PROC NLP LINCON declarations:

\begin{align*}
\text{con amounta + amountb} & = \text{pooltox + pooltoy}, \\
\text{pooltox + ctox} & = \text{amountx}, \\
\text{pooltoy + ctoy} & = \text{amounty}, \\
\text{ctox + ctoy} & = \text{amountc};
\end{align*}

But PROC OPTMODEL provides much more flexibility in defining linear constraints. For example, a coefficient can be a named parameter or any other expression that evaluates to a constant.

The cost parameters are declared explicitly in the PROC OPTMODEL model. Unlike the DATA step or the NLP procedure, PROC OPTMODEL requires names to be declared before they are used. There are multiple ways to set the values of these parameters. The preceding example uses assignments. You could make the values part of the declaration by using the \texttt{INIT expression} clause or the \texttt{=} \texttt{expression} clause. You could also read the values from a data set by using the \texttt{READ DATA} statement.

In the original PROC NLP statements, the assignment to a parameter such as \texttt{costa} occurs every time the objective function is evaluated. However, the assignment occurs just once in the PROC OPTMODEL statements, when the assignment statement is processed. This works because the values are constant. But the PROC OPTMODEL statements permit the parameters to be reassigned later so that you can interactively modify the model.

The following statements define the objective \( f \) in the PROC NLP model:

\begin{align*}
\text{max } f; \\
\ldots \\
\text{f} & = \text{costx} \times \text{amountx} + \text{costy} \times \text{amounty} \\
& \quad - \text{costa} \times \text{amounta} - \text{costb} \times \text{amountb} - \text{costc} \times \text{amountc};
\end{align*}

The PROC OPTMODEL version of the objective is defined by using the same expression text, as follows:

\begin{align*}
\text{max } f & = \text{costx} \times \text{amountx} + \text{costy} \times \text{amounty} \\
& \quad - \text{costa} \times \text{amounta} - \text{costb} \times \text{amountb} - \text{costc} \times \text{amountc};
\end{align*}

But the \texttt{MAX} statement and the assignment to the name \( f \) in the PROC NLP statements are combined in PROC OPTMODEL. There are advantages and disadvantages to this approach. The PROC OPTMODEL formulation is much closer to the mathematical formulation of the model. However, if multiple intermediate variables are used to structure the objective, then multiple \texttt{IMPVAR} declarations are required.

In the PROC NLP model, the nonlinear constraints use the following syntax:
In the PROC OPTMODEL model, the equivalent statements are as follows:

\[
\begin{align*}
nl1 &: 2.5 \times \text{amountx} - \text{pools} \times \text{pooltox} - 2. \times \text{ctox} \geq 0, \\
nl2 &: 1.5 \times \text{amounty} - \text{pools} \times \text{pooltoy} - 2. \times \text{ctoy} \geq 0, \\
nl3 &: 3 \times \text{amounta} + \text{amountb} - \text{pools} \times (\text{amounta} + \text{amountb}) = 0;
\end{align*}
\]

The nonlinear constraints in PROC OPTMODEL use the same syntax as linear constraints. In fact, if the variable pools were declared as a parameter, then all the preceding constraints would be linear. The nonlinear constraint in PROC OPTMODEL combines the NLINCON statement of PROC NLP with the assignment in the PROC NLP statements. Objective names can be used in nonlinear constraint expressions to structure the formula as they are in objective expressions.

The PROC OPTMODEL model does not use a RUN statement to invoke the solver. Instead the solver is invoked interactively by the SOLVE statement in PROC OPTMODEL. By default, the OPTMODEL procedure prints much less information about the optimization process. Generally this information consists of messages from the solver (such as the termination reason) and a short status display. The PROC OPTMODEL statements add a PRINT statement in order to display the variable estimates from the solver.

---

**Examples: OPTMODEL Procedure**

**Example 10.1: Matrix Square Root**

This example demonstrates the use of PROC OPTMODEL array parameters and variables. The following statements create a randomized positive definite symmetric matrix and define an optimization model to find the matrix square root of the generated matrix:

```plaintext
proc optmodel;
  number n = 5; /* size of matrix */
  /* random original array */
  number A{1..n, 1..n} = 10 - 20*rand('UNIFORM');
  /* compute upper triangle of the 
  * symmetric matrix A*transpose(A) */
  /* should be positive def unless A is singular */
  number P{i in 1..n, j in i..n};
  for {i in 1..n, j in i..n}
    P[i,j] = sum{k in 1..n} A[i,k]*A[j,k];
  /* coefficients of square root array 
  * (upper triangle of symmetric matrix) */
  var q{i in 1..n, i..n};
  /* The default initial value q[i,j]=0 is */
```

```plaintext
nlincon nlc1-nlc2 >= 0.,
              nlc3 = 0.;

  nlc1 = 2.5 * amountx - pools * pooltox - 2. * ctox;
  nlc2 = 1.5 * amounty - pools * pooltoy - 2. * ctoy;
  nlc3 = 3 * amounta + amountb - pools * (amounta + amountb);
```
* a local minimum of the objective, 
* so you must move it away from that point. */
q[1,1] = 1;
/* minimize difference of square of q from P */
min r = sum{i in 1..n, j in i..n}
( sum{k in 1..i} q[k,i]*q[k,j]
 + sum{k in i+1..j} q[i,k]*q[k,j]
 + sum{k in j+1..n} q[i,k]*q[j,k]
 - P[i,j] )**2;
solve;
print q;

These statements define a random array A of size n \times n. The product P is defined as the matrix product \( AA^T \). The product is symmetric, so the declaration of the parameter P gives it upper triangular indexing. The matrix represented by P should be positive definite unless A is singular. But singularity is unlikely because of the random generation of A. If P is positive definite, then it has a well-defined square root, Q, such that \( P = QQ^T \).

The objective \( r \) simply minimizes the sum of squares of the coefficients as

\[
r = \sum_{1 \leq i \leq j \leq n} R_{i,j}^2
\]

where \( R = QQ^T - P \). (This technique for computing matrix square roots is intended only for the demonstration of PROC OPTMODEL capabilities. Better methods exist.)

Output 10.1.1 shows part of the output from running these statements. The values that are actually displayed depend on the random numbers generated.

Output 10.1.1 Matrix Square Root Results

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-0.10557</td>
<td>-7.03961</td>
<td>8.64638</td>
<td>1.89284</td>
<td>-8.28542</td>
</tr>
<tr>
<td>2</td>
<td>5.23609</td>
<td>0.64662</td>
<td>-6.63339</td>
<td>6.71074</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-1.61894</td>
<td>-7.31866</td>
<td>1.14428</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>3.76627</td>
<td>0.32063</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>4.93412</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example 10.2: Reading From and Creating a Data Set

This example demonstrates how to use the READ DATA statement to read parameters from a SAS data set. The objective is the Bard function, which is the following least squares problem with \( I = \{1, 2, \ldots, 15\} \):

\[
f(x) = \frac{1}{2} \sum_{k \in I} \left[ y_k - \left( x_1 + \frac{k}{v_k x_2 + w_k x_3} \right) \right]^2
\]

\( x = (x_1, x_2, x_3), \quad y = (y_1, y_2, \ldots, y_{15}) \)

where \( v_k = 16 - k, \quad w_k = \min(k, v_k) \quad (k \in I), \) and

\( y = (0.14, 0.18, 0.22, 0.25, 0.29, 0.32, 0.35, 0.39, 0.37, 0.58, 0.73, 0.96, 1.34, 2.10, 4.39) \)
The minimum function value \( f(x^*) = 4.107E-3 \) is at the point \((0.08, 1.13, 2.34)\). The starting point \(x^0 = (1.1, 1, 1)\) is used. This problem is identical to the example “Using the DATA= Option” in Chapter 6, “The NLP Procedure” (SAS/OR User’s Guide: Mathematical Programming Legacy Procedures). The following statements use the READ DATA statement to input parameter values and the CREATE DATA statement to save the solution in a SAS data set:

```sas
data bard;
  input y @@;
datalines;
.14 .18 .22 .25 .29 .32 .35 .39 
.37 .58 .73 .96 1.34 2.10 4.39 
;
proc optmodel;
  set I = 1..15;
  number y{I};
  read data bard into [_n_] y;
  number v{k in I} = 16 - k;
  number w{k in I} = min(k, v[k]);
  var x{1..3} init 1;
  min f = 0.5* 
    sum{k in I}
      (y[k] - (x[1] + k / (v[k]*x[2] + w[k]*x[3])))**2;
  solve;
  print x;
  create data xdata from [i] xd=x;
```

In these statements the values for parameter \(y\) are read from the BARD data set. The set \(I\) indexes the terms of the objective in addition to the \(y\) array.

The preceding statements define two utility parameters that contain coefficients used in the objective function. These coefficients could have been defined in the expression for the objective, \(f\), but it was convenient to give them names and simplify the objective expression.

The result is shown in Output 10.2.1.

**Output 10.2.1** Bard Function Solution

<table>
<thead>
<tr>
<th>[I]</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.08241</td>
</tr>
<tr>
<td>2</td>
<td>1.13303</td>
</tr>
<tr>
<td>3</td>
<td>2.34370</td>
</tr>
</tbody>
</table>

The final CREATE DATA statement saves the solution values determined by the solver into the data set XDATA. The data set contains an observation for each \(x\) index. Each observation contains two variables. The output variable \(i\) contains the index, while \(xd\) contains the value for the indexed entry in the array \(x\). The resulting data can be seen by using the PRINT procedure as follows:

```sas
proc print data=xdata;
run;
```

The output from PROC PRINT is shown in **Output 10.2.2**.
Example 10.3: Model Construction

This example uses PROC OPTMODEL features to simplify the construction of a mathematically formulated model. The model is based on the example “An Assignment Problem” in Chapter 4, “The LP Procedure” (SAS/OR User’s Guide: Mathematical Programming Legacy Procedures). A single invocation of PROC OPTMODEL replaces several steps in the PROC LP statements.

The model assigns production of various grades of cloth to a set of machines in order to maximize profit while meeting customer demand. Each machine has different capacities to produce the various grades of cloth. (See the PROC LP example “An Assignment Problem” for more details.) The mathematical formulation, where $x_{ijk}$ represents the amount of cloth of grade $j$ to produce on machine $k$ for customer $i$, follows:

$$\begin{align*}
\text{max} & \quad \sum_{ijk} r_{ijk}x_{ijk} \\
\text{subject to} & \quad \sum_k x_{ijk} = d_{ij} \quad \text{for all } i \text{ and } j \\
& \quad \sum_i c_{jk}x_{ijk} \leq a_k \quad \text{for all } k \\
& \quad x_{ijk} \geq 0 \quad \text{for all } i, j, \text{ and } k
\end{align*}$$

The OBJECT, DEMAND, and RESOURCE data sets are the same as in the PROC LP example. A new data set, GRADE, is added to help separate the data from the model.

```plaintext
/* Example 10.3: Model Construction */

class proc optmodel;
proc optmodel;
  %define (i, j, k); data grade(drop=i);
  do i = 1 to 6;
    grade = 'grade'||put(i,1.);
    output;
  end;
run;

data object;
  input machine customer grade1 grade2 grade3 grade4 grade5 grade6;
  datalines;
  1 1 102 140 105 105 125 148
  1 2 115 133 118 118 143 166
  1 3 70 108 83 83 88 86
  1 4 79 117 87 87 107 105
  1 5 77 115 90 90 105 148
  2 1 123 150 125 124 154 .
  2 2 130 157 132 131 166 .
  2 3 103 130 115 114 129 .
  2 4 101 128 108 107 137 .
  2 5 118 145 130 129 154 .
run;
```
Example 10.3: Model Construction

The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to a SAS data set called SOLUTION:

```sas
proc optmodel;
   /* declare index sets */
   set CUSTOMERS;
   set <str> GRADES;
   set MACHINES;

   /* declare parameters */
   num return {CUSTOMERS, GRADES, MACHINES} init 0;
   num demand {CUSTOMERS, GRADES};
   num cost {GRADES, MACHINES} init 0;
   num avail {MACHINES};

   /* read the set of grades */
   read data grade into GRADES=[grade];

   /* read the set of customers and their demands */
   read data demand
      into CUSTOMERS=[customer]
```

{j in GRADES} <demand[customer,j]=col(j)>;

/* read the set of machines, costs, and availability */
read data resource nomiss
  into MACHINES=[machine]
  {j in GRADES} <cost[j,machine]=col(j)>
  avail;

/* read objective data */
read data object nomiss
  into [machine customer]
  {j in GRADES} <return[customer,j,machine]=col(j)>;

/* declare the model */
var AmountProduced {CUSTOMERS, GRADES, MACHINES} >= 0;
max TotalReturn = sum {i in CUSTOMERS, j in GRADES, k in MACHINES}
  return[i,j,k] * AmountProduced[i,j,k];
con req_demand {i in CUSTOMERS, j in GRADES}:
  sum {k in MACHINES} AmountProduced[i,j,k] = demand[i,j];
con req_avail {k in MACHINES}:
  sum {i in CUSTOMERS, j in GRADES}
    cost[j,k] * AmountProduced[i,j,k] <= avail[k];

/* call the solver and save the results */
solve;
create data solution
  from [customer grade machine] = {i in CUSTOMERS, j in GRADES,
    k in MACHINES: AmountProduced[i,j,k].sol ne 0}
  amount=AmountProduced;
quit;

The statements use both numeric (NUM) and character (STR) index sets, which are populated from the corresponding data set variables in the READ DATA statements. The OPTMODEL parameters can be either single-dimensional (A V A I L) or multiple-dimensional (C O S T, D E M A N D, R E T U R N). The RETURN and COST parameters are given initial values of 0, and the NOMISS option in the READ DATA statement tells PROC OPTMODEL to read only the nonmissing values from the input data sets. The model declaration is nearly identical to the mathematical formulation. The logical condition $\text{AmountProduced}[i,j,k].\text{sol} \neq 0$ in the CREATE DATA statement ensures that only the nonzero parts of the solution appear in the SOLUTION data set. In the PROC LP example, the creation of this data set required postprocessing of the PROC LP output data set.

The solver produces the following problem summary and solution summary:
The SOLUTION data set can be processed by PROC PRINT as follows to create a compact representation of the solution:

```plaintext
proc sort data=solution;
  by grade;
run;
proc print data=solution;
  id machine customer;
  by grade;
run;
```

These statements produce the table shown in Output 10.3.2.
### Output 10.3.2  An Assignment Problem

#### An Assignment Problem

**grade=grade1**

<table>
<thead>
<tr>
<th>machine</th>
<th>customer</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>300</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>400</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>250</td>
</tr>
</tbody>
</table>

**grade=grade2**

<table>
<thead>
<tr>
<th>machine</th>
<th>customer</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>100.000</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>92.270</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>125.000</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>507.730</td>
</tr>
</tbody>
</table>

**grade=grade3**

<table>
<thead>
<tr>
<th>machine</th>
<th>customer</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>150.000</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>300.000</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>256.721</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>750.000</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>143.279</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>300.000</td>
</tr>
</tbody>
</table>

**grade=grade4**

<table>
<thead>
<tr>
<th>machine</th>
<th>customer</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>150.000</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>210.312</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>275.000</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>289.688</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>750.000</td>
</tr>
</tbody>
</table>

**grade=grade5**

<table>
<thead>
<tr>
<th>machine</th>
<th>customer</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>175</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>340</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>310</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>210</td>
</tr>
</tbody>
</table>

**grade=grade6**

<table>
<thead>
<tr>
<th>machine</th>
<th>customer</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>250</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>325</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>360</td>
</tr>
</tbody>
</table>
Example 10.4: Set Manipulation

This example demonstrates PROC OPTMODEL set manipulation operators. These operators are used to compute the set of primes up to a given limit. This example does not solve an optimization problem, but similar set manipulation could be used to set up an optimization model. Here are the statements:

```plaintext
proc optmodel;
number maxprime; /* largest number to consider */
set composites =
    union {i in 3..sqrt(maxprime) by 2} i*i..maxprime by 2*i;
set primes = {2} union (3..maxprime by 2 diff composites);
maxprime = 500;
put primes;
```

The set `composites` contains the odd composite numbers up to the value of the parameter `maxprime`. The even numbers are excluded here to reduce execution time and memory requirements. The UNION aggregation operation is used in the definition to combine the sets of odd multiples of $i$ for $i = 3, 5, \ldots$. Any composite number less than the value of the parameter `maxprime` has a divisor $\leq \sqrt{\text{maxprime}}$, so the range of $i$ can be limited. The set of multiples of $i$ can also be started at $i \times i$ since smaller multiples are found in the set of multiples for a smaller index.

You can then define the set `primes`. The odd primes are determined by using the DIFF operator to remove the composites from the set of odd numbers no greater than the parameter `maxprime`. The UNION operator adds the single even prime, 2, to the resulting set of primes.

The PUT statement produces the result in Output 10.4.1.

**Output 10.4.1** Primes less than or equal to 500


Note that you were able to delay the definition of the value of the parameter `maxprime` until just before the PUT statement. Since the defining expressions of the SET declarations are handled symbolically, the value of `maxprime` is not necessary until you need the value of the set `primes`. Because the sets `composites` and `primes` are defined symbolically, their values reflect any changes to the parameter `maxprime`. You can see this update by appending the following statements to the preceding statements:

```plaintext
maxprime = 50;
pred primes;
```

The additional statements produce the results in Output 10.4.2. The value of the set `primes` has been recomputed to reflect the change to the parameter `maxprime`.

**Output 10.4.2** Primes less than or equal to 50

{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47}
Example 10.5: Multiple Subproblems

Many important optimization problems cannot be solved directly using a standard solver, either because the problem has constraints that cannot be modeled directly or because the resulting model would be too large to be practical. For these types of problems, you can use PROC OPTMODEL to synthesize solution methods by using a combination of the existing solvers and the modeling language programming constructions. This example demonstrates the use of multiple subproblems to solve the cutting stock problem.

The cutting stock problem determines how to efficiently cut raw stock into finished widths based on the demands for the final product. Consider the example from page 195 of Chvátal (1983), where raw stock has a width of 100 inches and the demands are shown in Table 10.18.

<table>
<thead>
<tr>
<th>Finished Width</th>
<th>Demand</th>
</tr>
</thead>
<tbody>
<tr>
<td>45 inches</td>
<td>97</td>
</tr>
<tr>
<td>35 inches</td>
<td>610</td>
</tr>
<tr>
<td>31 inches</td>
<td>395</td>
</tr>
<tr>
<td>14 inches</td>
<td>211</td>
</tr>
</tbody>
</table>

A portion of the demand can be satisfied using a cutting pattern. For example, with the demands in Table 10.18 a possible pattern cuts one final of width 35 inches, one final of width 31 inches, and two finals of width 14 inches. This gives:

\[
100 = 0 \times 45 + 1 \times 35 + 1 \times 31 + 2 \times 14 + \text{waste.}
\]

The cutting stock problem can be formulated as follows, where \(x_j\) represents the number of times pattern \(j\) appears, \(a_{ij}\) represents the number of times demand item \(i\) appears in pattern \(j\), \(d_i\) is the demand for item \(i\), \(w_i\) is the width of item \(i\), \(N\) represents the set of patterns, \(M\) represents the set of items, and \(W\) is the width of the raw stock:

\[
\begin{align*}
\text{minimize} & \quad \sum_{j \in N} x_j \\
\text{subject to} & \quad \sum_{j \in N} a_{ij} x_j \geq d_i \quad \text{for all } i \in M \\
& \quad x_j \text{ integer, } \geq 0 \quad \text{for all } j \in N
\end{align*}
\]

Also for each feasible pattern \(j\) you must have:

\[
\sum_{i \in M} w_i a_{ij} \leq W
\]

The difficulty with this formulation is that the number of patterns can be very large, with too many columns \(x_j\) to solve efficiently. But you can use column generation, as described on page 198 of Chvátal (1983), to generate a smaller set of useful patterns, starting from an initial feasible set.
The dual variables, \( \pi_i \), of the demand constraints are used to price out the columns. From linear programming (LP) duality theory, a column that improves the primal solution must have a negative reduced cost. For this problem the reduced cost for column \( x_j \) is

\[
1 - \sum_{i \in M} \pi_i a_{ij}
\]

Using this observation produces a knapsack subproblem:

\[
\begin{align*}
\text{minimize} & \quad 1 - \sum_{i \in M} \pi_i a_i \\
\text{subject to} & \quad \sum_{i \in M} w_i a_i \leq W \\
& \quad a_i \text{ integer, } \geq 0 \quad \text{for all } j \in N
\end{align*}
\]

where the objective is equivalent to:

\[
\text{maximize} \quad \sum_{i \in M} \pi_i a_i
\]

The pattern is useful if the associated reduced cost is negative:

\[
1 - \sum_{i \in M} \pi_i a_{ij}^* < 0
\]

So you can use the following steps to generate the patterns and solve the cutting stock problem:

1. Initialize a set of trivial (one item) patterns.
2. Solve the problem using the LP solver.
3. Minimize the reduced cost using a knapsack solver.
4. Include the new pattern if the reduced cost is negative.
5. Repeat steps 2 through 4 until there are no more negative reduced cost patterns.

These steps are implemented in the following statements. Since adding columns preserves primal feasibility, the statements use the primal simplex solver to take advantage of a warm start. The statements also solve the LP relaxation of the problem, but you want the integer solution. So the statements finish by using the MILP solver with the integer restriction applied. The result is not guaranteed to be optimal, but lower and upper bounds can be provided for the optimal objective.

```plaintext
/* cutting-stock problem */
/* uses delayed column generation from
   Chvatal's Linear Programming (1983), page 198 */

%macro csp(capacity);
proc optmodel printlevel=0;
   /* declare parameters and sets */
   num capacity = &capacity;
   set ITEMS;
   num demand {ITEMS};
```
num width {ITEMS};
num num_patterns init card(ITEMS);
set PATTERNS = 1..num_patterns;
num a {ITEMS, PATTERNS};
num c {ITEMS} init 0;
num epsilon = 1E-6;

/* read input data */
read data indata into ITEMS=[_N_] demand width;

/* generate trivial initial columns */
for {i in ITEMS, j in PATTERNS}
a[i,j] = (if (i = j) then floor(capacity/width[i]) else 0);

/* define master problem */
var x {PATTERNS} >= 0 integer;
minimize NumberOfRaws = sum {j in PATTERNS} x[j];
con demand_con {i in ITEMS}:
    sum {j in PATTERNS} a[i,j] * x[j] >= demand[i];
problem Master include x NumberOfRaws demand_con;

/* define column generation subproblem */
var y {ITEMS} >= 0 integer;
maximize KnapsackObjective = sum {i in ITEMS} c[i] * y[i];
con knapsack_con:
    sum {i in ITEMS} width[i] * y[i] <= capacity;
problem Knapsack include y KnapsackObjective knapsack_con;

/* main loop */
do while (1);
    print _page_ a;

    /* master problem */
    /* minimize sum_j x[j]
       subj. to sum_j a[i,j] * x[j] >= demand[i]
               x[j] >= 0 and integer */
    use problem Master;
    put "solve master problem";
    solve with lp relaxint /
        presolver=none solver=ps basis=warmstart printfreq=1;
    print x;
    print demand_con.dual;
    for {i in ITEMS} c[i] = demand_con[i].dual;

    /* knapsack problem */
    /* maximize sum_i c[i] * y[i]
       subj. to sum_i width[i] * y[i] <= capacity
               y[i] >= 0 and integer */
    use problem Knapsack;
    put "solve column generation subproblem";
    solve with milp / printfreq=0;
    for {i in ITEMS} y[i] = round(y[i]);
    print y;
    print KnapsackObjective;
if KnapsackObjective <= 1 + epsilon then leave;

    /* include new pattern */
    num_patterns = num_patterns + 1;
    for {i in ITEMS} a[i,num_patterns] = y[i];
end;

    /* solve IP, using rounded-up LP solution as warm start */
    use problem Master;
    for {j in PATTERNS} x[j] = ceil(x[j].sol);
    put "solve (restricted) master problem as IP";
    solve with milp / primalin;

    /* cleanup solution and save to output data set */
    for {j in PATTERNS} x[j] = round(x[j].sol);
    create data solution from [pattern]={j in PATTERNS: x[j] > 0}
        raws=x {i in ITEMS} <col('i'||i)=a[i,j]>;
quit;
%mend csp;

    /* Chvatal, p.199 */
    data indata;
    input demand width;
    datalines;
78 25.5
40 22.5
30 20
30 15
;
%csp(91)
/* LP solution is integer */

    /* Chvatal, p.195 */
    data indata;
    input demand width;
    datalines;
97 45
610 36
395 31
211 14
;
%csp(100)
/* LP solution is fractional */

The contents of the output data set for the second problem instance are shown in Output 10.5.1.

### Output 10.5.1 Cutting Stock Solution

<table>
<thead>
<tr>
<th>Obs</th>
<th>pattern</th>
<th>raws</th>
<th>i1</th>
<th>i2</th>
<th>i3</th>
<th>i4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td>49</td>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>5</td>
<td>206</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>6</td>
<td>198</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>
Chapter 10: The OPTMODEL Procedure

Example 10.6: Traveling Salesman Problem

This example demonstrates the use of the SUBMIT statement to execute a block of SAS statements from within a PROC OPTMODEL session. In this case, the SUBMIT block calls the SGPLOT procedure to display intermediate results during the solution of an instance of the traveling salesman problem (TSP). The problem is described in Example 14.4. For an example of how to use PROC OPTNETWORK to solve the TSP, see “Traveling Salesman Problem Applied to a Simple Directed Graph” (Chapter 3, SAS Optimization: Network Optimization Procedures).

The following DATA step converts a TSPLIB instance of type EUC_2D into a SAS data set that contains the coordinates of the vertices:

```sas
/* convert the TSPLIB instance into a data set */
data tspData(drop=H);
  infile "&tsplib"
  input H $1. @;
  if H not in ('N','T','C','D','E');
  input @1 var1-var3;
r隼;
run;
```

The following macro generates plots of the solution and objective value:

```sas
%macro plotTSP;
  /* create Annotate data set to draw subtours */
data anno;
  retain drawspace 'datavalue' linethickness 1 function 'line';
  set solData;
r隼;

  title1 h=2 "TSP: Iter = &i, Objective = &&obj&i";
  title2;
  proc sgplot data=tspData sganno=anno;
    scatter x=var2 y=var3 / datalabel=var1;
    xaxis display=none;
    yaxis display=none;
  run;
%mend plotTSP;
```

The following PROC OPTMODEL statements solve the TSP by using the subtour formulation and iteratively adding subtour constraints. The SUBMIT statement calls the `%plotTSP` macro to plot the solution and objective value at each stage.

```sas
/* iterative solution using the subtour formulation */
proc optmodel;
  set VERTICES;
  set EDGES = {i in VERTICES, j in VERTICES: i > j};
  num xc (VERTICES);
  num yc (VERTICES);

  num numsubtour init 0;
  set SUBTOUR {1..numsubtour};
```
Example 10.6: Traveling Salesman Problem

/* read in the instance and customer coordinates (xc, yc) */
read data tspData into VERTICES=[var1] xc=var2 yc=var3;

/* the cost is the euclidean distance rounded to the nearest integer */
num c {<i,j> in EDGES}
  init floor( sqrt( ((xc[i]-xc[j])**2 + (yc[i]-yc[j])**2)) + 0.5);

var x {EDGES} binary;

/* minimize the total cost */
min obj =
  sum {<i,j> in EDGES} c[i,j] * x[i,j];

/* each vertex has exactly one in-edge and one out-edge */
con two_match {i in VERTICES}:
  sum {j in VERTICES: i > j} x[i,j]
  + sum {j in VERTICES: i < j} x[j,i] = 2;

/* no subtours (these constraints are generated dynamically) */
con subtour_elim {s in 1..numsubtour}:
  sum {<i,j> in EDGES: (i in SUBTOUR[s] and j not in SUBTOUR[s])
  or (i not in SUBTOUR[s] and j in SUBTOUR[s])} x[i,j] >= 2;

/* this starts the algorithm to find violated subtours */
set <num,num> EDGES1;
set VERTICES1 = union{<i,j> in EDGES1} {i, j};
um numcomp init 2;
num iter init 1;
call symput('i',trim(left(put(round(iter),best.))));
um numiters init 1;

/* initial solve with just matching constraints */
solve;
call symput(compress('obj'||put(iter,best.)),
  trim(left(put(round(obj),best.))));

/* create a data set for use by PROC SGPLOT */
create solData from
  [i j]={<i,j> in EDGES: x[i,j].sol > 0.5}
x1=xc[i] y1=yc[i] x2=xc[j] y2=yc[j];
submit;
%plotTSP;
endsubmit;

/* while the solution is disconnected, continue */
do while (numcomp > 1);
  iter = iter + 1;
call symput('i',trim(left(put(round(iter),best.))));

/* find connected components of support graph */
EDGES1 = {<i,j> in EDGES: round(x[i,j].sol) = 1};
solve with network /
  links = (include=EDGES1)
Chapter 10: The OPTMODEL Procedure

```
nodes   = (include=VERTICES1)
concomp
out     = (concomp=component);

numcomp = _oroptmodel_num_['NUM_COMPONENTS'];
if numcomp = 1 then leave;

numiters = iter;
numsubtour = numsubtour + numcomp;
for {comp in 1..numcomp} do;
   SUBTOUR[numsubtour-numcomp+comp]
   = {i in VERTICES: component[i] = comp};
end;

solve;
call symput(compress('obj'||put(iter,best.)),
            trim(left(put(round(obj),best.))));

/* create a data set for use by PROC SGPlot */
create data solData from
   [i j]={<i,j> in EDGES: x[i,j].sol > 0.5}
   x1=xc[i] y1=yc[i] x2=xc[j] y2=yc[j];

call symput('numiters',put(numiters,best.));
submit;
   %plotTSP;
endsubmit;
end;
quit;
```

The plot in Output 10.6.1 shows the solution and objective value at each stage. Each stage restricts some subset of subtours. When you reach the final stage, you have a valid tour.

**Output 10.6.1** Iterative Solution of Traveling Salesman Problem
Example 10.6: Traveling Salesman Problem

Output 10.6.1 continued

TSP: Iter = 2, Objective = 652

TSP: Iter = 3, Objective = 673

TSP: Iter = 4, Objective = 675
Example 10.7: Sparse Modeling

This example demonstrates how to rewrite certain models for more efficient processing. Sometimes optimization models that run out of memory during problem generation can be rewritten to take advantage of sparsity to use memory more efficiently. This often occurs when a large array is modeled in a dense format but most of its entries are zeros. Usually, the array provides problem coefficients or it contains optimization variables.

The model for this example solves the facility location problem that is described in Example 14.3. This example is concerned with the resources that are required for PROC OPTMODEL problem generation and solver initialization. So the size of the problem has been increased, but the model has also been modified to make it easier to solve. In order to handle the larger problem size, the model eliminates a large number of the potential assignments of customers to facilities based on distance, making the problem sparse.

The following code generates a random instance of the facility location problem:

```plaintext
%let NumCustomers = 1500;
%let NumSites = 250;
%let SiteCapacity = 50;
%let MaxDemand = 10;
%let xmax = 200;
%let ymax = 100;
%let seed = 938;

/* generate random customer locations */
data cdata(drop=i);
  length name $8;
  call streaminit(&seed);
  do i = 1 to &NumCustomers;
    name = compress('C'||put(i,best.));
    x = rand('UNIFORM') * &xmax;
    y = rand('UNIFORM') * &ymax;
    demand = 1;
    output;
  end;
run;

/* generate random site locations and fixed charge */
data sdata(drop=i);
  length name $8;
  call streaminit(&seed);
  do i = 1 to &NumSites;
    name = compress('SITE'||put(i,best.));
    x = rand('UNIFORM') * &xmax;
    y = rand('UNIFORM') * &ymax;
    fixed_charge = 300 * (abs(&xmax/2-x)/&xmax + abs(&ymax/2-y)/&ymax);
    output;
  end;
run;
```

The following code uses a dense version of the facility location model. This model is equivalent to the model from Example 14.3 except for the added constraint distance_at_most_30. This constraint eliminates from consideration the assignment of customers to facilities over long distances by forcing the corresponding Assign variables to 0.
Example 10.7: Sparse Modeling

```
proc optmodel printlevel=2;
profile on percent=0.1;
set <str> CUSTOMERS;
set <str> SITES init {};

/* x and y coordinates of CUSTOMERS and SITES */
num x {CUSTOMERS union SITES};
num y {CUSTOMERS union SITES};
num demand {CUSTOMERS};
num fixed_charge {SITES};

/* distance from customer i to site j */
num dist {i in CUSTOMERS, j in SITES} = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2);

read data cdata into CUSTOMERS=[name] x y demand;
read data sdata into SITES=[name] x y fixed_charge;

var Assign {CUSTOMERS, SITES} binary;
var Build {SITES} binary;

min CostNoFixedCharge
= sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j];
min CostFixedCharge
= CostNoFixedCharge + sum {j in SITES} fixed_charge[j] * Build[j];

/* each customer assigned to exactly one site */
con assign_def {i in CUSTOMERS}:
    sum {j in SITES} Assign[i,j] = 1;

/* if customer i assigned to site j, then facility must be built at j */
con link {i in CUSTOMERS, j in SITES}:
    Assign[i,j] <= Build[j];

/* each site can handle at most &SiteCapacity demand */
con capacity {j in SITES}:
    sum {i in CUSTOMERS} demand[i] * Assign[i,j] <=
    &SiteCapacity * Build[j];

/* do not assign customer to site more than 30 units away */
con distance_at_most_30 {i in CUSTOMERS, j in SITES: dist[i,j] > 30}:
    Assign[i,j] = 0;

/* solve the MILP */
solve with milp/timetype=real;
quit;
```

If you inspect the log after running the preceding code, then you will see that the MILP presolver has pruned down the problem size considerably. If you also run the code with the SAS option FULLSTIMER enabled on a 64-bit system, then you will notice that about 1.3GB of memory is required for the PROC OPTMODEL step when you are running on a single CPU.
The solution and timing results for the dense model are shown in Output 10.7.1. The PROFILE ON statement requests further timing details, including evaluation time for declarations used by problem generation.

Output 10.7.1 Dense Model Results

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
</tbody>
</table>

| Relative Gap              | 0.0000261906  |
| Absolute Gap              | 0.4714482429  |
| Primal Infeasibility      | 2.839542E-13  |
| Bound Infeasibility       | 2.787291E-13  |
| Integer Infeasibility     | 2.787291E-13  |
| Best Bound                | 18000.668905  |
| Nodes                     | 13            |
| Solutions Found           | 5             |
| Iterations                | 34212         |
| Presolve Time             | 0.87          |
| Solution Time             | 54.75         |

<table>
<thead>
<tr>
<th>Procedure Task Timing</th>
<th>Time (sec.)</th>
<th>% Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Generation</td>
<td>6.00</td>
<td>9.75%</td>
</tr>
<tr>
<td>Solver Initialization</td>
<td>0.58</td>
<td>0.95%</td>
</tr>
<tr>
<td>Code Generation</td>
<td>0.04</td>
<td>0.06%</td>
</tr>
<tr>
<td>Presolve</td>
<td>0.87</td>
<td>1.41%</td>
</tr>
<tr>
<td>Root Node Processing</td>
<td>45.57</td>
<td>74.08%</td>
</tr>
<tr>
<td>Branch And Cut</td>
<td>6.32</td>
<td>10.27%</td>
</tr>
<tr>
<td>Synchronization</td>
<td>0.14</td>
<td>0.23%</td>
</tr>
<tr>
<td>Idle</td>
<td>1.81</td>
<td>2.94%</td>
</tr>
<tr>
<td>Other Tasks</td>
<td>0.04</td>
<td>0.07%</td>
</tr>
<tr>
<td>Solver Postprocessing</td>
<td>0.15</td>
<td>0.24%</td>
</tr>
</tbody>
</table>
The best approach for reducing the memory requirements is to eliminate the Assign variables that are always going to be 0. This is accomplished in the following sparse version of the code. Instead of indexing Assign over the crossproduct of CUSTOMERS and SITES, now the code defines a new set of pairs that satisfy the distance requirement, CUSTOMERS_SITES. This set replaces the constraint distance_at_most_30 in the dense model. The objective and constraints have been modified to use the new indexing scheme, with implicit set slicing (as described in the section “More on Index Sets” on page 406) for constraints assign_def and capacity.

```plaintext
proc optmodel printlevel=2;
    profile on percent=0.1;
    set <str> CUSTOMERS;
    set <str> SITES init {};

    /* x and y coordinates of CUSTOMERS and SITES */
    num x {CUSTOMERS union SITES};
    num y {CUSTOMERS union SITES};
    num demand {CUSTOMERS};
    num fixed_charge {SITES};

    /* distance from customer i to site j */
    num dist {i in CUSTOMERS, j in SITES}
        = sqrt((x[i] - x[j])^2 + (y[i] - y[j])^2);

    read data cdata into CUSTOMERS=[name] x y demand;
    read data sdata into SITES=[name] x y fixed_charge;

    set CUSTOMERS_SITES = {i in CUSTOMERS, j in SITES: dist[i,j] <= 30};
    var Assign {CUSTOMERS_SITES} binary;
    var Build {SITES} binary;

    min CostNoFixedCharge
        = sum {<i,j> in CUSTOMERS_SITES} dist[i,j] * Assign[i,j];
    min CostFixedCharge
        = CostNoFixedCharge + sum {j in SITES} fixed_charge[j] * Build[j];

    /* each customer assigned to exactly one site */
```
con assign_def {i in CUSTOMERS}:
  sum {<(i),j> in CUSTOMERS_SITES} Assign[i,j] = 1;

/* if customer i assigned to site j, then facility must be built at j */
con link {{i,j} in CUSTOMERS_SITES}:
  Assign[i,j] <= Build[j];

/* each site can handle at most &SiteCapacity demand */
con capacity {j in SITES}:
  sum {{i,(j)} in CUSTOMERS_SITES} demand[i] * Assign[i,j] <=
    &SiteCapacity * Build[j];

/* solve the MILP */
solve with milp/timetype=real;
quit;

The log from running the preceding code shows that the MILP presolver does not find anything to improve with this version of the model. On a 64-bit system, the FULLSTIMER option shows that memory requirements have been reduced to about 580MB when you are running on a single CPU, less than half the requirements of the previous model.

The solution and timing results for the dense model are shown in Output 10.7.2. Note that the dense model (Output 10.7.1) and the sparse model (Output 10.7.2) are equivalent after presolver processing and generate the same result using similar amounts of solver time. On the other hand, problem generation time is significantly reduced as are other times including presolve time. Both models used the solver option TIMETYPE=REAL so that all times are reported in seconds of real time. You can see from the “Profile Information” table that the overhead associated with problem declarations has been significantly reduced.

### Output 10.7.2 Sparse Model Results

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Relative Gap</td>
</tr>
<tr>
<td>Absolute Gap</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
</tr>
<tr>
<td>Integer Infeasibility</td>
</tr>
<tr>
<td>Best Bound</td>
</tr>
<tr>
<td>Nodes</td>
</tr>
<tr>
<td>Solutions Found</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>
Example 10.8: Chemical Equilibrium

This example illustrates how to convert PROC NLP code that handles arrays into PROC OPTMODEL form. The following PROC NLP model finds an equilibrium state for a mixture of chemicals. The same model is used in “Example 7.8: Chemical Equilibrium” in Chapter 6, “The NLP Procedure” (SAS/OR User’s Guide: Mathematical Programming Legacy Procedures).

```sas
proc nlp tech=tr pall;
  array x[10] x1-x10;
  min y;
  parms x1-x10 = .1;
  bounds 1.e-6 <= x1-x10;
  lincon 2. = x1 + 2. * x2 + 2. * x3 + x6 + x10,
    1. = x4 + 2. * x5 + x6 + x7,
    1. = x3 + x7 + x8 + 2. * x9 + x10;
  s = x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9 + x10;
  y = 0.;
  do j = 1 to 10;
    y = y + x[j] * (c[j] + log(x[j] / s));
  end;
run;
```
The following statements show a corresponding PROC OPTMODEL model:

```plaintext
proc optmodel;
  set CMP = 1..10;
  number c(CMP) = [-6.089 -17.164 -34.054 -5.914 -24.721
  var x{CMP} init 0.1 >= 1.e-6;
  /* replace the variable s in the PROC NLP model */
  impvar s = sum{i in CMP} x[i];
  min y = sum{j in CMP} x[j] * (c[j] + log(x[j] / s));
  solve;
  print x y;
```

The PROC OPTMODEL model uses the set CMP to represent the set of compounds, which are numbered 1 to 10 in the example. The array c was initialized by using the equivalent PROC OPTMODEL syntax. The individual array locations could also have been initialized by assignment or by READ DATA statements.

The VAR declaration for variable x combines the VAR and BOUNDS statements of the PROC NLP model. The index set of the array is based on the set of compounds CMP to simplify changes to the model.

The linear constraints are similar in form to the PROC NLP model. However, the PROC OPTMODEL version uses the array form of the variable names because it treats arrays as distinct variables, not as aliases of lists of scalar variables.

The implicit variable s replaces the intermediate variable of the same name in the PROC NLP model. This is an example of translating an intermediate variable from the other models to PROC OPTMODEL. An alternative way is to use an additional constraint for every intermediate variable. Instead of declaring objective s as in the preceding statements, you can use the following statements:

```plaintext
... var s;
  con s = sum{i in CMP} x[i];
... 
```

Note that this alternative formulation passes an extra variable and constraint to the solver. This formulation can sometimes be solved more efficiently, depending on the characteristics of the model.

The PROC OPTMODEL version uses a SUM operator over the set CMP, which enhances the flexibility of the model to accommodate possible changes in the set of compounds.

In the PROC NLP model, the objective function y is determined by an explicit loop. The DO loop in PROC NLP is replaced by a SUM aggregation operation in PROC OPTMODEL. The accumulation in the PROC NLP model is now performed by PROC OPTMODEL by using the SUM operator.

This PROC OPTMODEL model can be generalized further. Note that the array initialization and constraints assume a fixed set of compounds. You can rewrite the model to handle an arbitrary number of compounds and chemical elements. The new model loads the linear constraint coefficients from a data set along with the objective coefficients for the parameter c, as follows:
Example 10.8: Chemical Equilibrium

data comp;
   input c a_1 a_2 a_3;
datalines;
-6.089 1 0 0
-17.164 2 0 0
-34.054 2 0 1
-5.914 0 1 0
-24.721 0 2 0
-14.986 1 1 0
-24.100 0 1 1
-10.708 0 0 1
-26.662 0 0 2
-22.179 1 0 1 ;

data atom;
   input b @@;
datalines;
2. 1. 1.
;
proc optmodel;
   set CMP;
   set ELT;
   number c{CMP};
   number a{ELT,CMP};
   number b{ELT};
   read data atom into ELT=[_n_] b;
   read data comp into CMP=[_n_]
      c {i in ELT} < a[i,_n_]=col("a_"||i) >;
   var x{CMP} init 0.1 >= 1.e-6;
   con bal{i in ELT}: b[i] = sum{j in CMP} a[i,j]*x[j];
   impvar s = sum{i in CMP} x[i];
   min y = sum{j in CMP} x[j] * (c[j] + log(x[j] / s));
   print a b;
   solve;
   print x;

This version adds coefficients for the linear constraints to the COMP data set. The data set variable a_n represents the number of atoms in the compound for element n. The READ DATA statement for COMP uses the iterated column syntax to read each of the data set variables a_n into the appropriate location in the array a. In this example the expanded data set variable names are a_1, a_2, and a_3.

The preceding version also adds a new set, ELT, of chemical elements and a numeric parameter, b, that represents the left-hand side of the linear constraints. The data values for the parameters ELT and b are read from the data set ATOM. The model can handle varying sets of chemical elements because of this extra data set and the new parameters.

The linear constraints have been converted to a single, indexed family of constraints. One constraint is applied for each chemical element in the set ELT. The constraint expression uses a simple form that applies generally to linear constraints. The following PRINT statement in the model shows the values that are read from the data sets to define the linear constraints:

   print a b;

The PRINT statements in the model produce the results shown in Output 10.8.1.
In the preceding model, the chemical elements and compounds are designated by numbers. So in the PRINT output, for example, the row that is labeled “3” represents the amount of the compound \( \text{H}_2\text{O} \). PROC OPTMODEL is capable of using more symbolic strings to designate array indices. The following version of the model uses strings to index arrays:

```plaintext
data comp;
  input name $ c a_h a_n a_o;
  datalines;
  H  -6.089  1 0 0
  H2  -17.164 2 0 0
  H2O  -34.054 2 0 1
  N  -5.914  0 1 0
  N2  -24.721 0 2 0
  NH  -14.986 1 1 0
  NO  -24.100 0 1 1
  O  -10.708  0 0 1
  O2  -26.662 0 0 2
  OH  -22.179 1 0 1
;
data atom;
  input name $ b;
  datalines;
  H  2.
  N  1.
  O  1.
;
proc optmodel;
```
set<string> CMP;
set<string> ELT;
number c(CMP);
number a(ELT,CMP);
number b(ELT);
read data atom into ELT=[name] b;
read data comp into CMP=[name]
    c {i in ELT} < a[i,name]=col("a_"||i) >;
var x(CMP) init 0.1 >= 1.e-6;
con bal{i in ELT}: b[i] = sum{j in CMP} a[i,j]*x[j];
impvar s = sum{i in CMP} x[i];
min y = sum{j in CMP} x[j] * (c[j] + log(x[j] / s));
solve;
print x;

In this model, the sets CMP and ELT are now sets of strings. The data sets provide the names of the compounds and elements. The names of the data set variables for atom counts in the data set COMP now include the chemical element symbol as part of their spelling. For example, the atom count for element H (hydrogen) is named a_h. Note that these changes did not require any modification to the specifications of the linear constraints or of the objective.

The PRINT statement in the preceding statements produces the results shown in Output 10.8.2. The indices of variable x are now strings that represent the actual compounds.

**Output 10.8.2** PROC OPTMODEL Output with Strings

<table>
<thead>
<tr>
<th></th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>0.04066848</td>
</tr>
<tr>
<td>H2</td>
<td>0.14773067</td>
</tr>
<tr>
<td>H2O</td>
<td>0.78315260</td>
</tr>
<tr>
<td>N</td>
<td>0.00141459</td>
</tr>
<tr>
<td>N2</td>
<td>0.48524616</td>
</tr>
<tr>
<td>NH</td>
<td>0.00069358</td>
</tr>
<tr>
<td>NO</td>
<td>0.02739955</td>
</tr>
<tr>
<td>O</td>
<td>0.01794757</td>
</tr>
<tr>
<td>O2</td>
<td>0.03731444</td>
</tr>
<tr>
<td>OH</td>
<td>0.09687143</td>
</tr>
</tbody>
</table>
References


# Chapter 11
## The Constraint Programming Solver

### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overview: CLP Solver</td>
<td>442</td>
</tr>
<tr>
<td>Getting Started: CLP Solver</td>
<td>442</td>
</tr>
<tr>
<td>Send More Money</td>
<td>442</td>
</tr>
<tr>
<td>Eight Queens</td>
<td>445</td>
</tr>
<tr>
<td>Syntax: CLP Solver</td>
<td>447</td>
</tr>
<tr>
<td>Functional Summary</td>
<td>447</td>
</tr>
<tr>
<td>SOLVE WITH CLP Statement</td>
<td>447</td>
</tr>
<tr>
<td>General Options</td>
<td>448</td>
</tr>
<tr>
<td>Predicates</td>
<td>450</td>
</tr>
<tr>
<td>Common Syntax Components</td>
<td>450</td>
</tr>
<tr>
<td>ALLDIFF Predicate</td>
<td>451</td>
</tr>
<tr>
<td>ELEMENT Predicate</td>
<td>452</td>
</tr>
<tr>
<td>GCC Predicate</td>
<td>453</td>
</tr>
<tr>
<td>LEXICO Predicate</td>
<td>455</td>
</tr>
<tr>
<td>PACK Predicate</td>
<td>455</td>
</tr>
<tr>
<td>REIFY Predicate</td>
<td>456</td>
</tr>
<tr>
<td>Details: CLP Solver</td>
<td>457</td>
</tr>
<tr>
<td>Types of CSPs</td>
<td>457</td>
</tr>
<tr>
<td>Techniques for Solving CSPs</td>
<td>457</td>
</tr>
<tr>
<td>Differences between PROC OPTMODEL and PROC CLP</td>
<td>459</td>
</tr>
<tr>
<td>Macro Variable <em>OROPTMODEL</em></td>
<td>460</td>
</tr>
<tr>
<td>Examples: CLP Solver</td>
<td>461</td>
</tr>
<tr>
<td>Example 11.1: Logic-Based Puzzles</td>
<td>461</td>
</tr>
<tr>
<td>Example 11.2: Alphabet Blocks Problem</td>
<td>468</td>
</tr>
<tr>
<td>Example 11.3: Work-Shift Scheduling Problem</td>
<td>471</td>
</tr>
<tr>
<td>Example 11.4: A Nonlinear Optimization Problem</td>
<td>475</td>
</tr>
<tr>
<td>Example 11.5: Car Painting Problem</td>
<td>478</td>
</tr>
<tr>
<td>Example 11.6: Scene Allocation Problem</td>
<td>480</td>
</tr>
<tr>
<td>Example 11.7: Car Sequencing Problem</td>
<td>484</td>
</tr>
<tr>
<td>Example 11.8: Balanced Incomplete Block Design</td>
<td>488</td>
</tr>
<tr>
<td>Example 11.9: Progressive Party Problem</td>
<td>493</td>
</tr>
<tr>
<td>References</td>
<td>499</td>
</tr>
</tbody>
</table>
Overview: CLP Solver

You can use the constraint logic programming (CLP) solver in the OPTMODEL procedure to address finite-domain constraint satisfaction problems (CSPs) that have linear, logical, and global constraints. In addition to providing an expressive syntax for representing CSPs, the CLP solver features powerful built-in consistency routines and constraint propagation algorithms, a choice of nondeterministic search strategies, and controls for guiding the search mechanism. These features enable you to solve a diverse array of combinatorial problems.

Many important problems in areas such as artificial intelligence (AI) and operations research (OR) can be formulated as constraint satisfaction problems. A CSP is defined by a finite set of variables that take values from finite domains and by a finite set of constraints that restrict the values that the variables can simultaneously take.

A solution to a CSP is an assignment of values to the variables in order to satisfy all the constraints. The problem amounts to finding one or more solutions, or possibly determining that a solution does not exist.

A constraint satisfaction problem (CSP) can be defined as a triplet \( \langle X, D, C \rangle \):

- \( X = \{x_1, \ldots, x_n\} \) is a finite set of variables.
- \( D = \{D_1, \ldots, D_n\} \) is a finite set of domains, where \( D_i \) is a finite set of possible values that the variable \( x_i \) can take. \( D_i \) is known as the domain of variable \( x_i \).
- \( C = \{c_1, \ldots, c_m\} \) is a finite set of constraints that restrict the values that the variables can simultaneously take.

The domains do not need to represent consecutive integers. For example, the domain of a variable could be the set of all even numbers in the interval \([0, 100]\). In PROC OPTMODEL, variables are always numeric. Therefore, if your problem contains nonnumeric domains (such as colors), you must map the values in the domain to integers. For more information, see “Example 11.5: Car Painting Problem” on page 478.

You can use the CLP solver to find one or more (and in some instances, all) solutions to a CSP that has linear, logical, and global constraints. The numeric components of all variable domains are required to be integers.

Getting Started: CLP Solver

The following examples illustrate the use of the CLP solver in formulating and solving two well-known logical puzzles in constraint programming.

Send More Money

The Send More Money problem consists of finding unique digits for the letters D, E, M, N, O, R, S, and Y such that S and M are different from zero (no leading zeros) and the following equation is satisfied:
You can use the CLP solver to formulate this problem as a CSP by using an integer variable to represent each letter in the expression. The comments before each statement in the following code introduce variables, domains, and constraints:

```plaintext
/* Send More Money */
proc optmodel;
    /* Declare all variables as integer. */
    var S integer, E integer, N integer, D integer, M integer, O integer,
        R integer, Y integer;
    /* Set all domains to be between 0 and 9. Domains are unbounded by default.
     * Always declare domains to be as tight as possible. */
    for {j in 1.._NVAR_} do;
        _VAR_[j].lb = 0;
        _VAR_[j].ub = 9;
    end;
    /* Describe the arithmetic constraint. */
    con Arithmetic: 1000*S + 100*E + 10*N + D
                    + 1000*M + 100*O + 10*R + E
                    = 10000*M + 1000*O + 100*N + 10*E + Y;
    /* Forbid leading letters from taking the value zero.
     * Constraint names are optional. */
    con S ne 0;
    con M ne 0;
    /* Require all variables to take distinct values. */
    con alldiff(S E N D M O R Y);
    solve;
    print S E N D M O R Y;
quit;
```

The domain of each variable is the set of digits 0 through 9. The VAR statement identifies the variables in the problem. The Arithmetic constraint defines the linear constraint \( SEND + MORE = MONEY \) and the restrictions that \( S \) and \( M \) cannot take the value 0. (Alternatively, you can simply specify the domain for \( S \) and \( M \) as \( 1 \leq \text{domain} \leq 9 \) in the VAR statement.) Finally, the ALLDIFF predicate enforces the condition that the assignment of digits should be unique.

The PRINT statement produces the solution output as shown in Figure 11.1.
Figure 11.1 Solution to SEND + MORE = MONEY

The OPTMODE Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
<td>Minimization</td>
</tr>
<tr>
<td>Objective Function</td>
<td>(0)</td>
</tr>
<tr>
<td>Objective Type</td>
<td>Constant</td>
</tr>
<tr>
<td>Number of Variables</td>
<td>8</td>
</tr>
<tr>
<td>Bounded Above</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
<td>8</td>
</tr>
<tr>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>Fixed</td>
<td>0</td>
</tr>
<tr>
<td>Binary</td>
<td>0</td>
</tr>
<tr>
<td>Integer</td>
<td>8</td>
</tr>
<tr>
<td>Number of Constraints</td>
<td>4</td>
</tr>
<tr>
<td>Linear LE (&lt;=)</td>
<td>0</td>
</tr>
<tr>
<td>Linear EQ (=)</td>
<td>1</td>
</tr>
<tr>
<td>Linear GE (&gt;=)</td>
<td>0</td>
</tr>
<tr>
<td>Linear LT (&lt;)</td>
<td>0</td>
</tr>
<tr>
<td>Linear NE (&lt;=)</td>
<td>2</td>
</tr>
<tr>
<td>Linear GT (&gt;)</td>
<td>0</td>
</tr>
<tr>
<td>Linear Range</td>
<td>0</td>
</tr>
<tr>
<td>Alldiff</td>
<td>1</td>
</tr>
<tr>
<td>Element</td>
<td>0</td>
</tr>
<tr>
<td>GCC</td>
<td>0</td>
</tr>
<tr>
<td>Lexico (&lt;=)</td>
<td>0</td>
</tr>
<tr>
<td>Lexico (&lt;)</td>
<td>0</td>
</tr>
<tr>
<td>Pack</td>
<td>0</td>
</tr>
<tr>
<td>Reify</td>
<td>0</td>
</tr>
<tr>
<td>Constraint Coefficients</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>CLP</td>
</tr>
<tr>
<td>Variable Selection</td>
<td>MINR</td>
</tr>
<tr>
<td>Objective Function</td>
<td>(0)</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Solution Limit Reached</td>
</tr>
<tr>
<td>Objective Value</td>
<td>0</td>
</tr>
<tr>
<td>Solutions Found</td>
<td>1</td>
</tr>
<tr>
<td>Presolve Time</td>
<td>0.00</td>
</tr>
<tr>
<td>Solution Time</td>
<td>0.03</td>
</tr>
</tbody>
</table>

SEND MORY
9 5 6 7 1 0 8 2
The CLP solver determines the following unique solution to this problem:

\[
\begin{align*}
9 & 5 & 6 & 7 \\
+ & 1 & 0 & 8 & 5 \\
\hline \\
1 & 0 & 6 & 5 & 2
\end{align*}
\]

---

**Eight Queens**

The Eight Queens problem is a special instance of the $N$-Queens problem, where the objective is to position $N$ queens on an $N \times N$ chessboard such that no two queens can attack each other. The CLP solver provides an expressive constraint for variable arrays that can be used for solving this problem very efficiently.

You can model this problem by using a variable array $a$ of dimension $N$, where $a_i$ is the row number of the queen in column $i$. Because no two queens can be in the same row, it follows that all the $a_i$'s must be pairwise distinct.

In order to ensure that no two queens can be on the same diagonal, the following two expressions should be true for all $i$ and $j$:

\[
\begin{align*}
& a_j - a_i <> j - i \\
& a_j - a_i <> i - j
\end{align*}
\]

These expressions can be reformulated as follows:

\[
\begin{align*}
& a_i - i <> a_j - j \\
& a_i + i <> a_j + j
\end{align*}
\]

Hence, the $(a_i + i)$'s are pairwise distinct, and the $(a_i - i)$'s are pairwise distinct. One possible formulation is as follows:

```plaintext
proc optmodel;
  num n init 8;
  var A {1..n} >= 1 <= n integer;
  /* Define artificial offset variables. */
  var B {1..n, -1..1} >= 1 - n <= n + n integer;
  con Bdef {i in 1..n, k in -1..1}:
    B[i,k] = A[i] + k * i;

  con OffsetsMustBeAllDifferent {k in -1..1}:
    alldiff({i in 1..n} B[i,k]);
  solve with CLP / varselect=fifo;
  /* Replicate typical PROC CLP output from an OPTMODEL array */
  create data out from {i in 1..n}<col('A'||i)=A[i]>;
quit;
```
The `VARSELECT=` option specifies the variable selection strategy to be first-in, first-out—the order in which the CLP solver encounters the variables.

The corresponding solution to the Eight Queens problem is displayed in Figure 11.2.

Figure 11.2  A Solution to the Eight Queens Problem
Syntax: CLP Solver

SOLVE WITH CLP /
<FINDALLSOLNS>
<MAXSOLNS=number>
<MAXTIME=number>
<NOPREPROCESS>
<OBJTOL=number>
<prePROCESS>
<SHOWPROGRESS>
<TIMETYPE=number | string>
<VARASSIGN=string>
<VARSELECT=string>
;

The section “Functional Summary” on page 447 provides a quick reference for each option. Each option is then described in more detail in its own section, in alphabetical order.

Functional Summary

Table 11.1 summarizes the options available in the SOLVE WITH CLP statement.

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>General Options</td>
<td></td>
</tr>
<tr>
<td>Finds all possible solutions</td>
<td>FINDALLSOLNS</td>
</tr>
<tr>
<td>Specifies the number of solution attempts</td>
<td>MAXSOLNS=</td>
</tr>
<tr>
<td>Specifies the maximum time to spend calculating results</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Suppresses preprocessing</td>
<td>NOPREPROCESS</td>
</tr>
<tr>
<td>Specifies the tolerance of the objective value</td>
<td>OBJTOL=</td>
</tr>
<tr>
<td>Permits preprocessing</td>
<td>PREPROCESS</td>
</tr>
<tr>
<td>Indicates progress in the log</td>
<td>SHOWPROGRESS</td>
</tr>
<tr>
<td>Specifies whether time units are CPU time or real time</td>
<td>TIMETYPE=</td>
</tr>
<tr>
<td>Specifies the variable assignment strategy</td>
<td>VARASSIGN=</td>
</tr>
<tr>
<td>Specifies the variable selection strategy</td>
<td>VARSELECT=</td>
</tr>
</tbody>
</table>

SOLVE WITH CLP Statement

SOLVE WITH CLP / < options > ;

The SOLVE WITH CLP statement invokes the CLP solver. You can specify the following options to define various processing and diagnostic controls and to tune the algorithm to run.
General Options

You can specify the following general options.

**FINDALLSOLNS**

**ALLSOLNS**

**FINDALL**

attempts to find all possible solutions to the CSP.

**MAXSOLNS=number**

specifies the number of solution attempts to be generated for the CSP. By default, MAXSOLNS=1.

**MAXTIME=number**

specifies the maximum time to spend calculating results. The type of time (either CPU time or real time) is determined by the value of the TIMETYPE= option. The value of number can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

**NOPREPROCESS**

suppresses any preprocessing that would usually be performed for the problem.

**OBJTOL=number**

specifies the tolerance of the objective value. By default, OBJTOL=1E–6.

**PREPROCESS**

permits any preprocessing that would usually be performed for the problem.

**SHOWPROGRESS**

prints a message to the log whenever a solution is found.

**TIMETYPE=number | string**

specifies whether to use CPU time or real time for the MAXTIME= option. Table 11.2 describes the valid values of the TIMETYPE= option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CPU</td>
<td>Specifies units of CPU time</td>
</tr>
<tr>
<td>1</td>
<td>REAL</td>
<td>Specifies units of real time</td>
</tr>
</tbody>
</table>

By default, TIMETYPE=REAL.

**VARASSIGN=string**

specifies the variable assignment strategy. You can specify two value selection strategies:

- **MAX**, which selects the maximum value from the domain of the selected variable
- **MIN**, which selects the minimum value from the domain of the selected variable

By default, VARASSIGN=MIN.
**VARSELECT=string**

specifies the variable selection strategy. The strategy could be static, dynamic, or conflict-directed. Typically, static strategies exploit information about the initial state of the search, whereas dynamic strategies exploit information about the current state of the search process. Conflict-directed strategies exploit information from previous states of the search process as well as the current state (Boussemart et al. 2004). Table 11.3 describes the valid values of the VARSELECT= option.

<table>
<thead>
<tr>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Static strategies</strong></td>
<td></td>
</tr>
<tr>
<td>FIFO</td>
<td>Uses the first-in, first-out ordering of the variables as encountered by the procedure after adjusting for the values in the .priority suffix</td>
</tr>
<tr>
<td>MAXCS</td>
<td>Selects the variable that has the maximum number of constraints</td>
</tr>
<tr>
<td><strong>Dynamic strategies</strong></td>
<td></td>
</tr>
<tr>
<td>DOMDEG</td>
<td>Selects the variable that has the smallest ratio of domain size to degree</td>
</tr>
<tr>
<td>DOMDEG</td>
<td>Selects the variable that has the smallest ratio of domain size to degree</td>
</tr>
<tr>
<td>MAXC</td>
<td>Selects the variable that has the largest number of active constraints</td>
</tr>
<tr>
<td>MINR</td>
<td>Selects the variable that has the smallest range (that is, the minimum value of the upper bound minus the lower bound)</td>
</tr>
<tr>
<td>MINRMAXC</td>
<td>Selects the variable that has the smallest range, breaking ties by selecting one that has the largest number of active constraints</td>
</tr>
<tr>
<td><strong>Conflict-directed strategies</strong></td>
<td></td>
</tr>
<tr>
<td>DOMWDEG</td>
<td>Selects the variable that has the smallest ratio of domain size to weighted degree</td>
</tr>
<tr>
<td>WDEG</td>
<td>Selects the variable that has the largest weighted degree</td>
</tr>
</tbody>
</table>

The dynamic strategies embody the “fail-first principle” (FFP) of Haralick and Elliott (1980), which suggests, “To succeed, try first where you are most likely to fail.” By default, VARSELECT=MINR.

You can change the order in which the CLP solver selects variables after you have declared them by setting the .priority suffix for your variables. Variables with higher priority are selected first.

PROC OPTMODEL usually honors the variable declaration order when it invokes the CLP solver. However, when variables are indexed over dynamically computed sets, the variable order might be undefined. The value of .priority always overrides declaration order. To ensure that the CLP solver considers variables in the exact order that you want, set the .priority suffix for all variables. You can also experiment with the priorities by changing the suffixes without having to reorder the variable declarations in your code.

For more information, see the section “Integer Variable Suffixes” on page 384 in Chapter 10, “The OPTMODEL Procedure.”
Predicates

A predicate asserts a fact about its subject. You use a predicate as the first identifier in a constraint declaration to define what must hold true in any feasible solution. The following is an example of a constraint that uses the ELEMENT predicate:

```plaintext
var X integer, Y integer;
con ElementConstraintExample: element(X, 1 2 3 4, Y);
```

The following predicates are available in PROC OPTMODEL:

- `ALLDIFF(variable-list)`
- `ELEMENT(scalar-variable, data-list, scalar-variable)`
- `GCC(variable-list, set-of-numeric-triplets)`
- `LEXICO(variable-list, order-type, variable-list)`
- `PACK(variable-list, data-list, variable-list)`
- `REIFY(scalar-variable, linear-constraint)`

Common Syntax Components

The following syntax components are used in multiple predicates. They depend on the definition of an `identifier-expression`. For more information, see the section “Identifier Expressions” on page 347 in Chapter 10, “The OPTMODEL Procedure.” For information about other syntactic components, see the section that describes the corresponding predicate.

- `data-list` is a space-separated list of items, each of which can be prefixed by an indexing set and conforms to one of the following syntaxes:
  - a number
  - an `identifier-expression`, but excluding suffixes
  - the name of a numeric array
  - `( expression )`, which must evaluate to a number

For example, the first three constraints in the following statements refer to valid `data-lists`, whereas the last four do not. Each incorrect constraint is preceded by a comment that explains why the constraint is incorrect.

```plaintext
var X integer, Y integer, Z integer;
num n;
con CorrectDataList1: element(X, 1 2 3 4, Y);
con CorrectDataList2: element(X, 1 2 3 4 n, Y);
con CorrectDataList3: element(X, {i in 1..4} i n, Y);
/* The parentheses imply a scalar expression */
con IncorrectDataList1: element(X, (i in 1..4) i n, Y);
/* [1 2 3 4] is an array initializer, not an array name */
con IncorrectDataList2: element(X, [1 2 3 4], Y);
/* Z is a variable */
con IncorrectDataList3: element(X, 1 2 3 4 Z, Y);
/* Impossible to distinguish Z.sol from [ Z . sol ] */
con IncorrectDataList4: element(X, 1 2 3 4 Z.sol, Y);
```
Predicates

Scalar-variable

is an identifier-expression that refers to a single variable (that is, not to an array of variables).

Variable-list

is a space-separated list of items, like a data-list, except that the items can resolve to variables, variable arrays, linear expressions, and constant numeric values. The items, each of which can be prefixed by an indexing set, conform to one of the following forms of syntax:

- a number
- an identifier-expression (but excluding suffixes) that specifies a variable, implicit variable, objective, or numeric parameter
- the name of a variable, implicit variable, objective, or numeric array
- (expression), which must evaluate to a linear equation or numeric constant

When the value of an element of a variable-list refers to a linear equation or numeric constant, it is handled like a solver variable equal to the value.

As an example, the first four constraints in the following statements refer to valid variable-lists, whereas the last two do not. Each incorrect constraint is preceded by a comment that explains why the constraint is incorrect.

```plaintext
var X{1..3} integer, A integer, B integer;
impvar IX{i in 1..3} = X[i] + i;
num n init 2;
con CorrectVariableList1: alldiff({j in 1..3} X[j]);
con CorrectVariableList2: alldiff(A B);
con CorrectVariableList3: alldiff({j in 1..3} X[j] A B);
con CorrectVariableList4: alldiff(1 n (n+1) IX);
/* Indexing is not distributive in variable lists */
var Y{1..3} integer;
con IncorrectVariableList1: alldiff({j in 1..3} (X[j] Y[j]));
/* Expressions and suffixed ids must appear in parentheses */
con IncorrectVariableList2: alldiff(A.dual);
```

Alldiff Predicate

**Alldiff**(variable-list)

**Alldifferent**(variable-list)

The Alldiff predicate defines an all-different constraint, which defines a unique global constraint on a set of variables that requires all of them to be different from each other. A global constraint is equivalent to a conjunction of elementary constraints.

The syntax of the all-different constraint consists of one part, a variable-list, which is defined in the section “Common Syntax Components” on page 450. For example, the statements
Chapter 11: The Constraint Programming Solver

```plaintext
var X[1..3] integer, A integer, B integer;
con AD1: alldiff({j in 1..3} X[j]);
con AD2: alldiff(A, B);
```

are equivalent to

\[
\begin{align*}
X[1] & \neq X[2] \land \\
X[1] & \neq X[3] \land \\
A & \neq B
\end{align*}
\]

To apply the all-different constraint to all the variables, use the problem symbol `_VAR_` as follows:

```plaintext
con alldiff(_VAR_);
```

For a description of problem symbols, see the section “Problem Symbols” on page 399 in Chapter 10, “The OPTMODEL Procedure.”

## ELEMENT Predicate

**ELEMENT**(scalar-variable, data-list, scalar-variable)

The ELEMENT predicate specifies an array element lookup constraint, which enables you to define dependencies (which are not necessarily functional) between variables.

The predicate ELEMENT\((I, L, V)\) sets the variable \(V\) to be equal to the \(I\)th element in the list \(L\), where \(L = (v_1, \ldots, v_n)\) is a list of values (not necessarily distinct) that the variable \(V\) can take. The variable \(I\) is the index variable, and its domain is considered to be \([1, n]\). Each time the domain of \(I\) is modified, the domain of \(V\) is updated, and vice versa.

For example, the following statements use the ELEMENT predicate to determine whether there are squares greater than 1 that are also elements of the Fibonacci sequence:

```plaintext
/* Are there any squares > 1 in the Fibonacci sequence? */
proc optmodel;
  num n = 20;
  /* 1 appears twice in the Fibonacci sequence */
  num fib{i in 1..n} = if i < 3 then 1 else fib[i-1] + fib[i-2];

  var IFib integer, ISq integer,
      XFib integer, XSq integer;
  con IsFibAndIsSquare: XFib = XSq;
  /* You can use a numeric array to refer to a list */
  con IdxOfFib: element( IFib, fib, XFib );
  /* You can also build a list from a set iterator */
  con IdxOfSq: element( ISq, {i in 2..n} (i * i), XSq );
  solve;
  print XFib XSq;
quit;
```

An element constraint enforces the propagation rules

\[ V = v \iff I \in \{i_1, \ldots, i_m\} \]
where \( v \) is a value in the list \( L \) and \( i_1, \ldots, i_m \) are all the indices in \( L \) whose value is \( v \).

An element constraint is equivalent to a conjunction of reify and linear constraints. For example, both of the following examples implement the quadratic function, \( Y = X^2 \):

- Using the ELEMENT predicate:

```plaintext
proc optmodel;
  var X >= 1 <= 5 integer, Y >= 1 <= 25 integer;
  num a {i in 1..5} = i^2;
  con Mycon: element(X, a, Y);
  solve;
quit;
```

- Using linear constraints and the REIFY predicate:

```plaintext
proc optmodel;
  var X >= 1 <= 5 integer, Y >= 1 <= 25 integer, R {1..5} binary;
  con MyconX {i in 1..5}:
    reify(R[i], X = i);
  con MyconY {i in 1..5}:
    reify(R[i], Y = i^2);
  con SumToOne:
    sum {i in 1..5} R[i] = 1;
  solve;
quit;
```

You can also use element constraints to define positional mappings between two variables. For example, suppose the function \( Y = X^2 \) is defined on only odd numbers in the interval \([-5, 5]\). You can relate \( X \) and \( Y \) by using two element constraints and an artificial index variable:

```plaintext
var I integer, X integer, Y integer;
/* You can also build a list by providing explicit literals. */
con XsubI: element (I, -5 -3 -1 1 3 5, X);
con YsubI: element (I, 25 9 1 1 9 25, Y);
```

**GCC Predicate**

\[
\text{GCC}(\text{variable-list}, \text{set-of-numeric-triplets})
\]

The GCC predicate specifies a global cardinality constraint (GCC), which sets the minimum and maximum number of times each value can be assigned to a group of variables.

The syntax of the GCC constraint consists of two parts:

- \textit{variable-list}
- \textit{set-of-numeric-triplets}

See the section “Common Syntax Components” on page 450.
set-of-numeric-triplets

The triplets \(<v, l_v, u_v>\) provide, for each value \(v\), the minimum \(l_v\) and maximum \(u_v\) number of times that \(v\) can be assigned to the variables in the variable-list. The PROC OPTMODEL option INTFUZZ= determines which values are rounded to integers.

Consider the following statements:

```plaintext
var X {1..6} >= 1 <= 4 integer;
con Mycon: gcc(X, /<1,1,2>, <2,1,3>, <3,1,3>, <4,2,3>/);
```

These statements specify a constraint that expresses the following requirements about the values of variables \(\{X[1], \ldots, X[6]\}\):

- The value 1 must appear at least once but no more than twice.
- The value 2 must appear at least once but no more than three times.
- The value 3 must appear at least once but no more than three times.
- The value 4 must appear at least twice but no more than three times.


In general, a GCC constraint consists of a set of variables \(\{x_1, \ldots, x_n\}\) and, for each value \(v\) in \(D = \bigcup_{i=1}^{n} \text{Dom}(x_i)\), a pair of numbers \(l_v\) and \(u_v\). A GCC is satisfied if and only if the number of times that a value \(v\) in \(D\) is assigned to the variables \(x_1, \ldots, x_n\) is at least \(l_v\) and at most \(u_v\).

Values in the domain of variable-list that do not appear in any triplet are unconstrained. They can be assigned to as many of the variables in variable-list as needed to produce a feasible solution.

The following statements specify that each of the values in the set \(\{1, \ldots, 9\}\) can be assigned to at most one of the variables \(X[1], \ldots, X[9]\):

```plaintext
var X {1..9} >= 1 <= 9 integer;
con Mycon: gcc(X, setof{i in 1..9} <i,0,1>);
```

Note that the preceding global cardinality constraint is equivalent to the all-different constraint that is expressed as follows:

```plaintext
var X {1..9} >= 1 <= 9 integer;
con Mycon: alldiff(X);
```

The global cardinality constraint also provides a convenient way to define disjoint domains for a set of variables. For example, the following syntax limits assignment of the variables \(X[1], \ldots, X[9]\) to even numbers between 0 and 10:

```plaintext
var X {1..9} >= 0 <= 10 integer;
con Mycon: gcc(X, setof{i in 1..9 by 2} <i,0,0>);
```
LEXICO Predicate

**LEXICO**(variable-list order-type variable-list)

The LEXICO predicate defines a lexicographic ordering constraint, which compares two arrays of the same size from left to right. For example, a standings table in a sports competition is usually ordered lexicographically, with certain attributes (such as wins or points) to the left of others (such as goal difference).

The *order-type* can be either \( \leq \), to indicate lexicographically less than or equal to \((\leq_{\text{lex}})\), or \(<\), to indicate lexicographically less than \((<_{\text{lex}})\). Given two \(n\)-tuples \(x = (x_1, \ldots, x_n)\) and \(y = (y_1, \ldots, y_n)\), the \(n\)-tuple \(x\) is lexicographically less than or equal to \(y\) \((x \leq_{\text{lex}} y)\) if and only if

\[
(x_i = y_i \ \forall \ i = 1, \ldots, n) \lor (\exists j \text{ with } 1 \leq j \leq n \text{ s.t. } x_i = y_i \ \forall \ i = 1, \ldots, j-1 \text{ and } x_j < y_j)
\]

The \(n\)-tuple \(x\) is lexicographically less than \(y\) \((x <_{\text{lex}} y)\) if and only if \(x \leq_{\text{lex}} y\) and \(x \neq y\). Equivalently, \(x <_{\text{lex}} y\) if and only if

\[
\exists j \text{ with } 1 \leq j \leq n \text{ s.t. } x_i = y_i \ \forall \ i = 1, \ldots, j-1 \text{ and } x_j < y_j
\]

Informally, you can think of the lexicographic constraint \(\leq_{\text{lex}}\) as sorting the \(n\)-tuples in alphabetical order. Mathematically, \(\leq_{\text{lex}}\) is a partial order on a subset of \(n\)-tuples, and \(<_{\text{lex}}\) is a strict partial order on a subset of \(n\)-tuples (Brualdi 2010).

For example, you can express the lexicographic constraint \((X[1], \ldots, X[6]) \leq_{\text{lex}} (Y[1], \ldots, Y[6])\) by using a LEXICO predicate as follows:

```
con Mycon: lexicoo({j in 1..6} X[j] <= {j in 1..6} Y[j]);
```


Lexicographic ordering constraints can be useful for breaking a certain type of symmetry that arises in CSPs and involves matrices of decision variables. Frisch et al. (2002) introduce an optimal algorithm to establish generalized arc consistency (GAC) for the \(\leq_{\text{lex}}\) constraint between two vectors of variables.

PACK Predicate

**PACK**(variable-list, data-list, variable-list)

The PACK predicate specifies a pack constraint, which is used to assign items to bins, subject to the sizes of the items and the capacities of the bins.

For example, suppose you have three bins, whose capacities are 3, 4, and 5, and you have five items of sizes 4, 3, 2, 2, and 1, to be assigned to these three bins. The following statements formulate the problem and find a solution:

```
proc optmodel;
    var SpaceUsed {bin in 1..3} integer >= 0 <= bin + 2;
    var WhichBin {1..5} >= 1 <= 3 integer;
    num itemSize {1..5} = [4 3 2 2 1];
    con pack( WhichBin, itemSize, SpaceUsed );
    solve with clp / findall;
quit;
```

Each row of Table 11.4 represents a solution to the problem. The number in each item column is the number of the bin to which the corresponding item is assigned.
Table 11.4  Bin Packing Solutions

<table>
<thead>
<tr>
<th>WhichBin Variable</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>WhichBin[1]</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>WhichBin[2]</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>WhichBin[3]</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>WhichBin[4]</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>WhichBin[5]</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

When you assign a set of \( k \) items to \( m \) bins, the item variable \( b_i, i \in \{1, \ldots, k\} \) (WhichBin in the preceding example), contains the bin number for the \( i \)th item. The constant \( s_i \) (itemSize in the preceding example) holds the size or weight of the \( i \)th item. The domain of the load variable \( l_j \) (SpaceUsed in the preceding example) constrains the capacity of bin \( j \). The value of \( l_j \) in the solution is the amount of space used in bin \( j \).

**Note:** It can be more efficient to assign higher priority to item variables than to load variables, and within the item variables, to assign higher priority to larger items.

**REIFY Predicate**

**REIFY**(scalar-variable, linear-constraint)

The REIFY predicate specifies a reify constraint, which associates a binary variable with a constraint. The value of the binary variable is 1 or 0, depending on whether the constraint is satisfied or not, respectively. The constraint is said to be reified, and the binary variable is referred to as the control variable. Currently, only linear constraints can be reified.

The REIFY predicate provides a convenient mechanism for expressing logical constraints, such as disjunctive and implicative constraints. For example, the disjunctive constraint

\[
(3X + 4Y < 20) \lor (5X - 2Y > 50)
\]

can be expressed by the following statements:

```plaintext
var X integer, Y integer,
P binary,  Q binary;
con reify(P, 3 * X + 4 * Y < 20);
con reify(Q, 5 * X - 2 * Y > 50);
con AtLeastOneHolds: P + Q >= 1;
```

The binary variable \( P \) reifies the linear constraint \( 3X + 4Y < 20 \)

The binary variable \( Q \) reifies the linear constraint \( 5X - 2Y > 50 \)

The following linear constraint enforces the desired disjunction:

\[
P + Q \geq 1
\]

The following implicative constraint

\[
(3X + 4Y < 20) \Rightarrow (5X - 2Y > 50)
\]
can be enforced by the linear constraint

\[ P \leq Q \]

You can also use the REIFY constraint to express a constraint that involves the absolute value of a variable. For example, the constraint

\[ |X| = 5 \]

can be expressed by the following statements:

```plaintext
var X integer, P binary, Q binary;
con Xis5: reify(P, X = 5);
con XisMinus5: reify(Q, X = -5);
con OneMustHold: P + Q = 1;
```

---

**Details: CLP Solver**

**Types of CSPs**

The CLP solver is a finite-domain constraint programming solver for CSPs. A standard CSP is characterized by integer variables, linear constraints, global constraints, and reify constraints. The solver expects only linear, ALLDIFF, ELEMENT, GCC, LEXICO, PACK, and REIFY predicates.

Both PROC OPTMODEL and PROC CLP support standard CSPs. The CLP procedure also supports scheduling CSPs, which are characterized by activities, temporal constraints, and resource requirement constraints. For more information about the CLP procedure see *SAS/OR User’s Guide: Constraint Programming*.

**Techniques for Solving CSPs**

Several techniques for solving CSPs are available. Kumar (1992) and Tsang (1993) present a good overview of these techniques. It should be noted that the satisfiability problem (SAT) (Garey and Johnson 1979) can be regarded as a CSP. Consequently, most problems in this class are nondeterministic polynomial-time complete (NP-complete) problems, and a backtracking search mechanism is an important technique for solving them (Floyd 1967).

One of the most popular tree search mechanisms is chronological backtracking. However, a chronological backtracking approach is not very efficient because conflicts are detected late; that is, the approach is oriented toward *recovering* from failures rather than *avoiding* them to begin with. The search space is reduced only after a failure is detected, and the performance of this technique is drastically reduced as the problem size increases. Another drawback of using chronological backtracking, for the same reason, is encountering repeated failures, sometimes called “thrashing.” The presence of late detection and “thrashing” has led researchers to develop consistency techniques that can achieve superior pruning of the search tree. This strategy uses constraints actively, rather than passively.
Chapter 11: The Constraint Programming Solver

Constraint Propagation

A more efficient technique than backtracking is constraint propagation, which uses consistency techniques to effectively prune the domains of variables. Consistency techniques are based on the idea of a priori pruning, which uses the constraint to reduce the domains of the variables. Consistency techniques are also known as relaxation algorithms (Tsang 1993), and the process is also called problem reduction, domain filtering, or pruning.

One of the earliest applications of consistency techniques was in the AI field to solve the scene labeling problem, which required recognizing objects in three-dimensional space by interpreting two-dimensional line drawings of the object. The Waltz filtering algorithm (Waltz 1975) analyzes line drawings by systematically labeling the edges and junctions while maintaining consistency between the labels.

Constraint propagation is characterized by the extent of propagation (also called the level of consistency) and whether domain propagation or interval propagation is the domain pruning scheme that is followed. In practice, interval propagation is preferred because of its lower computational costs. This mechanism is discussed in detail in Van Hentenryck (1989). However, constraint propagation is not a complete solution technique and needs to be complemented by a search technique in order to ensure success (Kumar 1992).

Finite-Domain Constraint Programming

Finite-domain constraint programming is an effective and complete solution technique that embeds incomplete constraint propagation techniques into a nondeterministic backtracking search mechanism that is implemented as follows: Whenever a node is visited, constraints are propagated to attain a desired level of consistency. If the domain of each variable reduces to a singleton set, the node represents a solution to the CSP. If the domain of a variable becomes empty, the node is pruned. Otherwise a variable is selected, its domain is distributed, and a new set of CSPs is generated, each of which is a child node of the current node. Several factors play a role in determining the outcome of this mechanism, such as the extent of propagation (or level of consistency enforced), the variable selection strategy, and the variable assignment or domain distribution strategy.

For example, the lack of any propagation reduces this technique to a simple generate-and-test approach, whereas performing consistency checking using variables that are already selected reduces this approach to chronological backtracking, one of the systematic search techniques. These are also known as look-back schemas, because they share the disadvantage of late conflict detection. Look-ahead schemas, on the other hand, work to prevent future conflicts. Some popular examples of look-ahead schemas, in increasing degree of consistency level, are forward checking (FC), partial look ahead (PLA), and full look ahead (LA) (Kumar 1992). Forward checking enforces consistency between the current variable and future variables; PLA and LA extend this even further to pairs of not yet instantiated variables.

Two important consequences of this technique are that inconsistencies are discovered early and that the current set of alternatives that are coherent with the existing partial solution is dynamically maintained. These consequences are powerful enough to prune large parts of the search tree, thereby reducing the “combinatorial explosion” of the search process. However, although constraint propagation at each node results in fewer nodes in the search tree, the processing at each node is more expensive. The ideal scenario is to strike a balance between the extent of propagation and the subsequent computation cost.

Variable selection is another strategy that can affect the solution process. The order in which variables are chosen for instantiation can have a substantial impact on the complexity of the backtrack search. Several heuristics have been developed and analyzed for selecting variable ordering. One of the most common ones is a dynamic heuristic based on the fail-first principle (Haralick and Elliott 1980), which selects the variable
whose domain has minimal size. Subsequent analysis of this heuristic by several researchers has validated this strategy as providing substantial improvement for a significant class of problems. Another popular strategy is to instantiate the most constrained variable first. Both these strategies are based on the principle of selecting the variable most likely to fail and detecting such failures as early as possible.

The domain distribution strategy for a selected variable is yet another area that can influence the performance of a backtracking search. However, good value-ordering heuristics are expected to be very problem-specific (Kumar 1992).

Consistency Techniques

The CLP solver features a full look-ahead algorithm for standard CSPs that follows a strategy of maintaining a version of generalized arc consistency that is based on the AC-3 consistency routine (Mackworth 1977). This strategy maintains consistency between the selected variables and the unassigned variables and also maintains consistency between unassigned variables.

Selection Strategy

A search algorithm for CSPs searches systematically through the possible assignments of values to variables. The order in which a variable is selected can be based on a static ordering, which is determined before the search begins, or on a dynamic ordering, in which the choice of the next variable depends on the current state of the search. The VARSELECT= option in the SOLVE statement defines the variable selection strategy for a standard CSP. The default strategy is the dynamic MINR strategy, which selects the variable that has the smallest range.

Assignment Strategy

After a variable is selected, the assignment strategy dictates the value that is assigned to it. For variables, the assignment strategy is specified in the VARASSIGN= option in the SOLVE statement. The default assignment strategy selects the minimum value from the domain of the selected variable.

Differences between PROC OPTMODEL and PROC CLP

You can invoke the CLP solver from PROC OPTMODEL by using any of the predicates that are defined in the standard mode of PROC CLP. The standard mode gives you access to all-different, element, GCC, linear, pack, and reify constraints.

To replicate the FOREACH statement that PROC CLP supports, you can use PROC OPTMODEL’s expressions and iteration machinery. For an example, see the Eight Queens example in the “Getting Started: CLP Solver” on page 442. For more information about the FOREACH predicate, see SAS/OR User’s Guide: Constraint Programming.

PROC OPTMODEL has different syntax and semantics for variable declarations:

- Because all CLP variables are discrete, you must declare every variable that a CLP model uses as integer or binary.
- The default variable bounds in PROC OPTMODEL are \(-\infty\) to \(\infty\). The default lower bound in PROC CLP is 0. Thus, to replicate the behavior of PROC CLP, you must explicitly add a lower bound of 0 to the variable declaration.
Macro Variable _OROPTMODEL_

The OPTMODEL procedure always creates and initializes a SAS macro variable called _OROPTMODEL_, which contains a character string. The variable contains information about the execution of the most recently invoked solver.

Each keyword and value pair in _OROPTMODEL_ also appears in two other places: the PROC OPTMODEL automatic arrays _OROPTMODEL_NUM_ and _OROPTMODEL_STR_; and the ODS tables ProblemSummary and SolutionSummary, which appear after a SOLVE statement, unless you set the PRINTLEVEL= option to NONE. You can use these variables to obtain details about the solution.

After the solver is called, the various keywords in the variable are interpreted as follows.

**STATUS** indicates the solver status at termination. It can take one of the following values:

- **OK**: The solver terminated normally.
- **SYNTAX_ERROR**: The use of syntax is incorrect.
- **DATA_ERROR**: The input data are inconsistent.
- **OUT_OF_MEMORY**: Insufficient memory was allocated to the procedure.
- **IO_ERROR**: A problem in reading or writing of data has occurred.
- **SEMANTIC_ERROR**: An evaluation error, such as an invalid operand type, has occurred.
- **ERROR**: The status cannot be classified into any of the preceding categories.

**SOLUTION_STATUS** indicates the solution status at termination. It can take one of the following values:

- **ABORT_NOSOL**: The solver was stopped by the user and did not find a solution.
- **ABORT_SOL**: The solver was stopped by the user but still found a solution.
- **ALL_SOLUTIONS**: All solutions were found.
- **BAD_PROBLEM_TYPE**: The problem type is not supported by the solver.
- **CONDITIONAL_OPTIMAL**: The optimality of the solution cannot be proven.
- **ERROR**: The algorithm encountered an error.
- **FAIL_NOSOL**: The solver stopped because of errors and did not find a solution.
- **FAIL_SOL**: The solver stopped because of errors but still found a solution.
- **INFEASIBLE**: The problem is infeasible.
- **INTERRUPTED**: The solver was interrupted by the system or the user before completing its work.
- **OK**: The algorithm terminated normally.
OPTIMAL
The solution is optimal.

OUTMEM_NOSOL
The solver ran out of memory and either did not find a solution or failed to output the solution because of insufficient memory.

OUTMEM_SOL
The solver ran out of memory but still found a solution.

SOLUTION_LIM
The solver reached the maximum number of solutions specified in the MAXSOLNS= option.

TIME_LIM_NOSOL
The solver reached the execution time limit specified in the MAXTIME= option and did not find a solution.

TIME_LIM_SOL
The solver reached the execution time limit specified in the MAXTIME= option and found a solution.

UNBOUNDED
The problem is unbounded.

ALGORITHM
indicates the algorithm that produces the solution data in the macro variable. This term appears only when STATUS=OK. It can take only one value in the CLP solver: CLP, which indicates that the constraint satisfaction algorithm produced the solution data.

OBJECTIVE
indicates the objective value that the solver obtained at termination. If a problem does not have an explicit objective, the value of this keyword in the _OROPTMODEL_ macro variable is missing (.).

PRESOLVE_TIME
indicates the time (in seconds) that the algorithm used for preprocessing. You can use the TIMETYPE= option to select real time or CPU time.

SOLUTION_TIME
indicates the time (in seconds) that the algorithm used to perform iterations to solve the problem. You can use the TIMETYPE= option to select real time or CPU time.

SOLUTIONS_FOUND
indicates the number of solutions found, which might be 0 if the problem is infeasible. This keyword is always present in the solution status when you call the CLP solver. The value might not be the total number of solutions possible (for example, if the solver reached its time limit).

Examples: CLP Solver

Example 11.1: Logic-Based Puzzles
Many logic-based puzzles can be formulated as CSPs. Several such puzzles are shown in this example.
Sudoku

Sudoku is a logic-based, combinatorial number-placement puzzle that uses a partially filled $9 \times 9$ grid. The objective is to fill the grid with the digits 1 to 9 so that each column, each row, and each of the nine $3 \times 3$ blocks contains only one of each digit. Figure 11.1.1 shows an example of a sudoku grid.

Output 11.1.1 Example of an Unsolved Sudoku Grid

<table>
<thead>
<tr>
<th></th>
<th>5</th>
<th>7</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>9</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
</tbody>
</table>

This example illustrates the use of the all-different constraint to solve the preceding sudoku problem. The data set InData contains the partially filled values for the grid.

```plaintext
data InData;
  input C1-C9;
data lines;
  . . 5 . . 7 . . 1
  . 7 . . 9 . . 3 .
  . . 6 . . . . .
  . . 3 . . 1 . . 5
  . 9 . . 8 . . 2 .
  1 . . 2 . . 4 . .
  . . 2 . . 6 . . 9
  . . 4 . . 8 . .
  8 . . 1 . . 5 . .
;```

Let the variable $X_{ij} (i = 1, \ldots, 9, j = 1, \ldots, 9)$ represent the value of cell $(i, j)$ in the grid. The domain of each of these variables is $[1, 9]$. When cell $(i, j)$ is not missing in the data set, $X_{ij}$ is fixed to that value.

Three sets of all-different constraints specify the required rules for each row, each column, and each of the $3 \times 3$ blocks. The RowCon constraint forces all values in row $i$ to be different, the ColumnCon constraint forces all values in column $j$ to be different, and the BlockCon constraint forces all values in each block to be different.

The following statements express the preceding constraints in PROC OPTMODEL and solve the sudoku puzzle:
Example 11.1: Logic-Based Puzzles

proc optmodel;
    /* Declare variables */
    set ROWS = 1..9;
    set COLS = ROWS; /* Use an alias for convenience and clarity */
    var X {ROWS, COLS} >= 1 <= 9 integer;

    /* Nine row constraints */
    con RowCon {i in ROWS}:
        alldiff({j in COLS} X[i,j]);

    /* Nine column constraints */
    con ColCon {j in COLS}:
        alldiff({i in ROWS} X[i,j]);

    /* Nine 3x3 block constraints */
    con BlockCon {s in 0..2, t in 0..2}:
        alldiff({i in 3*s+1..3*s+3, j in 3*t+1..3*t+3} X[i,j]);

    /* Fix variables to cell values */
    /* X[i,j] = c[i,j] if c[i,j] is not missing */
    num c {ROWS, COLS};
    read data indata into [_N_] {j in COLS} <c[_N_,j]=col('C'||j)>;
    for {i in ROWS, j in COLS: c[i,j] ne .}
        fix X[i,j] = c[i,j];

    solve;
    quit;

Output 11.1.2 shows the solution.

**Output 11.1.2** Solution of the Sudoku Grid

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>8</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>7</td>
<td>6</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>1</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td>2</td>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>1</td>
<td>8</td>
<td>9</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>6</td>
<td>1</td>
<td>8</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>8</td>
<td>2</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>2</td>
<td>8</td>
<td>3</td>
<td>6</td>
<td>7</td>
<td>1</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>6</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>9</td>
<td>1</td>
<td>7</td>
<td>2</td>
<td>5</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>
The basic structure of the classical sudoku problem can easily be extended to formulate more complex puzzles. One such example is the Pi Day sudoku puzzle.

Pi Day is a celebration of the number $\pi$ that occurs every March 14. In honor of Pi Day, Brainfreeze Puzzles (Riley and Taalman 2008) celebrates this day with a special 12 $\times$ 12 grid sudoku puzzle. The 2008 Pi Day sudoku puzzle is shown in Figure 11.1.3.

The rules for this puzzle are a little different from the rules for standard sudoku:

1. Rather than using regular 3 $\times$ 3 blocks, this puzzle uses jigsaw regions such that highlighted regions in the middle resemble the Greek letter $\pi$. Each jigsaw region consists of 12 contiguous cells.

2. The first 12 digits of $\pi$ are used instead of the digits 1–9. Each row, column, and jigsaw region contains the first 12 digits of $\pi$ (314159265358) in some order. In other words, there are no 7s; one each of 2, 4, 6, 8, and 9; two each of 1 and 3; and three 5s.

To generalize the original sudoku model:

1. Replace the expression that calculates the starting and ending cells of a region by an array that maps each cell to one region.

2. Replace the all-different constraints with GCC constraints. GCC constraints describe how often each value can be assigned to a set of variables. Conceptually, an all-different constraint is a specialized GCC constraint in which both the lower bound and the upper bound of every value is 1.

The data set Raw contains the partially filled values for the grid. It contains missing values where the cell does not yet contain a number.
data Raw;
    input C1-C12;
    datalines;
    3 . . 1 5 4 . . 1 . 9 5
    . 1 . . 3 . . . 1 3 6
    . . 4 . . 3 . 8 . . 2 .
    5 . . 1 . . 9 2 5 . . 1
    . 9 . . 5 . . 5 . . . .
    5 8 1 . . 9 2 5 . . 1
    . 5 . 8 . . 2 . . 5 5 3
    . . . 5 . . 6 . . 1 .
    2 . . 5 1 5 . . 5 . . 9
    . 6 . . 4 . 1 . . 3 . .
    1 5 1 . . . 5 . . 5 .
    5 5 . 4 . . 3 1 6 . . 8
    ;

The following statements define the GCC constraints in order to find all solutions of the Pi Day sudoku 2008 puzzle:

proc optmodel;
    set ROWS = 1..12;
    /* These declarations are inexpensive and improve clarity: */
    set COLS = ROWS, REGIONS = ROWS, CELLS = ROWS cross COLS;

    /* specify a 12x12 array of region identifiers. */
    /* The spacing is just to make the regions easier to visualize. */
    num region{CELLS} = [
        1 1 1 2 2 2 2 2 2 3 3 3
        1 1 1 2 2 2 2 2 2 3 3 3
        1 1 4 4 4 4 5 5 5 5 3 3
        1 1 4 4 4 4 5 5 5 5 3 3
        1 1 4 4 4 4 5 5 5 5 3 3
        6 6 6 7 7 8 8 9 9 10 10 10
        6 6 6 7 7 8 8 9 9 10 10 10
        6 6 6 7 7 8 8 9 9 10 10 10
        6 6 6 7 7 8 8 9 9 10 10 10
        11 11 11 7 7 8 8 9 9 12 12 12
        11 11 11 7 7 8 8 9 9 12 12 12
        11 11 11 11 11 12 12 12 12 ];

    /* Each area must contain two 1's, two 3's, three 5's, no 7's, */
    /* and one for each of other values from 1 to 9. */
    num nTimes{1..9} = [2 1 2 1 3 1 0 1 1];
    /* For convenience, create a triplet set version of nTimes. */
    /* In this model, GCC's lower and upper bounds are the same. */
    set N_TIMES = setof{ni in 1..9} <ni,nTimes[ni],nTimes[ni]>
    var X {CELLS} >= 1 <= 9 integer;

    /* X[i,j] = c[i,j] if c[i,j] is not missing */
num c {CELLS};
read data raw into [_N_] {j in COLS} <c[_N_,j]=col('C'||j)>;
for {<i,j> in CELLS: c[i,j] ne .}
  fix X[i,j] = c[i,j];

con RowCon {i in ROWS}:
  gcc({j in COLS} X[i,j], N_TIMES);

con ColCon {j in COLS}:
  gcc({i in ROWS} X[i,j], N_TIMES);

con RegionCon {ri in REGIONS}:
  gcc({<i,j> in CELLS: region[i,j] = ri} X[i,j], N_TIMES);

solve;
/* Replicate typical PROC CLP output from PROC OPTMODEL arrays */
create data pdsout from
  {<i,j> in ROWS cross COLS}<col('X_'||i||'_'||j)=X[i,j]>;
quit;

The only solution of the 2008 Pi Day sudoku puzzle is shown in Output 11.1.4.

Output 11.1.4 Solution to Pi Day Sudoku 2008

<table>
<thead>
<tr>
<th>Pi Day Sudoku 2008</th>
</tr>
</thead>
<tbody>
<tr>
<td>Obs  C1  C2  C3  C4  C5  C6  C7  C8  C9  C10  C11  C12</td>
</tr>
<tr>
<td>-----------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
</tbody>
</table>

The corresponding completed grid is shown in Figure 11.1.5.
Output 11.1.5 Solution to Pi Day Sudoku 2008

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>6</td>
<td>3</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>9</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>9</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>3</td>
<td>8</td>
<td>1</td>
<td>6</td>
<td>2</td>
<td>4</td>
<td>9</td>
<td>5</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>8</td>
<td>9</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>8</td>
<td>5</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>

**Magic Square**

A magic square is an arrangement of the distinct positive integers from 1 to \( n^2 \) in an \( n \times n \) matrix such that the sum of the numbers of any row, any column, or any main diagonal is the same number, known as the magic constant. The magic constant of a normal magic square depends only on \( n \) and has the value \( n(n^2 + 1)/2 \).

This example illustrates the use of the MINRMAXC selection strategy, which is controlled by the VAR-SELECT= option.

```sas
%macro magic(n);
proc optmodel;
  num n = &n;
  /* magic constant */
  num sum = n*(n^2+1)/2;
  set ROWS = 1..n;
  set COLS = 1..n;

  /* X[i,j] = entry (i,j) */
  var X {ROWS, COLS} >= 1 <= n^2 integer;

  /* row sums */
  con RowCon {i in ROWS}:
    sum {j in COLS} X[i,j] = sum;

  /* column sums */
  con ColCon {j in COLS}:
    sum {i in ROWS} X[i,j] = sum;

  /* diagonal: upper left to lower right */
  con DiagCon:
    sum {i in ROWS} X[i,i] = sum;
%mend magic;
```
The solution is displayed in Output 11.1.6.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>39</td>
<td>24</td>
<td>40</td>
<td>31</td>
<td>38</td>
</tr>
<tr>
<td>43</td>
<td>4</td>
<td>34</td>
<td>41</td>
<td>42</td>
<td>5</td>
</tr>
<tr>
<td>18</td>
<td>20</td>
<td>23</td>
<td>29</td>
<td>30</td>
<td>22</td>
</tr>
<tr>
<td>36</td>
<td>37</td>
<td>25</td>
<td>3</td>
<td>7</td>
<td>35</td>
</tr>
<tr>
<td>14</td>
<td>19</td>
<td>44</td>
<td>13</td>
<td>47</td>
<td>12</td>
</tr>
<tr>
<td>17</td>
<td>45</td>
<td>9</td>
<td>21</td>
<td>8</td>
<td>48</td>
</tr>
<tr>
<td>46</td>
<td>11</td>
<td>16</td>
<td>28</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

**Example 11.2: Alphabet Blocks Problem**

This example illustrates the use of the global cardinality constraint (GCC). The alphabet blocks problem consists of finding an arrangement of letters on four alphabet blocks. Each alphabet block has a single letter on each of its six sides. Collectively, the four blocks contain every letter of the alphabet except Q and Z. By arranging the blocks in various ways, the following words should be spelled out: BAKE, ONYX, ECHO, OVAL, GIRD, SMUG, JUMP, TORN, LUCK, VINY, LUSH, and WRAP.
You can formulate this problem as a CSP by representing each of the 24 letters as an integer variable. The domain of each variable is the set \( \{1, 2, 3, 4\} \), which represents block1 through block4. The assignment \( A = 1 \) indicates that the letter A is on a side of block1. Because each block has six sides, each value \( v \) in \( \{1, 2, 3, 4\} \) must be assigned to exactly six variables so that each side of a block has a letter on it. This restriction can be formulated as a global cardinality constraint over all 24 variables, with common lower and upper bounds set equal to 6.

Moreover, in order to spell all the words listed previously, the four letters in each of the 12 words must be on different blocks. Another GCC statement that specifies 12 global cardinality constraints enforces these conditions. You can also formulate these restrictions by using 12 all-different constraints. Finally, four FIX statements break the symmetries that blocks are interchangeable. These constraints preset the blocks that contain the letters B, A, K, and E as block1, block2, block3, and block4, respectively.

The complete representation of the problem is as follows:

```plaintext
proc optmodel;
/* Each letter except Q and Z is represented with a variable. */
/* The domain of each variable is the set of 4 blocks, */
/* or \( \{1, 2, 3, 4\} \) for short. */
set LETTERS = / A B C D E F G H I J K L M N O P R S T U V W X Y /;
var Block {LETTERS} integer >= 1 <= 4;
set BLOCKS = 1..4;

/* There are exactly 6 letters on each alphabet block */
con SixLettersPerBlock:
   gcc(Block, setof {b in BLOCKS} <b,6,6>);

/* The letters in each word must be on different blocks. */
set WORDS = / BAKE ONYX ECHO OVAL GIRD TORN LUCK VINY LUSH WRAP /;
con CanSpell {w in WORDS}:
   gcc({k in 1..length(w)} Block[char(w,k)], setof {b in BLOCKS} <b,0,1>);

/* Note 2: These restrictions can also be enforced by ALLDIFF constraints: */
con CanSpellv2 {w in WORDS}:
   alldiff({k in 1..length(w)} Block[char(w,k)]);
/*
/* Breaking the symmetry that blocks can be interchanged by setting */
/* the block that contains the letter B as block1, the block that */
/* contains the letter A as block2, etc. */
for {k in 1..length('BAKE')} fix Block[char('BAKE',k)] = k;

solve;
print Block;
quit;
```

The solution to this problem is shown in Output 11.2.1.
### Output 11.2.1 Solution to Alphabet Blocks Problem

#### The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
<td>Minimization</td>
</tr>
<tr>
<td>Objective Function</td>
<td>(0)</td>
</tr>
<tr>
<td>Objective Type</td>
<td>Constant</td>
</tr>
<tr>
<td>Number of Variables</td>
<td>24</td>
</tr>
<tr>
<td>Bounded Above</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
<td>20</td>
</tr>
<tr>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>Fixed</td>
<td>4</td>
</tr>
<tr>
<td>Binary</td>
<td>1</td>
</tr>
<tr>
<td>Integer</td>
<td>23</td>
</tr>
<tr>
<td>Number of Constraints</td>
<td>13</td>
</tr>
<tr>
<td>Linear LE (&lt;=)</td>
<td>0</td>
</tr>
<tr>
<td>Linear EQ (=)</td>
<td>0</td>
</tr>
<tr>
<td>Linear GE (&gt;=)</td>
<td>0</td>
</tr>
<tr>
<td>Linear Range</td>
<td>0</td>
</tr>
<tr>
<td>Alldiff</td>
<td>0</td>
</tr>
<tr>
<td>Element</td>
<td>0</td>
</tr>
<tr>
<td>GCC</td>
<td>13</td>
</tr>
<tr>
<td>Lexico (&lt;=)</td>
<td>0</td>
</tr>
<tr>
<td>Lexico (&lt;)</td>
<td>0</td>
</tr>
<tr>
<td>Pack</td>
<td>0</td>
</tr>
<tr>
<td>Reify</td>
<td>0</td>
</tr>
</tbody>
</table>

| Constraint Coefficients     | 0   |

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>CLP</td>
</tr>
<tr>
<td>Variable Selection</td>
<td>MINR</td>
</tr>
<tr>
<td>Objective Function</td>
<td>(0)</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Solution Limit Reached</td>
</tr>
<tr>
<td>Objective Value</td>
<td>0</td>
</tr>
<tr>
<td>Solutions Found</td>
<td>1</td>
</tr>
<tr>
<td>Presolve Time</td>
<td>0.00</td>
</tr>
<tr>
<td>Solution Time</td>
<td>0.00</td>
</tr>
</tbody>
</table>
Example 11.3: Work-Shift Scheduling Problem

This example illustrates the use of the GCC constraint to find a feasible solution to a work-shift scheduling problem and then the use of the element constraint to incorporate cost information in order to find a minimum-cost schedule.

Six workers (Alan, Bob, Juanita, Mike, Ravi, and Aisha) are to be assigned to three working shifts. The first shift needs at least one and at most four people; the second shift needs at least two and at most three people; and the third shift needs exactly two people. Alan cannot work on the first shift; Bob can work only on the third shift. The others can work on any shift. The objective is to find a feasible assignment for this problem.

You can model the minimum and maximum shift requirements by using a GCC constraint and formulate the problem as a standard CSP. The variables \( W[1], \ldots, W[6] \) identify the shift to which each of the six workers is assigned: Alan, Bob, Juanita, Mike, Ravi, and Aisha.

```
proc optmodel;
   /* Six workers (Alan, Bob, Juanita, Mike, Ravi and Aisha) are to be assigned to 3 working shifts. */
   set WORKERS = 1..6;
   var W {WORKERS} integer >= 1 <= 3;

   /* The first shift needs at least 1 and at most 4 people; */
```

the second shift needs at least 2 and at most 3 people; and the third shift needs exactly 2 people. */
con ShiftNeeds:
    gcc(W, /<1,1,1>,<2,2,3>,<3,2,2>/);

/* Alan doesn't work on the first shift. */
con Alan:
    W[1] ne 1;

/* Bob works only on the third shift. */
fix W[2] = 3;

solve;
print W;
quit;

The resulting assignment is shown in Output 11.3.1.

Output 11.3.1  Solution to Work-Shift Scheduling Problem
Solution to Work-Shift Scheduling Problem

<table>
<thead>
<tr>
<th>Obs</th>
<th>W1</th>
<th>W2</th>
<th>W3</th>
<th>W4</th>
<th>W5</th>
<th>W6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

A Gantt chart of the corresponding schedule is displayed in Output 11.3.2.

Output 11.3.2  Work-Shift Schedule

Example 3: Feasible Work-Shift Assignment
Now suppose that every work-shift assignment has a cost associated with it and that the objective of interest is to determine the schedule that has the lowest cost.

The costs of assigning the workers to the different shifts are shown in Table 11.5. A dash ("-") in position \((i, j)\) indicates that worker \(i\) cannot work on shift \(j\).

<table>
<thead>
<tr>
<th></th>
<th>Shift 1</th>
<th>Shift 2</th>
<th>Shift 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alan</td>
<td>-</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>Bob</td>
<td>-</td>
<td>-</td>
<td>6</td>
</tr>
<tr>
<td>Juanita</td>
<td>16</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>Mike</td>
<td>10</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Ravi</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>Aisha</td>
<td>12</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

Based on the cost structure in Table 11.5, the previously derived schedule has a cost of $54. The objective now is to determine the optimal schedule—one that results in the minimum cost.

Let the variable \(C_i\) represent the cost of assigning worker \(i\) to a shift. This variable is shift-dependent and is given a high value (for example, 100) if the worker cannot be assigned to a shift. The costs can also be interpreted as preferences if desired. You can use an element constraint to associate the cost \(C_i\) with the shift assignment for each worker. For example, \(C_1\), the cost of assigning Alan to a shift, can be determined by the constraint \(\text{ELEMENT}(W_1, (100, 12, 10), C_1)\).

By adding a linear constraint, \(\sum_{i=1}^{n} C_i \leq \text{obj}\), you can limit the solutions to feasible schedules that cost no more than \(\text{obj}\). Although an upper bound of $100 is used in this example, it would suffice to use an upper bound of $54, the cost of the feasible schedule that was determined earlier.

```plaintext
proc optmodel;
/* Six workers (Alan, Bob, Juanita, Mike, Ravi and Aisha) are to be assigned to 3 working shifts. */
set WORKERS = 1..6;
set SHIFTS = 1..3;
var W {WORKERS} integer >= 1 <= 3;
var C {WORKERS} integer >= 1 <= 100;
/* The first shift needs at least 1 and at most 4 people; 
   the second shift needs at least 2 and at most 3 people; 
   and the third shift needs exactly 2 people. */
con GccCon:
   gcc(W, /<1,1,4>,<2,2,3>,<3,2,2>/);
/* Alan doesn't work on the first shift. */
con Alan:
   W[1] ne 1;
/* Bob works only on the third shift. */
fix W[2] = 3;
/* Specify the costs of assigning the workers to the shifts. 
   Use 100 (a large number) to indicate an assignment */
```
Chapter 11: The Constraint Programming Solver

num a (WORKERS, SHIFTS) = [
  100, 12, 10,
  100, 100, 6,
  16, 8, 12
  10, 6, 8
  6, 6, 8
  12, 4, 4
];
con ElementCon {j in WORKERS}:
  element(W[j], {k in SHIFTS} a[j,k], C[j]);

/* Minimize total cost. */
min TotalCost = sum {j in WORKERS} C[j];
con TotalCost_bounds:
  1 <= TotalCost <= 100;

solve;
print W;
create data clpout from
  {j in WORKERS} <col('W'||j)=W[j]> {j in WORKERS} <col('C'||j)=C[j]>
quit;

The cost of the optimal schedule, which corresponds to the solution shown in the following output, is $40.

Solution to Optimal Work-Shift Scheduling Problem

<table>
<thead>
<tr>
<th>Obs</th>
<th>W1</th>
<th>W2</th>
<th>W3</th>
<th>W4</th>
<th>W5</th>
<th>W6</th>
<th>C1</th>
<th>C2</th>
<th>C3</th>
<th>C4</th>
<th>C5</th>
<th>C6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>10</td>
<td>6</td>
<td>8</td>
<td>6</td>
<td>6</td>
<td>4</td>
</tr>
</tbody>
</table>

The minimum-cost schedule is displayed in the Gantt chart in Output 11.3.3.
Example 11.4: A Nonlinear Optimization Problem

This example illustrates how you can use the element constraint to represent almost any function between two variables in addition to representing nonstandard domains. Consider the following nonlinear optimization problem:

\[
\text{maximize } f(x) = x_1^3 + 5x_2 - 2x_3 \\
\text{subject to } \left\{ \begin{array}{c}
x_1 - \frac{1}{2}x_2 + x_3^2 \leq 50 \\
\text{mod}(x_1, 4) + \frac{1}{25}x_2 \geq 1.5 \\
\end{array} \right.
\]

where \(x_1\) is any integer in \([-5, 5]\), \(x_2\) is any odd integer in \([-5, 9]\), and \(x_3\) is any integer in \([1, 10]\).

You can solve this problem by introducing four artificial variables, \(y_1 - y_4\), to represent each of the nonlinear terms. Let \(y_1 = x_1^3\), \(y_2 = 2x_3\), \(y_3 = x_3^2\), and \(y_4 = \text{mod}(x_1, 4)\). You can represent the domains of \(x_1\) and \(x_2\) (which are not consecutive integers that start from 1) by using element constraints and index variables. For example, any of the following three element constraints specifies that the domain of \(x_2\) is the set of odd integers in \([-5, 9]\):

\[
\text{con element}(z2, -5 -3 -1 \ 1 \ 3 \ 5 \ 7 \ 9, x2); \\
\text{con element}(z2, \{ri in -5..9 by 2\} ri, x2); \\
\text{num range}(ri in -5..9 by 2) = ri; \\
\text{con element}(z2, \text{range}, x2);
\]

Any functional dependencies on \(x_1\) or \(x_2\) can now be defined using \(z_1\) or \(z_2\), respectively, as the index variable in an element constraint. Because the domain of \(x_3\) is \([1, 10]\), you can directly use \(x_3\) as the index variable in an element constraint to define dependencies on \(x_3\).

For example, the following constraint specifies the function \(y_1 = x_1^3, x_1 \in [-5, 5]\):
To solve the problem, define the objective function as demonstrated in the following statements:

```plaintext
proc optmodel;
    set DOM{1..3} = [ (-5 .. 5) (-5 .. 9 by 2) (1 .. 10) ];
    var X{i in 1..3} integer >= min{j in DOM[i]} j <= max{j in DOM[i]} j;

    /* map the domain of X[1] and X[2] to 1 .. list size */
    var Z {1..2} integer;
    /* map nonlinear expressions */
    var Y {1..4} integer;

    /* Use an element constraint to represent noncontiguous domains */
    /* domains with negative numbers, and nonlinear functions. */
    /* Z[2] does not appear anywhere else. Its only purpose is */
    /* to restrict X[2] to take a value from DOM[2]. */
    con MapDomainTo1ToCard{i in 1..2}:
        element(Z[i], {k in DOM[i]} k, X[i]);

    /* Functional Dependencies on X[1] */
    con Y1:
        element(Z[1], {k in DOM[1]} (k^3), Y[1]);

    /* Y[4] = mod(X[1], 4) */
    con Y4:
        element(Z[1], {k in DOM[1]} (mod(k,4)), Y[4]);

    /* Functional Dependencies on X[3] */
    con Y2:
        element(X[3], {k in DOM[3]} (2^k), Y[2]);

    con Y3:
        element(X[3], {k in DOM[3]} (k^2), Y[3]);

    con Con1:

    /* mod(X[1],4) + 0.25 * X[2] >= 1.5 */
    con Con2:
        Y[4] + 0.25 * X[2] >= 1.5;


solve;
    print X Y Z;
quit;
```
Example 11.4: A Nonlinear Optimization Problem

Output 11.4.1 shows the solution that corresponds to the optimal objective value of 168.

**Output 11.4.1 Nonlinear Optimization Problem Solution**

**The OPTMODEL Procedure**

**Problem Summary**

<table>
<thead>
<tr>
<th>Objective Sense</th>
<th>Maximization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Function</td>
<td>Objective</td>
</tr>
<tr>
<td>Objective Type</td>
<td>Linear</td>
</tr>
</tbody>
</table>

Number of Variables: 9  
Bounded Above: 0  
Bounded Below: 0  
Bounded Below and Above: 3  
Free: 6  
Fixed: 0  
Binary: 0  
Integer: 9

Number of Constraints: 8  
Linear LE (≤): 1  
Linear EQ (=): 0  
Linear GE (≥): 1  
Linear Range: 0  
Alldiff: 0  
Element: 6  
GCC: 0  
Lexico (≤): 0  
Lexico (<): 0  
Pack: 0  
Reify: 0

**Constraint Coefficients**  

**Solution Summary**

<table>
<thead>
<tr>
<th>Solver</th>
<th>CLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable Selection</td>
<td>MINR</td>
</tr>
<tr>
<td>Objective Function</td>
<td>Objective</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Optimal</td>
</tr>
<tr>
<td>Objective Value</td>
<td>168</td>
</tr>
</tbody>
</table>

Solutions Found: 1  
Presolve Time: 0.00  
Solution Time: 0.00

```
[1]  X   Y   Z  
  1  5  125  11  
  2   9   2   8  
  3   1   1   3  
  4   1   1   3
```
Example 11.5: Car Painting Problem

The car painting process is an important part of the automobile manufacturing industry. Purging (the act of changing colors during assembly) is expensive because of the added cost of wasted paint and solvents from each color change and the extra time that the purging process requires. The objective of the car painting problem is to sequence the cars in the assembly line in order to minimize the number of paint color changes (Sokol 2002; Trick 2004).

Suppose an assembly line contains 10 cars, which are ordered 1, 2, . . . , 10. A car must be painted within three positions of its assembly order. For example, car 5 can be painted in positions 2 through 8. Cars 1, 5, and 9 are red; 2, 6, and 10 are blue; 3 and 7 green; and 4 and 8 are yellow. The initial sequence 1, 2, . . . , 10 corresponds to the color pattern RBGYRBGYRB and has nine purgings. The objective is to find a solution that minimizes the number of purgings.

This problem can be formulated as a CSP as follows:

- The input is the color of each car currently on the assembly line.
- The output variables are the slot in which each car will be painted and whether purging will be required after that painting operation.
- Set the bounds of the slot variables to their feasible range, at most three slots before or after the car’s current position.
- To determine whether purging is needed, use another variable to store the color of the car painted in each slot.
- Use an element constraint to determine the color used in each slot from the car assigned to that slot.
- Use a reify constraint to determine whether two consecutive slots are assigned different colors, and thus whether purging is required.
- Finally, use a linear constraint to limit the total number of purgings.

The following %CAR_PAINTING macro determines all feasible solutions for the number of purgings that is specified as a parameter to the macro:
Example 11.5: Car Painting Problem

%macro car_painting(purgings);
   proc optmodel;
      num nCars = 10;
      /* a car is identified by its original slots */
      set SLOTS = 1..nCars;

      /* maximum reshuffling of any car on the line*/
      num maxMove init 3;
      /* which car is in slot i. */
      var S {si in SLOTS} integer >= max(1, si - maxMove)
      <= min(nCars, si + maxMove) ;

      /* which color the car in slot i is. */
      /* Red=1; Blue=2; Green=3; Yellow=4 */
      num nColors=4;
      num colorOf{SLOTS} = [ 1 2 3 4 1 2 3 4 1 2 ];
      var C {SLOTS} integer >= 1 <= nColors;

      con ElementCon {i in SLOTS}:
         element(S[i], colorOf, C[i]);

      /* A car can be painted only once. */
      con PaintOnlyOnce:
         alldiff(S);

      /* Whether there is a purge after slot i. 
       You can ignore any purging that would happen at the end of the shift. */
      var P {SLOTS diff {nCars}} binary;

      con ReifyCon {i in SLOTS diff {nCars}}:
         reify(P[i], C[i] ne C[i+1]);

      /* Calculate the number of purgings. */
      con PurgingsCon:
         sum {i in SLOTS diff {nCars}} P[i] <= &purgings;

      solve with CLP / findall;
      /* Replicate typical PROC CLP output from PROC OPTMODEL arrays */
      create data car_ds(drop=k) from [k]={(1.._NSOL_)
         {i in SLOTS} <col('S'||i)=S[i].sol[k]>
         {i in SLOTS} <col('C'||i)=C[i].sol[k]>
      quit;
   %mend;
%
car_painting(5);

The problem is infeasible for four purgings. The CLP solver finds 87 possible solutions for the five-purgings problem. The solutions are sorted by the total distance that all cars are moved in the sequencing, which ranges from 12 to 22 slots. The first 15 solutions are displayed in the Gantt chart in Output 11.5.1. Each row represents a solution, and each color transition represents a paint purging.
Example 11.6: Scene Allocation Problem

The scene allocation problem consists of deciding when to shoot each scene of a movie in order to minimize the total production cost (Van Hentenryck 2002). Each scene involves a number of actors, and at most five scenes a day can be shot. All actors who appear in a scene must be present in the studio on the day the scene is shot. Each actor earns a daily rate for each day spent in the studio, regardless of the number of scenes in which he or she appears on that day. The goal is to shoot the movie for the lowest possible production cost.

The actors’ names, their daily fees, and the scenes in which they appear are contained in the Scene data set, which is shown in Output 11.6.1. The data set variables S_Var1, . . . , S_Var9 indicate the scenes in which the
actor appears. For example, the first observation indicates that Patt’s daily fee is 26,481 and that Patt appears in scenes 2, 5, 7, 10, 11, 13, 15, and 17.

### Output 11.6.1 Scene Data Set

<table>
<thead>
<tr>
<th>Obs</th>
<th>Number</th>
<th>Actor</th>
<th>DailyFee</th>
<th>S_Var1</th>
<th>S_Var2</th>
<th>S_Var3</th>
<th>S_Var4</th>
<th>S_Var5</th>
<th>S_Var6</th>
<th>S_Var7</th>
<th>S_Var8</th>
<th>S_Var9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Patt</td>
<td>26481</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>10</td>
<td>11</td>
<td>13</td>
<td>15</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Casta</td>
<td>25043</td>
<td>4</td>
<td>7</td>
<td>9</td>
<td>10</td>
<td>13</td>
<td>16</td>
<td>19</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Scolaro</td>
<td>30310</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>10</td>
<td>14</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>.</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Murphy</td>
<td>4085</td>
<td>2</td>
<td>8</td>
<td>12</td>
<td>13</td>
<td>15</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>Brown</td>
<td>7562</td>
<td>2</td>
<td>3</td>
<td>12</td>
<td>17</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>Hacket</td>
<td>9381</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>13</td>
<td>18</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>Anderson</td>
<td>8770</td>
<td>5</td>
<td>6</td>
<td>14</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>McDougal</td>
<td>5788</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>9</td>
<td>10</td>
<td>12</td>
<td>15</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>Mercer</td>
<td>7423</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>8</td>
<td>9</td>
<td>16</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>Spring</td>
<td>3303</td>
<td>5</td>
<td>6</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td>Thompson</td>
<td>9593</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

There are 19 scenes. At most 5 scenes can be filmed in one day, so at least four days are needed to schedule all the scenes \( \left\lceil \frac{19}{5} \right\rceil = 4 \). Let \( S_{jk} \) be a binary variable that equals 1 if scene \( j \) is shot on day \( k \). Let \( A_{ik} \) be another binary variable that equals 1 if actor \( i \) is present on day \( k \). The input daily_fee\(_i\) is the daily cost of the \( i \)th actor.

The objective function that represents the total production cost is

\[
\min \sum_{i=1}^{11} \sum_{k=1}^{4} \text{daily_fee}_i \times A_{ik}
\]

This example illustrates the use of symmetry-breaking constraints. In this model, the “1” in day 1 does not refer to sequence but simply to the label of the day. Thus, you can call day 1 the day on which scene 1 is shot, whichever day that is. Similarly, either scene 2 is shot on the same day as scene 1 (day 1) or it is shot on another day, which you can call day 2. Scene 3 is shot either on one of those two days or on another day. Adding constraints that eliminate symmetry can significantly improve the performance of a CLP model. In this model, the symmetry-breaking constraints prevent the solver from considering three other assignments that do not differ in any meaningful way.

The following PROC OPTMODEL statements implement these ideas:

```plaintext
proc optmodel;
set ACTORS;
str actor_name {ACTORS};
um daily_fee {ACTORS};
um most_scenes = 9; /* most scenes by any actor */
um scene_list {ACTORS, 1..most_scenes};
read data scene into ACTORS=[_N_]
actor_name=Actor daily_fee=DailyFee
{[j in 1..most_scenes] <scene_list[_N_,j]=col('S_Var'||j)>;
print actor_name daily_fee scene_list;

set SCENES_actor {i in ACTORS} =
(setof {j in 1..most_scenes} scene_list[i,j]) diff {.};
```
set SCENES = 1..19;
set DAYS = 1..4;

/* Indicates if actor i is present on day k. */
var A {ACTORS, DAYS} binary;

/* Indicates if scene j is shot on day k. */
var S {SCENES, DAYS} binary;

/* Every scene is shot exactly once.*/
con SceneCon {j in SCENES}:
  gcc((k in DAYS) S[j,k], {<1,1,1>});

/* At least 4 and at most 5 scenes are shot per day. */
con NumScenesPerDayCon {k in DAYS}:
  gcc((j in SCENES) S[j,k], {<1,4,5>});

/* Actors for a scene must be present on day of shooting. */
con LinkCon {i in ACTORS, j in SCENES_actor[i], k in DAYS}:
  S[j,k] <= A[i,k];

/* Symmetry-breaking constraints. Without loss of any generality, you
 can assume Scene1 to be shot on day 1, Scene2 to be shot on day 1
 or day 2, and Scene3 to be shot on either day 1, day 2, or day 3. */
fix S[1,1] = 1;
for (k in 2..4) fix S[1,k] = 0;
for (k in 3..4) fix S[2,k] = 0;
fix S[3,4] = 0;

/* If Scene2 is shot on day 1, (as opposed to day 2) */
/* then Scene3 can be shot on day 1 or day 2 (but not day 3). */
con Symmetry:
  S[2,1] + S[3,3] <= 1;

/* Minimize total cost. */
min TotalCost = sum {i in ACTORS, k in DAYS} daily_fee[i] * A[i,k];

/* Set lower and upper bounds for the objective value */
/* Lower bound: every actor appears on one day. */
/* Upper bound: every actor appears on all four days. */
um obj_lb = sum {i in ACTORS} daily_fee[i];
um obj_ub = sum {i in ACTORS, k in DAYS} daily_fee[i];
put obj_lb= obj_ub=;
con TotalCost_bounds:
  obj_lb <= TotalCost <= obj_ub;

solve with CLP / varselect=maxc;
quit;
The optimal production cost is 334,144, as reported in the _OROPTMODEL_ macro variable. The corresponding actor schedules and scene schedules are displayed in Output 11.6.2 and Output 11.6.3, respectively.

Output 11.6.2 Scene Allocation Problem: Actor Schedules
Example 11.7: Car Sequencing Problem

This problem is an instance of a category of problems known as the car sequencing problem. A considerable amount of literature discusses this problem (Dincbas, Simonis, and Van Hentenryck 1988; Gravel, Gagne, and Price 2005; Solnon et al. 2008).

A number of cars are to be produced on an assembly line. Each car is customized to include a specific set of options, such as air conditioning, a sunroof, a navigation system, and so on. The assembly line moves through several workstations for installation of these options. The cars cannot be positioned randomly, because each workstation has limited capacity and needs time to set itself up to install the options as the car moves in front of the station. These capacity constraints are formalized using constraints of the form $m$ out of $N$, which indicates that the workstation can install the option on $m$ out of every sequence of $N$ cars. The car sequencing problem is to determine a sequencing of the cars on the assembly line that satisfies the demand constraints for each set of car options and the capacity constraints for each workstation.

This example comes from Dincbas, Simonis, and Van Hentenryck (1988). Ten cars need to be customized with five possible options. A class of car is defined by a specific set of options; there are six classes of cars. The data are presented in Table 11.6.
Example 11.7: Car Sequencing Problem

Table 11.6 Option Installation Data

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Type</th>
<th>Capacity m/N</th>
<th>Car Class 1</th>
<th>Car Class 2</th>
<th>Car Class 3</th>
<th>Car Class 4</th>
<th>Car Class 5</th>
<th>Car Class 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option 1</td>
<td>1</td>
<td>1/2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Option 2</td>
<td>2</td>
<td>2/3</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Option 3</td>
<td>3</td>
<td>1/3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Option 4</td>
<td>4</td>
<td>2/5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Option 5</td>
<td>5</td>
<td>1/5</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

| Number of Cars | 1 | 1 | 2 | 2 | 2 | 2 |

For example, car class 4 requires installation of option 2 and option 4, and two cars of this class are required. The workstation for option 2 can process only two out of every sequence of three cars. The workstation for option 4 has even less capacity—two out of every five cars.

The data for this problem are used to create a SAS data set, which drives the generation of variables and constraints in PROC OPTMODEL.

The decision variables for this problem are shown in Table 11.7.

Table 11.7 Decision Variables

<table>
<thead>
<tr>
<th>Variable Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S{SLOTS} &gt;=1 &lt;= 6</td>
<td>$S_i$ is the class of cars assigned to slot $i$.</td>
</tr>
<tr>
<td>var O {SLOTS, OPTIONS} binary</td>
<td>$O_{ij} = 1$ if the class assigned to slot $i$ needs option $j$.</td>
</tr>
</tbody>
</table>

The following SAS statements express the workstation capacity constraints by using a set of linear constraints for each workstation. A single GCC constraint expresses the demand constraints for each car class. An element constraint for each option variable expresses the relationships between slot variables and option variables.

This model also includes a set of redundant constraints, in the sense that the preceding logical constraints correctly represent the set of feasible solutions. However, the redundant constraints provide the solver with further information specific to this problem, significantly improving the efficiency of domain propagation. Redundant constraints are a core fixture of CLP models. They can determine whether a model will linger and be unsolvable for real data or will produce instant results.

The idea behind the redundant constraint in this model is the following realization: if the workstation for option $j$ has capacity $r$ out of $s$, then at most $r$ cars in the sequence $(n - s + 1), \ldots, n$ can have option $j$, where $n$ is the total number of cars. Consequently, at least $n_j - r$ cars in the sequence $1, \ldots, n - s$ must have option $j$, where $n_j$ is the number of cars that have option $j$. Generalizing this further, at least $n_j - k \times r$ cars in the sequence $1, \ldots, (n - k \times s)$ must have option $j$, $k = 1, \ldots, \lfloor n/s \rfloor$. 


Chapter 11: The Constraint Programming Solver

data class_data;
   input class cars_cls;
datalines;
1 1
2 1
3 2
4 2
5 2
6 2;

data option_data;
   input option max blSz class1-class6;
datalines;
1 1 2 1 0 0 1 1
2 2 3 0 1 1 0 1
3 1 3 1 0 0 0 1 0
4 2 5 1 1 0 1 0 0
5 1 5 0 1 0 0 0;

%macro car_sequencing(outdata);
   proc optmodel;
      set CLASSES;
      num nClasses = card(CLASSES);
      num cars_cls {CLASSES};
      read data class_data into CLASSES=[class] cars_cls;

      set OPTIONS;
      num max {OPTIONS};
      num blSz {OPTIONS};
      num list {OPTIONS, CLASSES};
      num cars_opt {i in OPTIONS} = sum {k in CLASSES} cars_cls[k] * list[i,k];
      read data option_data into OPTIONS=[option] max blSz
         {k in CLASSES} <list[option,k]=col('class'||k)>;

      num nCars = sum {k in CLASSES} cars_cls[k];
      set SLOTS = 1..nCars;

      /* Declare Variables */
      /* Slot variables: S[i] - class of car assigned to Slot i */
       var S {SLOTS} integer >= 1 <= nClasses;

      /* Option variables: O[i,j]*/
      /* indicates if class assigned to Slot i needs Option j */
       var O {SLOTS, OPTIONS} binary;

      /* Capacity Constraints: for each option j */
      /* Install in at most max[j] out of every sequence of blSz[j] cars */
      con CapacityCon {j in OPTIONS, i in 0..(nCars-blSz[j])}:
         sum {k in 1..blSz[j]} O[i+k,j] <= max[j];

      /* Demand Constraints: for each class k */
/* Exactly cars_cls[k] cars */
con MeetDemandCon:
    gcc(S, setof(k in CLASSES) <k,cars_cls[k],cars_cls[k]>);

/* Element Constraints: For each slot i and each option j */
/* relate the slot variable to the option variables. */
/* O[i,j] = list[j,S[i]] */
con OptionsAtSlotCon {i in SLOTS, j in OPTIONS}:
    element(S[i], {k in CLASSES} list[j,k], O[i,j]);

/* Redundant Constraints to improve efficiency - for every */
/* option j. */
/* At most max[j] out of every sequence of blSz[j] cars */
/* requires option j. */
/* All the other slots contain at least cars_opt[j] - max[j]*/
/* cars with option j */
con BoundRemainingCon {j in OPTIONS, i in 1..(nCars/blSz[j])}:
    sum {k in 1..(nCars-i*blSz[j])} O[k,j] >= cars_opt[j] - i * max[j];

solve with CLP / varselect=minrmaxc findall;

/* Replicate typical PROC CLP output from PROC OPTMODEL arrays */
create data &outdata.(drop=sol) from [sol]=(1.._NSOL_)
    {i in SLOTS} <col('S_'||i)=S[i].sol[sol]>
    {i in SLOTS, j in OPTIONS} <col('O_'||i||'_'||j)=O[i,j].sol[sol]>;
quit;
%mend;
%carsequencing(sequence_out);

This problem has six solutions, as shown in Output 11.7.1.
Example 11.8: Balanced Incomplete Block Design

Balanced incomplete block design (BIBD) generation is a standard combinatorial problem from design theory. The concept was originally developed in the design of statistical experiments; applications have expanded to other fields, such as coding theory, network reliability, and cryptography. A BIBD is an arrangement of $v$ distinct objects into $b$ blocks such that the following conditions are met:

- Each block contains exactly $k$ distinct objects.
- Each object occurs in exactly $r$ different blocks.
- Every two distinct objects occur together in exactly $\lambda$ blocks.

A BIBD is therefore specified by its parameters $(v, b, r, k, \lambda)$. It can be proved that when a BIBD exists, its parameters must satisfy the following conditions:

- $rv = bk$
- $\lambda(v - 1) = r(k - 1)$
- $b \geq v$
The preceding conditions are not sufficient to guarantee the existence of a BIBD (Prestwich 2001). For example, the parameters \( (15, 21, 7, 5, 2) \) satisfy the preceding conditions, but a BIBD that has these parameters does not exist. Computational methods of BIBD generation usually suffer from combinatorial explosion, in part because of the large number of symmetries: for any solution, any two objects or blocks can be exchanged to obtain another solution.

This example demonstrates how to express a BIBD problem as a CSP and how to use lexicographic ordering constraints to break symmetries. Note that this example is for illustration only. SAS provides an autocall macro, MKTBIBD, for solving BIBD problems. The most direct CSP model for BIBD, as described in Meseguer and Torras (2001), represents a BIBD as a \( v \times b \) matrix \( X \). Each matrix entry is a Boolean decision variable \( X_{i,c} \) that satisfies \( X_{i,c} = 1 \) if and only if block \( c \) contains object \( i \). The condition that each object occurs in exactly \( r \) blocks (or, equivalently, that there are \( r \) 1s per row) can be expressed as \( v \) linear constraints:

\[
\sum_{c=1}^{b} X_{i,c} = r \quad \text{for} \quad i = 1, \ldots, v
\]

Alternatively, you can use global cardinality constraints to ensure that there are exactly \( b - r \) 0s and \( r \) 1s in \( X_{i,1}, \ldots, X_{i,b} \) for each object \( i \):

\[
gcc(X_{1,1}, \ldots, X_{v,b}) = ((0, 0, b - r)(1, 0, r)) \quad \text{for} \quad i = 1, \ldots, v
\]

Similarly, you can use the following constraints to specify the condition that each block contain exactly \( k \) objects (there are \( k \) 1s per column):

\[
gcc(X_{1,c}, \ldots, X_{v,c}) = ((0, 0, v - k)(1, 0, k)) \quad \text{for} \quad c = 1, \ldots, b
\]

To enforce the final condition that every two distinct objects occur together in exactly \( \lambda \) blocks (equivalently, that the scalar product of every pair of rows equal \( \lambda \)), you can introduce the auxiliary variables \( P_{i,j,c} \) for every \( i < j \), which indicate whether objects \( i \) and \( j \) both occur in block \( c \). The following reify constraint ensures that \( P_{i,j,c} = 1 \) if and only if block \( c \) contains both objects \( i \) and \( j \):

\[
\text{reify } P_{i,j,c} : (X_{i,c} + X_{j,c} = 2)
\]

The following constraints ensure that the final condition holds:

\[
gcc(P_{i,j,1}, \ldots, P_{i,j,b}) = ((0, 0, b - \lambda)(1, 0, \lambda)) \quad \text{for} \quad i = 1, \ldots, v - 1 \text{ and } j = i + 1, \ldots, v
\]

The objects and the blocks are interchangeable, so the matrix \( X \) has total row symmetry and total column symmetry. Because of the constraints on the rows, no pair of rows can be equal unless \( r = \lambda \). To break the row symmetry, you can impose strict lexicographic ordering on the rows of \( X \) as follows:

\[
(X_{i,1}, \ldots, X_{i,b}) \prec_{\text{lex}} (X_{i-1,1}, \ldots, X_{i-1,b}) \quad \text{for} \quad i = 2, \ldots, v
\]

To break the column symmetry, you can impose lexicographic ordering on the columns of \( X \) as follows:

\[
(X_{1,c}, \ldots, X_{v,c}) \preceq_{\text{lex}} (X_{1,c-1}, \ldots, X_{v,c-1}) \quad \text{for} \quad c = 2, \ldots, b
\]

The following SAS macro incorporates all the preceding constraints. For the specified parameters \( (v, b, r, k, \lambda) \), the macro either finds BIBDs or proves that a BIBD does not exist.
%macro bibd(v, b, r, k, lambda, out=bibdout);
/* Arrange v objects into b blocks such that:
   (i) each object occurs in exactly r blocks,
   (ii) each block contains exactly k objects,
   (iii) every pair of objects occur together in exactly lambda blocks.
Equivalently, create a binary matrix with v rows and b columns,
with r 1s per row, k 1s per column,
and scalar product lambda between any pair of distinct rows.
*/
/* Check necessary conditions */
%if (%eval(&r * &v) ne %eval(&b * &k)) or
   (%eval(&lambda * (&v - 1)) ne %eval(&r * (&k - 1))) or
   (&v > &b) %then %do;
   %put BIBD necessary conditions are not met.;
   %goto EXIT;
%end;
proc optmodel;
num v = &v;
num b = &b;
um r = &r;
um k = &k;
um lambda = &lambda;
set OBJECTS = 1..v;
set BLOCKS = 1..b;
/* Decision variable X[i,c] = 1 iff object i occurs in block c. */
var X {OBJECTS, BLOCKS} binary;
/* Mandatory constraints: */
/* (i) Each object occurs in exactly r blocks. */
con Exactly_r_blocks {i in OBJECTS}:
gcc({c in BLOCKS} X[i,c], {<0,0,b-r>,<1,0,r>});
/* (ii) Each block contains exactly k objects. */
con Exactly_k_objects {c in BLOCKS}:
gcc({i in OBJECTS} X[i,c], {<0,0,v-k>,<1,0,k>});
/* (iii) Every pair of objects occurs in exactly lambda blocks. */
set PAIRS = {i in OBJECTS, j in OBJECTS: i < j};
/* auxiliary variable P[i,j,c] = 1 iff both i and j occur in c */
var P {PAIRS, BLOCKS} binary;
con Pairs_reify {<i,j> in PAIRS, c in BLOCKS}:
   reify(P[i,j,c], X[i,c] + X[j,c] = 2);
con Pairs_gcc {<i,j> in PAIRS}:
   gcc({c in BLOCKS} P[i,j,c], {<0,0,b-lambda>,<1,0,lambda>});
/* symmetry-breaking constraints: */
/* Break row symmetry via lexicographic ordering constraints. */
con Symmetry_i {i in OBJECTS diff {1}}:
   lexico({c in BLOCKS} X[i,c] < {c in BLOCKS} X[i-1,c]);
/* Break column symmetry via lexicographic ordering constraints. */
con Symmetry_c (c in BLOCKS diff {1}):
  lexico((i in OBJECTS) X[i,c] <= (i in OBJECTS) X[i,c-1]);

solve with CLP / varselect=FIFO;
create data &out from
  {i in OBJECTS, c in BLOCKS} <col('X'||i||'_'||c)=X[i,c]>;
quit;
%put &_oroptmodel_;
%EXIT:
%mend bibd;

The following statement invokes the macro to find a BIBD design for the parameters (15, 15, 7, 7, 3):

%bibd(15,15,7,7,3);

The output is displayed in Output 11.8.1.
Output 11.8.1 Balanced Incomplete Block Design for (15,15,7,7,3)

Balanced Incomplete Block Design Problem
(15, 15, 7, 7, 3)

<table>
<thead>
<tr>
<th>Obs</th>
<th>Block1</th>
<th>Block2</th>
<th>Block3</th>
<th>Block4</th>
<th>Block5</th>
<th>Block6</th>
<th>Block7</th>
<th>Block8</th>
<th>Block9</th>
<th>Block10</th>
<th>Block11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obs</th>
<th>Block12</th>
<th>Block13</th>
<th>Block14</th>
<th>Block15</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Example 11.9: Progressive Party Problem

This example demonstrates the use of the pack constraint to solve an instance of the progressive party problem (Smith et al. 1996). In the original progressive party problem, a number of yacht crews and their boats congregate at a yachting rally. In order for each crew to socialize with as many other crews as possible, some of the boats are selected to serve as “host boats” for six rounds of parties. The crews of the host boats stay with their boats for all six rounds. The crews of the remaining boats, called “guest crews,” are assigned to visit a different host boat in each round.

Given the number of boats at the rally, the capacity of each boat, and the size of each crew, the objective of the original problem is to assign all the guest crews to host boats for each of the six rounds, using the minimum number of host boats. The partitioning of crews into guests and hosts is fixed throughout all rounds. No two crews should meet more than once. The assignments are constrained by the spare capacities (total capacity minus crew size) of the host boats and the crew sizes of the guest boats. Some boats cannot be hosts (zero spare capacity), and other boats must be hosts.

In this instance of the problem, the designation of the minimum requirement of 13 hosts is assumed (boats 1 through 12 and 14). The formulation solves up to eight rounds, but only two rounds are scheduled for this example. The total capacities and crew sizes of the boats are shown in Figure 11.3.

Figure 11.3 Progressive Party Problem Input

<table>
<thead>
<tr>
<th>boatnum</th>
<th>capacity</th>
<th>crewsize</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>12</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>17</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>19</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>8</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>boatnum</th>
<th>capacity</th>
<th>crewsize</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>23</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>24</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>25</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>26</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>27</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>29</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>30</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>31</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>33</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>34</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>35</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>36</td>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>37</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>38</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>39</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>41</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>42</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>
The following statements and DATA steps process the data and designate host boats:

```latex
\begin{verbatim}
data hostability;
  set capacities;
  spareCapacity = capacity - crewsize;
run;

data hosts guests;
  set hostability;
  if (boatnum <= 12 or boatnum eq 14) then do;
    output hosts;
  end;
  else do;
    output guests;
  end;
run;

/* sort so guest boats with larger crews appear first */
proc sort data=guests;
  by descending crewsize;
run;

data capacities;
  format boatnum capacity 2.;
  set hosts guests;
  seqno = _n_; 
run;
\end{verbatim}
```

To model the progressive party problem for the CLP solver, first define the following sets of variables:

- **Item variables** $x_{it}$ contain the host boat number for the assignment of guest boat $i$ in round $t$.
- **Load variables** $L_{ht}$ contain the load of host boat $h$ in round $t$.
- **Variable** $m_{ijt}$ are binary variables that take a value of 1 if and only if guest boats $i$ and $j$ are assigned to the same host boat in round $t$.

Next, describe the set of constraints that are used in the model:

- **All-different constraints** ensure that a guest boat is not assigned to the same host boat in different rounds.
- **Reify constraints** regulate the values that are assigned to the aforementioned indicator variables $m_{ijt}$.
- **The reified indicator variables appear in linear constraints to enforce the requirement to meet no more than once.**
- **One pack constraint per round** maintains the capacity limits of the host boats.
- **Finally, a symmetry-breaking linear constraint orders the host boat assignments for the highest-numbered guest boat across rounds.**

The following statements call PROC OPTMODEL to define the variables, specify the constraints, and solve the problem:
%let rounds=2;
%let numhosts=13;
proc optmodel;
  num numrounds = &rounds;
  set ROUNDS = 1..numrounds;
  num numhosts = &numhosts;
  set HOSTS = 1..numhosts;
  set BOATS;
  num numboats = card(BOATS);
  num capacity {BOATS};
  num crewsize {BOATS};
  num spareCapacity{hi in HOSTS} = capacity[hi] - crewsize[hi];
  /* Use the descending crew order for guests (seqno)
   rather than the actual boat id (boatnum)
   to help the performance of the PACK predicate. */
  read data capacities into BOATS=[seqno] capacity crewsize;
  /* Assume that the first numhosts boats are hosts,
   and process each round in turn.
   X is the host assigned to non-host i for round t. */
  var X {numhosts+1..numboats, ROUNDS} integer >= 1 <= numhosts;
  /* The load of the host boat. */
  var L {hi in HOSTS, ROUNDS} integer >= 0 <= spareCapacity[hi];
  /* Assign different hosts each round. */
  con AlldiffCon {i in numhosts+1..numboats}:
    alldiff({t in ROUNDS} X[i,t]);
  /* Two crews cannot meet more than once. */
  var M {i in numhosts+1..numboats-1, j in i+1..numboats, t in ROUNDS} binary;
  con ReifyCon {i in numhosts+1..numboats-1, j in i+1..numboats, t in ROUNDS}:
    reify(M[i,j,t], X[i,t] = X[j,t]);
  con Assign {i in numhosts+1..numboats-1, j in i+1..numboats}:
    sum {t in ROUNDS} M[i,j,t] <= 1;
  /* Honor capacities. */
  con PackCon {t in ROUNDS}:
    pack(
      {i in numhosts+1..numboats} X[i,t],
      {i in numhosts+1..numboats} crewsize[i],
      {h in HOSTS} L[h,t]
    );
  /* Break symmetries. */
  con SymmetryCon {t in 1..numrounds-1}:
    X[numboats,t] < X[numboats,t+1];
  solve with CLP / varselect=FIFO;
quit;

The two charts in Output 11.9.1 show the boat assignments for the first two rounds. The horizontal axis shows the load for each host boat. Slack capacity is highlighted in red.
Output 11.9.1  Gantt Chart: Boat Schedule by Round

**Schedule for Round 1**

<table>
<thead>
<tr>
<th>Host</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>28</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>41</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>21</td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>24</td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>8</td>
<td>27</td>
<td></td>
<td></td>
<td>30</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>9</td>
<td></td>
<td>37</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td>17</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The charts in Output 11.9.2 break down the assignments by boat number for selected boats.

### Output 11.9.1 continued

**Schedule for Round 2**

<table>
<thead>
<tr>
<th>Host</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>3</td>
<td></td>
<td>16</td>
<td></td>
<td></td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
<td>39</td>
<td></td>
<td></td>
<td></td>
<td>41</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>5</td>
<td></td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>6</td>
<td>38</td>
<td></td>
<td></td>
<td>17</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td>7</td>
<td></td>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td>27</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>8</td>
<td>23</td>
<td></td>
<td>18</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>9</td>
<td>24</td>
<td></td>
<td></td>
<td>30</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td></td>
<td>37</td>
<td></td>
<td>29</td>
<td>33</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>11</td>
<td></td>
<td>42</td>
<td></td>
<td>34</td>
<td>35</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td></td>
<td>20</td>
<td>31</td>
<td>36</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>14</td>
<td></td>
<td>25</td>
<td>32</td>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Output 11.9.2 Gantt Chart: Host Boat Schedule by Round

**Schedule for Boat 36**

<table>
<thead>
<tr>
<th>Round</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>14</td>
<td>35</td>
<td>36</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>20</td>
<td>31</td>
<td>36</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Schedule for Boat 38

<table>
<thead>
<tr>
<th>Round</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>5</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>41</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Schedule for Boat 40

<table>
<thead>
<tr>
<th>Round</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>12</td>
<td>33</td>
<td>34</td>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>14</td>
<td>25</td>
<td>32</td>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
References


Chapter 12
The Linear Programming Solver

Contents

Overview: LP Solver ................................................................. 502
Getting Started: LP Solver ....................................................... 502
Syntax: LP Solver ................................................................. 504
  Functional Summary ......................................................... 504
  LP Solver Options .......................................................... 505
Details: LP Solver ............................................................... 510
  Presolve ............................................................................ 510
  Pricing Strategies for the Primal and Dual Simplex Algorithms .... 511
  The Network Simplex Algorithm ......................................... 511
  The Interior Point Algorithm ............................................. 512
  Iteration Log for the Primal and Dual Simplex Algorithms ....... 514
  Iteration Log for the Network Simplex Algorithm .................. 514
  Iteration Log for the Interior Point Algorithm ...................... 515
  Iteration Log for the Crossover Algorithm ......................... 516
  Concurrent LP .................................................................. 516
  Parallel Processing ........................................................... 517
  Problem Statistics ............................................................ 517
  Variable and Constraint Status .......................................... 518
  Irreducible Infeasible Set .................................................. 519
  Macro Variable _OROPTMODEL_ ........................................ 520
Examples: LP Solver ............................................................. 522
  Example 12.1: Diet Problem ............................................... 522
  Example 12.2: Reoptimizing the Diet Problem Using BASIS=WARMSTART .... 525
  Example 12.3: Two-Person Zero-Sum Game ......................... 533
  Example 12.4: Finding an Irreducible Infeasible Set ............... 536
  Example 12.5: Using the Network Simplex Algorithm ............. 539
  Example 12.6: Migration to OPTMODEL: Generalized Networks ...... 547
  Example 12.7: Migration to OPTMODEL: Maximum Flow ........... 551
  Example 12.8: Migration to OPTMODEL: Production, Inventory, Distribution .... 554
  Example 12.9: Migration to OPTMODEL: Shortest Path .............. 562
References ........................................................................ 565
Overview: LP Solver

The OPTMODEL procedure provides a framework for specifying and solving linear programs (LPs). A standard linear program has the following formulation:

\[
\begin{align*}
\text{min} & \quad c^T x \\
\text{subject to} & \quad Ax \{\geq, =, \leq\} b \\
& \quad 1 \leq x \leq u
\end{align*}
\]

where

- \( x \in \mathbb{R}^n \) is the vector of decision variables
- \( A \in \mathbb{R}^{m \times n} \) is the matrix of constraints
- \( c \in \mathbb{R}^n \) is the vector of objective function coefficients
- \( b \in \mathbb{R}^m \) is the vector of constraints’ right-hand sides (RHS)
- \( l \in \mathbb{R}^n \) is the vector of lower bounds on variables
- \( u \in \mathbb{R}^n \) is the vector of upper bounds on variables

The following LP algorithms are available in the OPTMODEL procedure:

- primal simplex algorithm
- dual simplex algorithm
- network simplex algorithm
- interior point algorithm

The primal and dual simplex algorithms implement the two-phase simplex method. In phase I, the algorithm tries to find a feasible solution. If no feasible solution is found, the LP is infeasible; otherwise, the algorithm enters phase II to solve the original LP. The network simplex algorithm extracts a network substructure, solves this using network simplex, and then constructs an advanced basis to feed to either primal or dual simplex. The interior point algorithm implements a primal-dual predictor-corrector interior point algorithm. If any of the decision variables are constrained to be integer-valued, then the relaxed version of the problem is solved.

Getting Started: LP Solver

The following example illustrates how you can use the OPTMODEL procedure to solve linear programs. Suppose you want to solve the following problem:

\[
\begin{align*}
\text{max} & \quad x_1 + x_2 + x_3 \\
\text{subject to} & \quad 3x_1 + 2x_2 - x_3 \leq 1 \\
& \quad -2x_1 - 3x_2 + 2x_3 \leq 1 \\
& \quad x_1, \ x_2, \ x_3 \geq 0
\end{align*}
\]
You can use the following statements to call the OPTMODEL procedure for solving linear programs:

```plaintext
proc optmodel;
    var x{i in 1..3} >= 0;
    max f = x[1] + x[2] + x[3];
    solve with lp / algorithm = ps presolver = none logfreq = 1;
    print x;
quit;
```

The optimal solution and the optimal objective value are displayed in Figure 12.1.

**Figure 12.1 Solution Summary**

**The OPTMODEL Procedure**

<table>
<thead>
<tr>
<th>Problem Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
<td>Maximization</td>
</tr>
<tr>
<td>Objective Function</td>
<td>f</td>
</tr>
<tr>
<td>Objective Type</td>
<td>Linear</td>
</tr>
</tbody>
</table>

| Number of Variables | 3 |
| Bounded Above | 0 |
| Bounded Below | 3 |
| Bounded Below and Above | 0 |
| Free | 0 |
| Fixed | 0 |

| Number of Constraints | 2 |
| Linear LE (<=) | 2 |
| Linear EQ (=) | 0 |
| Linear GE (>=) | 0 |
| Linear Range | 0 |

| Constraint Coefficients | 6 |

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
</tr>
<tr>
<td>Dual Infeasibility</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>
The iteration log displaying problem statistics, progress of the solution, and the optimal objective value is shown in Figure 12.2.

**Figure 12.1 continued**

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

**Figure 12.2 Log**

NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 2 linear constraints (2 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 6 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Primal Simplex algorithm is used.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Iteration</th>
<th>Objective Value</th>
<th>Time</th>
<th>Entering Variable</th>
<th>Leaving Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>P 2</td>
<td>1</td>
<td>0.0000000E+00</td>
<td>0</td>
<td>x[3]</td>
<td>c2 (S)</td>
</tr>
<tr>
<td>P 2</td>
<td>2</td>
<td>5.0000000E-01</td>
<td>0</td>
<td>x[2]</td>
<td>c1 (S)</td>
</tr>
<tr>
<td>P 2</td>
<td>3</td>
<td>8.0000000E+00</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Optimal.
NOTE: Objective = 8.
NOTE: The Primal Simplex solve time is 0.01 seconds.

**Syntax: LP Solver**

The following statement is available in the OPTMODEL procedure:

```
SOLVE WITH LP < / options > ;
```

**Functional Summary**

Table 12.1 summarizes the list of options available for the SOLVE WITH LP statement, classified by function.

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solver Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the type of algorithm</td>
<td>ALGORITHM=</td>
</tr>
<tr>
<td>Specifies the type of algorithm called after network simplex</td>
<td>ALGORITHM2=</td>
</tr>
</tbody>
</table>
### Table 12.1 (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enables or disables IIS detection</td>
<td>IIS=</td>
</tr>
<tr>
<td><strong>Presolve Option</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the type of presolve</td>
<td>PRESOLVER=</td>
</tr>
<tr>
<td>Controls the dualization of the problem</td>
<td>DUALIZE=</td>
</tr>
<tr>
<td><strong>Control Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the feasibility tolerance</td>
<td>FEASTOL=</td>
</tr>
<tr>
<td>Specifies the frequency of printing solution progress</td>
<td>LOGFREQ=</td>
</tr>
<tr>
<td>Specifies the detail of solution progress printed in log</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the maximum number of iterations</td>
<td>MAXITER=</td>
</tr>
<tr>
<td>Specifies the time limit for the optimization process</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Specifies the optimality tolerance</td>
<td>OPTTOL=</td>
</tr>
<tr>
<td>Specifies units of CPU time or real time</td>
<td>TIMETYPE=</td>
</tr>
<tr>
<td><strong>Simplex Algorithm Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the type of initial basis</td>
<td>BASIS=</td>
</tr>
<tr>
<td>Specifies the type of pricing strategy</td>
<td>PRICETYPE=</td>
</tr>
<tr>
<td>Specifies the queue size for determining entering variable</td>
<td>QUEUESIZE=</td>
</tr>
<tr>
<td>Enables or disables scaling of the problem</td>
<td>SCALE=</td>
</tr>
<tr>
<td>Specifies the initial seed for the random number generator</td>
<td>SEED=</td>
</tr>
<tr>
<td><strong>Interior Point Algorithm Options</strong></td>
<td></td>
</tr>
<tr>
<td>Enables or disables interior crossover</td>
<td>CROSSOVER=</td>
</tr>
<tr>
<td>Specifies the stopping criterion based on duality gap</td>
<td>DUALITYGAP=</td>
</tr>
<tr>
<td><strong>Decomposition Algorithm Options</strong></td>
<td></td>
</tr>
<tr>
<td>Enables decomposition algorithm and specifies general control options</td>
<td>DECOMP=()</td>
</tr>
<tr>
<td>Specifies options for the master problem</td>
<td>DECOMPMASTER=()</td>
</tr>
<tr>
<td>Specifies options for the subproblem</td>
<td>DECOMPSUBPROB=()</td>
</tr>
<tr>
<td><strong>Parallel Options</strong></td>
<td></td>
</tr>
<tr>
<td>Enables the LP solver to run deterministically</td>
<td>DETERMINISTIC=</td>
</tr>
<tr>
<td>Specifies the number of threads for the parallel LP solver to use</td>
<td>NTHREADS=</td>
</tr>
</tbody>
</table>
Solver Options

**ALGORITHM=option**

**SOLVER=option**

**SOL=option**

specifies an LP algorithm. You can specify the following options:

- **PRIMAL (PS)** uses the primal simplex algorithm.
- **DUAL (DS)** uses the dual simplex algorithm.
- **NETWORK (NS)** uses the network simplex algorithm.
- **INTERIORPOINT (IP)** uses the interior point algorithm.
- **CONCURRENT (CON)** uses several different algorithms in parallel.

The valid abbreviated value for each option is indicated in parentheses. By default, ALGORITHM=DUAL.

**ALGORITHM2=option**

**SOLVER2=option**

specifies an LP algorithm if ALGORITHM=NS. You can specify the following options:

- **PRIMAL (PS)** uses the primal simplex algorithm (after network simplex).
- **DUAL (DS)** uses the dual simplex algorithm (after network simplex).

The valid abbreviated value for each option is indicated in parentheses. By default, the LP solver decides which algorithm is best to use after calling the network simplex algorithm on the extracted network.

**IIS=FALSE | TRUE**

specifies whether the LP solver attempts to identify a set of constraints and variables that form an irreducible infeasible set (IIS). You can specify the following values:

- **FALSE** disables IIS detection.
- **TRUE** enables IIS detection.

If an IIS is found, information about the infeasibilities can be found in the .status values of the constraints and variables. By default, IIS=FALSE. For more information about the IIS= option, see the section “Irreducible Infeasible Set” on page 519. For more information about the .status suffix, see the section “Suffixes” on page 380.

Presolve Options

**DUALIZE=AUTOMATIC | OFF | ON**

controls the dualization of the problem. You can specify the following values:

- **AUTOMATIC** the presolver uses a heuristic to decide whether to dualize the problem or not.
- **OFF** disables dualization. The optimization problem is solved in the form that you specify.
- **ON** the presolver formulates the dual of the linear optimization problem.
Dualization is usually helpful for problems that have many more constraints than variables. You can use this option with all simplex algorithms in the SOLVE WITH LP statement, but it is most effective with the primal and dual simplex algorithms.

By default DUALIZE=AUTOMATIC.

**PRESOLVER=**AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE

specifies the presolve option. You can specify the following values:

- **AUTOMATIC** applies the presolver by using default settings.
- **NONE** disables the presolver.
- **BASIC** performs a minimal presolve, such as removing empty rows, columns, and fixed variables.
- **MODERATE** performs a basic presolve and applies other inexpensive presolve techniques.
- **AGGRESSIVE** performs a moderate presolve and applies other aggressive (but computationally expensive) presolve techniques.

By default, PRESOLVER=AUTOMATIC. For more information, see the section “Presolve” on page 510.

**Control Options**

**FEASTOL=**\(\epsilon\)

specifies the feasibility tolerance, \(\epsilon \in [10^{-9}, 10^{-4}]\), for determining the feasibility of a variable. The default value is \(10^{-6}\). To compute the feasibility tolerance, simplex algorithms use the absolute error, and interior point algorithms use the relative error.

**LOGFREQ=**\(k\)

**PRINTFREQ=**\(k\)

specifies that the printing of the solution progress to the iteration log is to occur after every \(k\) iterations. The print frequency, \(k\), is an integer between zero and the largest four-byte signed integer, which is \(2^{31} - 1\).

The value \(k = 0\) disables the printing of the progress of the solution. If the primal or dual simplex algorithms are used, the default value of this option is determined dynamically according to the problem size. If the network simplex algorithm is used, the default value of this option is 10,000. If the interior point algorithm is used, the default value of this option is 1.

**LOGLEVEL=**\(string\)

controls the amount of information displayed in the SAS log by the LP solver, from a short description of presolve information and summary to details at each iteration. The valid values for this option are described below.

- **NONE** turns off all solver-related messages to SAS log.
- **BASIC** displays a solver summary after stopping.
- **MODERATE** prints a solver summary and an iteration log by using the interval dictated by the LOGFREQ= option.
Chapter 12: The Linear Programming Solver

AGGRESSIVE prints a detailed solver summary and an iteration log by using the interval dictated by the LOGFREQ= option.

The default value is MODERATE.

MAXITER= \( k \) specifies the maximum number of iterations. The value \( k \) can be any integer between one and the largest four-byte signed integer, which is \( 2^{31} - 1 \). If you do not specify this option, the procedure does not stop based on the number of iterations performed. For network simplex, this iteration limit corresponds to the algorithm called after network simplex (either primal or dual simplex).

MAXTIME= \( t \) specifies an upper limit of \( t \) units of time for the optimization process, including problem generation time and solution time. The value of the TIMETYPE= option determines the type of units used. If you do not specify the MAXTIME= option, the solver does not stop based on the amount of time elapsed. The value of \( t \) can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

OPTTOL= \( \epsilon \) specifies the optimality tolerance, \( \epsilon \in [1\text{E}^{-9}, 1\text{E}^{-4}] \), for declaring optimality. The default value is \( 1\text{E}^{-6} \). Simplex algorithms use the absolute error and interior point algorithms use the relative error for the computation of optimality tolerance.

TIMETYPE=CPU | REAL specifies the units of time used by the MAXTIME= option and reported by the PRESOLVE_TIME and SOLUTION_TIME terms in the _OROPTMODEL_ macro variable. The valid values of the TIMETYPE= option are described below.

- **CPU** specifies units of CPU time.
- **REAL** specifies units of real time.

The “Optimization Statistics” table, an output of the OPTMODEL procedure if you specify PRINT-LEVEL=2 in the PROC OPTMODEL statement, also includes the same time units for Presolver Time and Solver Time. The other times (such as Problem Generation Time) in the “Optimization Statistics” table are also in the same units.

By default, TIMETYPE=REAL.

### Simplex Algorithm Options

**BASIS=CRASH | SLACK | WARMSTART** specifies the options for generating an initial basis. You can specify the following values:

- **CRASH** generates an initial basis by using crash techniques (Maros 2003). The procedure creates a triangular basic matrix consisting of both decision variables and slack variables.
- **SLACK** generates an initial basis by using all slack variables.
- **WARMSTART** starts the primal and dual simplex algorithms with the available basis.

The default option is determined automatically based on the problem structure. For the network simplex algorithm, this option has no effect.
PRICETYPE=HYBRID | PARTIAL | FULL | DEVE  | STEEPESTEDGE

specifies the pricing strategy for the primal and dual simplex algorithms. You can specify the following values:

- **HYBRID** uses hybrid Devex and steepest-edge pricing strategies. This strategy is available for the primal simplex algorithm only.
- **PARTIAL** uses partial pricing strategy. Optionally, you can specify QUEUESIZE=. This strategy is available for the primal simplex algorithm only.
- **FULL** uses the most negative reduced-cost pricing strategy.
- **DEVE** uses the Devex pricing strategy.
- **STEEPESTEDGE** uses the steepest-edge pricing strategy.

The default option is determined automatically based on the problem structure. For the network simplex algorithm, this option applies only to the algorithm specified by the ALGORITHM2= option. For more information, see the section “Pricing Strategies for the Primal and Dual Simplex Algorithms” on page 511.

QUEUESIZE=k

specifies the queue size, $k \in [1, n]$, where $n$ is the number of decision variables. This queue is used for finding an entering variable in the simplex iteration. The default value is chosen adaptively based on the number of decision variables. This option is used only when PRICETYPE=PARTIAL.

SCALE=None | AUTOMATIC

specifies the scaling option. You can specify the following values:

- **NONE** disables scaling.
- **AUTOMATIC** automatically applies scaling procedure if necessary.

By default, SCALE=AUTOMATIC.

SEED=number

specifies the initial seed for the random number generator. Because the seed affects the perturbation in the simplex algorithms, the result might be a different optimal solution and a different solver path, but the effect is usually negligible. The value of number can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. By default, SEED=100.

**Interior Point Algorithm Options**

CROSSOVER=FALSE | TRUE

specifies whether to convert the interior point solution to a basic simplex solution. You can specify the following values:

- **FALSE** disables crossover.
- **TRUE** applies the crossover algorithm to the interior point solution.

If the interior point algorithm terminates with a solution, the crossover algorithm uses the interior point solution to create an initial basic solution. After performing primal fixing and dual fixing, the crossover algorithm calls a simplex algorithm to locate an optimal basic solution. By default, CROSSOVER=TRUE.
**DUALITYGAP=δ**

specifies the desired relative duality gap, $\delta \in [1E-9, 1E-4]$. This is the relative difference between the primal and dual objective function values and is the primary solution quality parameter. The default value is $1E-6$. For more information, see the section “The Interior Point Algorithm” on page 512.

**Parallel Options**

**DETERMINISTIC=TRUE | FALSE**

specifies whether to run the LP solver in deterministic parallel or nondeterministic parallel mode. You can specify the following values:

- **TRUE** runs in deterministic parallel mode.
- **FALSE** runs in nondeterministic parallel mode.

By default, DETERMINISTIC=TRUE.

**NTHREADS=number**

specifies the maximum number of threads for the LP solver to use for multithreaded processing. The value of *number* can be any integer between 1 and 256, inclusive. The default is the value of the NTHREADS= option in PROC OPTMODEL.

**Decomposition Algorithm Options**

The following options are available for the decomposition algorithm in the LP solver. For information about the decomposition algorithm, see Chapter 18, “The Decomposition Algorithm.”

**DECOMP=(options)**

enables the decomposition algorithm and specifies overall control options for the algorithm. For more information about this option, see Chapter 18, “The Decomposition Algorithm.”

**DECOMPMASTER=(options)**

specifies options for the master problem. For more information about this option, see Chapter 18, “The Decomposition Algorithm.”

**DECOMPSUBPROB=(options)**

specifies option for the subproblem. For more information about this option, see Chapter 18, “The Decomposition Algorithm.”

---

**Details: LP Solver**

**Presolve**

Presolve in the simplex LP algorithms of PROC OPTMODEL uses a variety of techniques to reduce the problem size, improve numerical stability, and detect infeasibility or unboundedness (Andersen and Andersen 1995; Gondzio 1997). During presolve, redundant constraints and variables are identified and removed.
Presolve can further reduce the problem size by substituting variables. Variable substitution is a very effective technique, but it might occasionally increase the number of nonzero entries in the constraint matrix.

In most cases, using presolve is very helpful in reducing solution times. You can enable presolve at different levels or disable it by specifying the PRESOLVER= option.

---

### Pricing Strategies for the Primal and Dual Simplex Algorithms

Several pricing strategies for the primal and dual simplex algorithms are available. Pricing strategies determine which variable enters the basis at each simplex pivot. These can be controlled by specifying the PRICETYPE= option.

The primal simplex algorithm has the following five pricing strategies:

- **PARTIAL**: scans a queue of decision variables to find an entering variable. You can optionally specify the QUEUESIZE= option to control the length of this queue.
- **FULL**: uses Dantzig’s most violated reduced cost rule (Dantzig 1963). It compares the reduced cost of all decision variables, and selects the variable with the most violated reduced cost as the entering variable.
- **DEVEX**: implements the Devex pricing strategy developed by Harris (1973).
- **STEEPESTEDGE**: uses the steepest-edge pricing strategy developed by Forrest and Goldfarb (1992).
- **HYBRID**: uses a hybrid of the Devex and steepest-edge pricing strategies.

The dual simplex algorithm has only three pricing strategies available: FULL, DEVEX, and STEEPEST-EDGE.

---

### The Network Simplex Algorithm

The network simplex algorithm in PROC OPTMODEL attempts to leverage the speed of the network simplex algorithm to more efficiently solve linear programs by using the following process:

1. It heuristically extracts the largest possible network substructure from the original problem.
2. It uses the network simplex algorithm to solve for an optimal solution to this substructure.
3. It uses this solution to construct an advanced basis to warm-start either the primal or dual simplex algorithm on the original linear programming problem.

The network simplex algorithm is a specialized version of the simplex algorithm that uses spanning-tree bases to more efficiently solve linear programming problems that have a pure network form. Such LPs can be modeled using a formulation over a directed graph, as a minimum-cost flow problem. Let $G = (N, A)$ be a directed graph, where $N$ denotes the nodes and $A$ denotes the arcs of the graph. The decision variable $x_{ij}$ denotes the amount of flow sent from node $i$ to node $j$. The cost per unit of flow on the arcs is designated by $c_{ij}$, and the amount of flow sent across each arc is bounded to be within $[l_{ij}, u_{ij}]$. The demand (or supply) at
each node is designated as \( b_i \), where \( b_i > 0 \) denotes a supply node and \( b_i < 0 \) denotes a demand node. The corresponding linear programming problem is as follows:

\[
\begin{align*}
\text{min} & \quad \sum_{(i,j) \in A} c_{ij} x_{ij} \\
\text{subject to} & \quad \sum_{(i,j) \in A} x_{ij} - \sum_{(j,i) \in A} x_{ji} = b_i \quad \forall i \in N \\
& \quad x_{ij} \leq u_{ij} \quad \forall (i, j) \in A \\
& \quad x_{ij} \geq l_{ij} \quad \forall (i, j) \in A.
\end{align*}
\]

The network simplex algorithm used in PROC OPTMODEL is the primal network simplex algorithm. This algorithm finds the optimal primal feasible solution and a dual solution that satisfies complementary slackness. Sometimes the directed graph \( G \) is disconnected. In this case, the problem can be decomposed into its weakly connected components, and each minimum-cost flow problem can be solved separately. After solving each component, the optimal basis for the network substructure is augmented with the non-network variables and constraints from the original problem. This advanced basis is then used as a starting point for the primal or dual simplex method. The solver automatically selects the algorithm to use after network simplex. However, you can override this selection with the \texttt{ALGORITHM2=} option.

The network simplex algorithm can be more efficient than the other algorithms on problems that have a large network substructure. The size of this network structure can be seen in the log.

---

**The Interior Point Algorithm**

The interior point LP algorithm in PROC OPTMODEL implements an infeasible primal-dual predictor-corrector interior point algorithm. To illustrate the algorithm and the concepts of duality and dual infeasibility, consider the following LP formulation (the primal):

\[
\begin{align*}
\text{min} & \quad c^T x \\
\text{subject to} & \quad Ax \geq b \\
& \quad x \geq 0
\end{align*}
\]

The corresponding dual is as follows:

\[
\begin{align*}
\text{max} & \quad b^T y \\
\text{subject to} & \quad A^T y + w = c \\
& \quad y \geq 0 \\
& \quad w \geq 0
\end{align*}
\]

where \( y \in \mathbb{R}^m \) refers to the vector of dual variables and \( w \in \mathbb{R}^n \) refers to the vector of dual slack variables.

The dual makes an important contribution to the certificate of optimality for the primal. The primal and dual constraints combined with complementarity conditions define the first-order optimality conditions, also known as KKT (Karush-Kuhn-Tucker) conditions, which can be stated as follows:

\[
\begin{align*}
Ax - s &= b & (\text{Primal Feasibility}) \\
A^T y + w &= c & (\text{Dual Feasibility}) \\
WXe &= 0 & (\text{Complementarity}) \\
SYe &= 0 & (\text{Complementarity}) \\
x, y, w, s &\geq 0
\end{align*}
\]

where \( e \equiv (1, \ldots, 1)^T \) of appropriate dimension and \( s \in \mathbb{R}^m \) is the vector of primal slack variables.
NOTE: Slack variables (the \(s\) vector) are automatically introduced by the algorithm when necessary; it is therefore recommended that you not introduce any slack variables explicitly. This enables the algorithm to handle slack variables much more efficiently.

The letters \(X\), \(Y\), \(W\), and \(S\) denote matrices with corresponding \(x\), \(y\), \(w\), and \(s\) on the main diagonal and zero elsewhere, as in the following example:

\[
X = \begin{bmatrix}
  x_1 & 0 & \cdots & 0 \\
  0 & x_2 & \cdots & 0 \\
  \vdots & \vdots & \ddots & \vdots \\
  0 & 0 & \cdots & x_n
\end{bmatrix}
\]

If \((x^*, y^*, w^*, s^*)\) is a solution of the previously defined system of equations representing the KKT conditions, then \(x^*\) is also an optimal solution to the original LP model.

At each iteration the interior point algorithm solves a large, sparse system of linear equations as follows:

\[
\begin{bmatrix}
  Y^{-1}S & A \\
  A^T & -X^{-1}W
\end{bmatrix}
\begin{bmatrix}
  \Delta y \\
  \Delta x
\end{bmatrix}
= \begin{bmatrix}
  \Xi \\
  \Theta
\end{bmatrix}
\]

where \(\Delta x\) and \(\Delta y\) denote the vector of search directions in the primal and dual spaces, respectively; \(\Theta\) and \(\Xi\) constitute the vector of the right-hand sides.

The preceding system is known as the reduced KKT system. The interior point algorithm uses a preconditioned quasi-minimum residual algorithm to solve this system of equations efficiently.

An important feature of the interior point algorithm is that it takes full advantage of the sparsity in the constraint matrix, thereby enabling it to efficiently solve large-scale linear programs.

The interior point algorithm works simultaneously in the primal and dual spaces. It attains optimality when both primal and dual feasibility are achieved and when complementarity conditions hold. Therefore it is of interest to observe the following four measures:

- Relative primal infeasibility measure \(\alpha\):
  \[
  \alpha = \frac{\|Ax - b - s\|_2}{\|b\|_2 + 1}
  \]

- Relative dual infeasibility measure \(\beta\):
  \[
  \beta = \frac{\|c - A^T y - w\|_2}{\|c\|_2 + 1}
  \]

- Relative duality gap \(\delta\):
  \[
  \delta = \frac{|c^T x - b^T y|}{|c^T x| + 1}
  \]

- Absolute complementarity \(\gamma\):
  \[
  \gamma = \sum_{i=1}^{n} x_i w_i + \sum_{i=1}^{m} y_i s_i
  \]

where \(\|v\|_2\) is the Euclidean norm of the vector \(v\). These measures are displayed in the iteration log.
Iteration Log for the Primal and Dual Simplex Algorithms

The primal and dual simplex algorithms implement a two-phase simplex algorithm. Phase I finds a feasible solution, which phase II improves to an optimal solution.

When LOGFREQ=1, the following information is printed in the iteration log:

- **Algorithm** indicates which simplex method is running by printing the letter P (primal) or D (dual).
- **Phase** indicates whether the algorithm is in phase I or phase II of the simplex method.
- **Iteration** indicates the iteration number.
- **Objective Value** indicates the current amount of infeasibility in phase I and the primal objective value of the current solution in phase II.
- **Time** indicates the time elapsed (in seconds).
- **Entering Variable** indicates the entering pivot variable. A slack variable that enters the basis is indicated by the corresponding row name followed by “(S)”. If the entering nonbasic variable has distinct and finite lower and upper bounds, then a “bound swap” can take place in the primal simplex method.
- **Leaving Variable** indicates the leaving pivot variable. A slack variable that leaves the basis is indicated by the corresponding row name followed by “(S)”. The leaving variable is the same as the entering variable if a bound swap has taken place.

When you omit the LOGFREQ= option or specify a value larger than 1, only the algorithm, phase, iteration, objective value, and time information is printed in the iteration log.

The behavior of objective values in the iteration log depends on both the current phase and the chosen algorithm. In phase I, both simplex methods have artificial objective values that decrease to 0 when a feasible solution is found. For the dual simplex method, phase II maintains a dual feasible solution, so a minimization problem has increasing objective values in the iteration log. For the primal simplex method, phase II maintains a primal feasible solution, so a minimization problem has decreasing objective values in the iteration log.

During the solution process, some elements of the LP model might be perturbed to improve performance. In this case the objective values that are printed correspond to the perturbed problem. After reaching optimality for the perturbed problem, the LP solver solves the original problem by switching from the primal simplex method to the dual simplex method (or from the dual simplex method to the primal simplex method). Because the problem might be perturbed again, this process can result in several changes between the two algorithms.

Iteration Log for the Network Simplex Algorithm

After finding the embedded network and formulating the appropriate relaxation, the network simplex algorithm uses a primal network simplex algorithm. In the case of a connected network, with one (weakly connected) component, the log will show the progress of the simplex algorithm. The following information is displayed in the iteration log:
Iteration Log for the Interior Point Algorithm

Iteration indicates the iteration number.
PrimalObj indicates the primal objective value of the current solution.
Primal Infeas indicates the maximum primal infeasibility of the current solution.
Time indicates the time spent on the current component by network simplex.

The frequency of the simplex iteration log is controlled by the LOGFREQ= option. The default value of the LOGFREQ= option is 10,000.

If the network relaxation is disconnected, the information in the iteration log shows progress at the component level. The following information is displayed in the iteration log:

Component indicates the component number being processed.
Nodes indicates the number of nodes in this component.
Arcs indicates the number of arcs in this component.
Iterations indicates the number of simplex iterations needed to solve this component.
Time indicates the time spent so far in network simplex.

The frequency of the component iteration log is controlled by the LOGFREQ= option. In this case, the default value of the LOGFREQ= option is determined by the size of the network.

The LOGLEVEL= option adjusts the amount of detail shown. By default, LOGLEVEL=MODERATE and reports as in the preceding description. If LOGLEVEL=NONE, no information is shown. If LOGLEVEL=BASIC, the only information shown is a summary of the network relaxation and the time spent solving the relaxation. If LOGLEVEL=AGGRESSIVE, in the case of one component, the log displays as in the preceding description; in the case of multiple components, for each component, a separate simplex iteration log is displayed.

Iteration Log for the Interior Point Algorithm

The interior point algorithm implements an infeasible primal-dual predictor-corrector interior point algorithm. The following information is displayed in the iteration log:

Iter indicates the iteration number
Complement indicates the (absolute) complementarity
Duality Gap indicates the (relative) duality gap
Primal Infeas indicates the (relative) primal infeasibility measure
Bound Infeas indicates the (relative) bound infeasibility measure
Dual Infeas indicates the (relative) dual infeasibility measure
Time indicates the time elapsed (in seconds).

If the sequence of solutions converges to an optimal solution of the problem, you should see all columns in the iteration log converge to zero or very close to zero. If they do not, it can be the result of insufficient iterations being performed to reach optimality. In this case, you might need to increase the value specified in
the option MAXITER= or MAXTIME=. If the complementarity and/or the duality gap do not converge, the problem might be infeasible or unbounded. If the infeasibility columns do not converge, the problem might be infeasible.

---

**Iteration Log for the Crossover Algorithm**

The crossover algorithm takes an optimal solution from the interior point algorithm and transforms it into an optimal basic solution. The iterations of the crossover algorithm are similar to simplex iterations; this similarity is reflected in the format of the iteration logs.

When LOGFREQ=1, the following information is printed in the iteration log:

- **Phase** indicates whether the primal crossover (PC) or dual crossover (DC) technique is used.
- **Iteration** indicates the iteration number.
- **Objective Value** indicates the total amount by which the superbasic variables are off their bound. This value decreases to 0 as the crossover algorithm progresses.
- **Time** indicates the time elapsed (in seconds).
- **Entering Variable** indicates the entering pivot variable. A slack variable that enters the basis is indicated by the corresponding row name followed by “(S)”.
- **Leaving Variable** indicates the leaving pivot variable. A slack variable that leaves the basis is indicated by the corresponding row name followed by “(S)”.

When you omit the LOGFREQ= option or specify a value greater than 1, only the phase, iteration, objective value, and time information are printed in the iteration log.

After all the superbasic variables have been eliminated, the crossover algorithm continues with regular primal or dual simplex iterations.

---

**Concurrent LP**

The ALGORITHM=CON option starts several different linear optimization algorithms in parallel in a single-machine mode. The LP solver automatically determines which algorithms to run and how many threads to assign to each algorithm. If sufficient resources are available, the solver runs all four standard algorithms. When the first algorithm finishes, the LP solver returns the results from that algorithm and terminates any other algorithms that are still running. If you specify a value of TRUE for the DETERMINISTIC= option, the algorithm for which the results are returned is not necessarily the one that finished first. The LP solver deterministically selects the algorithm for which the results are returned. Regardless of which mode (deterministic or nondeterministic) is in effect, terminating algorithms that are still running might take a significant amount of time.

During concurrent optimization, the procedure displays the iteration log for the dual simplex algorithm. See the section “Iteration Log for the Primal and Dual Simplex Algorithms” on page 514 for more information about this iteration log. Upon termination, the solver displays the iteration log for the algorithm that finishes first, unless the dual simplex algorithm finishes first. If you specify LOGLEVEL=AGGRESSIVE, the LP solver displays the iteration logs for all algorithms that were run concurrently.
If you specify PRINTLEVEL=2 in the PROC OPTMODEL statement and ALGORITHM=CON in the SOLVE WITH LP statement, the LP solver produces an ODS table called ConcurrentSummary. This table contains a summary of the solution statuses of all algorithms that are run concurrently.

### Parallel Processing

The interior point, concurrent LP, and decomposition algorithms can be run in single-machine mode; in this mode, the computation is executed by multiple threads on a single computer.

### Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of $10^9$) can cause difficulty when the remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTMODEL procedure causes the ODS table “ProblemStatistics” to be generated when the LP solver is called. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Figure 12.3 demonstrates the contents of the ODS table “ProblemStatistics.”

**Figure 12.3 ODS Table ProblemStatistics**

**The OPTMODEL Procedure**

<table>
<thead>
<tr>
<th>Problem Statistics</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Constraint Matrix Nonzeros</td>
<td>6</td>
</tr>
<tr>
<td>Maximum Constraint Matrix Coefficient</td>
<td>3</td>
</tr>
<tr>
<td>Minimum Constraint Matrix Coefficient</td>
<td>1</td>
</tr>
<tr>
<td>Average Constraint Matrix Coefficient</td>
<td>2.1666666667</td>
</tr>
<tr>
<td>Number of Objective Nonzeros</td>
<td>3</td>
</tr>
<tr>
<td>Maximum Objective Coefficient</td>
<td>1</td>
</tr>
<tr>
<td>Minimum Objective Coefficient</td>
<td>1</td>
</tr>
<tr>
<td>Average Objective Coefficient</td>
<td>1</td>
</tr>
<tr>
<td>Number of RHS Nonzeros</td>
<td>2</td>
</tr>
<tr>
<td>Maximum RHS</td>
<td>1</td>
</tr>
<tr>
<td>Minimum RHS</td>
<td>1</td>
</tr>
<tr>
<td>Average RHS</td>
<td>1</td>
</tr>
<tr>
<td>Maximum Number of Nonzeros per Column</td>
<td>2</td>
</tr>
<tr>
<td>Minimum Number of Nonzeros per Column</td>
<td>2</td>
</tr>
<tr>
<td>Average Number of Nonzeros per Column</td>
<td>2</td>
</tr>
<tr>
<td>Maximum Number of Nonzeros per Row</td>
<td>3</td>
</tr>
<tr>
<td>Minimum Number of Nonzeros per Row</td>
<td>3</td>
</tr>
<tr>
<td>Average Number of Nonzeros per Row</td>
<td>3</td>
</tr>
</tbody>
</table>
Variable and Constraint Status

Upon termination of the LP solver, the .status suffix of each decision variable and constraint stores information about the status of that variable or constraint. For more information about suffixes in the OPTMODEL procedure, see the section “Suffixes” on page 380.

Variable Status

The .status suffix of a decision variable specifies the status of that decision variable. The suffix can take one of the following values:

- B  basic variable
- L  nonbasic variable at its lower bound
- U  nonbasic variable at its upper bound
- F  free variable
- A  superbasic variable (a nonbasic variable that has a value strictly between its bounds)
- I  LP model infeasible (all decision variables have .status equal to I)

For the interior point algorithm with IIS=FALSE, .status is blank.

The following values can appear only if IIS=TRUE. See the section “Irreducible Infeasible Set” on page 519 for details.

- I_L  the lower bound of the variable is needed for the IIS
- I_U  the upper bound of the variable is needed for the IIS
- I_F  both bounds of the variable are needed for the IIS (the variable is fixed or has conflicting bounds)

Constraint Status

The .status suffix of a constraint specifies the status of the slack variable for that constraint. The suffix can take one of the following values:

- B  basic variable
- L  nonbasic variable at its lower bound
- U  nonbasic variable at its upper bound
- F  free variable
- A  superbasic variable (a nonbasic variable that has a value strictly between its bounds)
- I  LP model infeasible (all decision variables have .status equal to I)

The following values can appear only if option IIS=TRUE. See the section “Irreducible Infeasible Set” on page 519 for details.
Irreducible Infeasible Set

For a linear programming problem, an irreducible infeasible set (IIS) is an infeasible subset of constraints and variable bounds that will become feasible if any single constraint or variable bound is removed. It is possible to have more than one IIS in an infeasible LP. Identifying an IIS can help isolate the structural infeasibility in an LP.

The presolver in the LP algorithms can detect infeasibility, but it identifies only the variable bound or constraint that triggers the infeasibility.

The IIS=TRUE option directs the LP solver to search for an IIS in a specified LP. You should specify the OPTMODEL option PRESOLVER=NONE when you specify IIS=TRUE; otherwise the IIS results can be incomplete. The LP solver does not apply the LP presolver to the problem during the IIS search. If the LP solver detects an IIS, it updates the .status suffix of the decision variables and constraints, and then it stops.

The number of iterations that are reported in the macro variable and the ODS table is the total number of simplex iterations. This total includes the initial LP solve and all subsequent iterations during the constraint deletion phase.

The IIS= option can add special values to the .status suffixes of variables and constraints. (For more information, see the section “Variable and Constraint Status” on page 518.) For constraints, a status of “I_L”, “I_U”, or “I_F” indicates that the “GE” (≥), “LE” (≤), or “EQ” (=) constraint, respectively, is part of the IIS. For range constraints, a status of “I_L” or “I_U” indicates that the lower or upper bound, respectively, of the constraint is needed for the IIS, and “I_F” indicates that the bounds in the constraint are conflicting. For variables, a status of “I_L”, “I_U”, or “I_F” indicates that the lower, upper, or both bounds, respectively, of the variable are needed for the IIS. From this information, you can identify both the names of the constraints (variables) in the IIS and the corresponding bound where infeasibility occurs.

Making any one of the constraints or variable bounds in the IIS nonbinding removes the infeasibility from the IIS. In some cases, changing a right-hand side or bound by a finite amount removes the infeasibility. However, the only way to guarantee removal of the infeasibility is to set the appropriate right-hand side or bound to ∞ or −∞. Because it is possible for an LP to have multiple irreducible infeasible sets, simply removing the infeasibility from one set might not make the entire problem feasible. To make the entire problem feasible, you can specify IIS=TRUE and rerun the LP solver after removing the infeasibility from an IIS. Repeating this process until the LP solver no longer detects an IIS results in a feasible problem. This approach to infeasibility repair can produce different end problems depending on which right-hand sides and bounds you choose to relax.

The IIS= option in the LP solver uses two different methods to identify an IIS:

1. Based on the result of the initial solve, the sensitivity filter removes several constraints and variable bounds immediately while still maintaining infeasibility. This phase is quick and dramatically reduces the size of the IIS.
2. Next, the deletion filter removes each remaining constraint and variable bound one by one to check which of them are needed to obtain an infeasible system. This second phase is more time consuming, but it ensures that the IIS set that the LP solver returns is indeed irreducible. The progress of the deletion filter is reported at regular intervals. The sensitivity filter might be called again during the deletion filter to improve performance.

See Example 12.4 for an example that demonstrates the use of the IIS= option in locating and removing infeasibilities.

---

### Macro Variable _OROPTMODEL_

The OPTMODEL procedure always creates and initializes a SAS macro called _OROPTMODEL_. This variable contains a character string. After each PROC OROPTMODEL run, you can examine this macro by specifying `%put &_OROPTMODEL_;` and check the execution of the most recently invoked solver from the value of the macro variable. The various terms of the variable after the LP solver is called are interpreted as follows.

**STATUS**

indicates the solver status at termination. It can take one of the following values:

- **OK**: The solver terminated normally.
- **SYNTAX_ERROR**: Incorrect syntax was used.
- **DATA_ERROR**: The input data were inconsistent.
- **OUT_OF_MEMORY**: Insufficient memory was allocated to the procedure.
- **IO_ERROR**: A problem occurred in reading or writing data.
- **SEMANTIC_ERROR**: An evaluation error, such as an invalid operand type, occurred.
- **ERROR**: The status cannot be classified into any of the preceding categories.

**ALGORITHM**

indicates the algorithm that produces the solution data in the macro variable. This term appears only when STATUS=OK. It can take one of the following values:

- **PS**: The primal simplex algorithm produced the solution data.
- **DS**: The dual simplex algorithm produced the solution data.
- **NS**: The network simplex algorithm produced the solution data.
- **IP**: The interior point algorithm produced the solution data.
- **DECOMP**: The decomposition algorithm produced the solution data.

When you run algorithms concurrently (ALGORITHM=CON), this term indicates which algorithm is the first to terminate.
SOLUTION_STATUS
indicates the solution status at termination. It can take one of the following values:

OPTIMAL The solution is optimal.
CONDITIONAL_OPTIMAL The solution is optimal, but some infeasibilities (primal, dual or bound) exceed tolerances due to scaling or pre-processing.
FEASIBLE The problem is feasible.
INFEASIBLE The problem is infeasible.
UNBOUNDED The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED The problem is infeasible or unbounded.
BAD_PROBLEM_TYPE The problem type is unsupported by the solver.
ITERATION_LIMIT_REACHED The maximum allowable number of iterations was reached.
TIME_LIMIT_REACHED The solver reached its execution time limit.
FUNCTION_CALL_LIMIT_REACHED The solver reached its limit on function evaluations.
INTERRUPTED The solver was interrupted externally.
FAILED The solver failed to converge, possibly due to numerical issues.

When SOLUTION_STATUS has a value of OPTIMAL, CONDITIONAL_OPTIMAL, ITERATION_LIMIT_REACHED, or TIME_LIMIT_REACHED, all terms of the _OROPTMODEL_ macro variable are present; for other values of SOLUTION_STATUS, some terms do not appear.

OBJECTIVE
indicates the objective value obtained by the solver at termination.

PRIMAL_INFEASIBILITY
indicates, for the primal simplex and dual simplex algorithms, the maximum (absolute) violation of the primal constraints by the primal solution. For the interior point algorithm, this term indicates the relative violation of the primal constraints by the primal solution.

DUAL_INFEASIBILITY
indicates, for the primal simplex and dual simplex algorithms, the maximum (absolute) violation of the dual constraints by the dual solution. For the interior point algorithm, this term indicates the relative violation of the dual constraints by the dual solution.

BOUND_INFEASIBILITY
indicates, for the primal simplex and dual simplex algorithms, the maximum (absolute) violation of the lower or upper bounds by the primal solution. For the interior point algorithm, this term indicates the relative violation of the lower or upper bounds by the primal solution.

DUALITY_GAP
indicates the (relative) duality gap. This term appears only if the option ALGORITHM=INTERIORPOINT is specified in the SOLVE statement.
COMPLEMENTARITY indicates the (absolute) complementarity. This term appears only if the option ALGORITHM=INTERIORPOINT is specified in the SOLVE statement.

ITERATIONS indicates the number of iterations taken to solve the problem. When the network simplex algorithm is used, this term indicates the number of network simplex iterations taken to solve the network relaxation. When crossover is enabled, this term indicates the number of interior point iterations taken to solve the problem.

ITERATIONS2 indicates the number of simplex iterations performed by the secondary algorithm. The network simplex algorithm selects the secondary algorithm automatically unless a value has been specified for the ALGORITHM2= option. When crossover is enabled, the secondary algorithm is selected automatically. This term appears only if the network simplex algorithm is used or if crossover is enabled.

PRESOLVE_TIME indicates the time (in seconds) used in preprocessing.

SOLUTION_TIME indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

When SOLUTION_STATUS has a value of OPTIMAL, CONDITIONAL_OPTIMAL, ITERATION_LIMIT_REACHED, or TIME_LIMIT_REACHED, all terms of the _OROPTMODEL_ macro variable are present; for other values of SOLUTION_STATUS, some terms do not appear.

Examples: LP Solver

Example 12.1: Diet Problem

Consider the problem of diet optimization. There are six different foods: bread, milk, cheese, potato, fish, and yogurt. The cost and nutrition values per unit are displayed in Table 12.2.

<table>
<thead>
<tr>
<th>Table 12.2</th>
<th>Cost and Nutrition Values</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bread</td>
</tr>
<tr>
<td>Cost</td>
<td>2.0</td>
</tr>
<tr>
<td>Protein, g</td>
<td>4.0</td>
</tr>
<tr>
<td>Fat, g</td>
<td>1.0</td>
</tr>
<tr>
<td>Carbohydrates, g</td>
<td>15.0</td>
</tr>
<tr>
<td>Calories</td>
<td>90</td>
</tr>
</tbody>
</table>

The following SAS code creates the data set fooddata of Table 12.2:
Example 12.1: Diet Problem

```plaintext
data fooddata;
  infile datalines;
  input name $  cost prot fat carb cal;
  datalines;
  Bread  2  4  1  15  90
  Milk   3.5 8  5  11.7 120
  Cheese 8  7  9  0.4 106
  Potato 1.5 1.3 0.1 22.6 97
  Fish   11  8  7  0  130
  Yogurt 1  9.2 1  17  180
;
```

The objective is to find a minimum-cost diet that contains at least 300 calories, not more than 10 grams of protein, not less than 10 grams of carbohydrates, and not less than 8 grams of fat. In addition, the diet should contain at least 0.5 unit of fish and no more than 1 unit of milk.

You can model the problem and solve it by using PROC OPTMODEL as follows:

```plaintext
proc optmodel;
  /* declare index set */
  set<str> FOOD;

  /* declare variables */
  var diet{FOOD} >= 0;

  /* objective function */
  num cost{FOOD};
  min f=sum{i in FOOD}cost[i]*diet[i];

  /* constraints */
  num prot{FOOD};
  num fat{FOOD};
  num carb{FOOD};
  num cal{FOOD};
  num min_cal, max_prot, min_carb, min_fat;
  con cal_con: sum{i in FOOD}cal[i]*diet[i] >= 300;
  con prot_con: sum{i in FOOD}prot[i]*diet[i] <= 10;
  con carb_con: sum{i in FOOD}carb[i]*diet[i] >= 10;
  con fat_con: sum{i in FOOD}fat[i]*diet[i] >= 8;

  /* read parameters */
  read data fooddata into FOOD=[name] cost prot fat carb cal;

  /* bounds on variables */
  diet['Fish'].lb = 0.5;
  diet['Milk'].ub = 1.0;

  /* solve and print the optimal solution */
  solve with lp/logfreq=1; /* print each iteration to log */
  print diet;
```

The optimal solution and the optimal objective value are displayed in Output 12.1.1.
### Output 12.1.1 Optimal Solution to the Diet Problem

#### The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
<td>Minimization</td>
</tr>
<tr>
<td>Objective Function</td>
<td>f</td>
</tr>
<tr>
<td>Objective Type</td>
<td>Linear</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Variables</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded Above</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below</td>
<td>5</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
<td>1</td>
</tr>
<tr>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>Fixed</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Constraints</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear LE (≤)</td>
<td>1</td>
</tr>
<tr>
<td>Linear EQ (=)</td>
<td>0</td>
</tr>
<tr>
<td>Linear GE (≥)</td>
<td>3</td>
</tr>
<tr>
<td>Linear Range</td>
<td>0</td>
</tr>
</tbody>
</table>

| Constraint Coefficients | 23 |

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>LP</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Dual Simplex</td>
</tr>
<tr>
<td>Objective Function</td>
<td>f</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Optimal</td>
</tr>
<tr>
<td>Objective Value</td>
<td>12.081337881</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Primal Infeasibility</th>
<th>1.776357E-15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
<td>0</td>
</tr>
</tbody>
</table>

| Iterations | 5 |
| Presolve Time | 0.00 |
| Solution Time | 0.01 |

<table>
<thead>
<tr>
<th>[1] diet</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Bread</td>
<td>0.000000</td>
</tr>
<tr>
<td>Cheese</td>
<td>0.449499</td>
</tr>
<tr>
<td>Fish</td>
<td>0.500000</td>
</tr>
<tr>
<td>Milk</td>
<td>0.053599</td>
</tr>
<tr>
<td>Potato</td>
<td>1.865168</td>
</tr>
<tr>
<td>Yogurt</td>
<td>0.000000</td>
</tr>
</tbody>
</table>
Example 12.2: Reoptimizing the Diet Problem Using BASIS=WARMSTART

After an LP is solved, you might want to change a set of the parameters of the LP and solve the problem again. This can be done efficiently in PROC OPTMODEL. The warm start technique uses the optimal solution of the solved LP as a starting point and solves the modified LP problem faster than it can be solved again from scratch. This example illustrates reoptimizing the diet problem described in Example 12.1.

Assume the optimal solution is found by the SOLVE statement. Instead of quitting the OPTMODEL procedure, you can continue to solve several variations of the original problem.

Suppose the cost of cheese increases from 8 to 10 per unit and the cost of fish decreases from 11 to 7 per serving unit. You can change the parameters and solve the modified problem by submitting the following code:

```plaintext
cost['Cheese']=10; cost['Fish']=7;
solve with lp/presolver=none
   basis=warmstart
   algorithm=ps
   logfreq=1;
print diet;
```

Note that the primal simplex algorithm is preferred because the primal solution to the last-solved LP is still feasible for the modified problem in this case. The solutions to the original diet problem and the modified problem are shown in Output 12.2.1.

Output 12.2.1 Optimal Solutions to the Original Diet Problem and the Diet Problem with Modified Objective Function

<table>
<thead>
<tr>
<th>The OPTMODEL Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Summary</td>
</tr>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
<tr>
<td>Number of Variables</td>
</tr>
<tr>
<td>Bounded Above</td>
</tr>
<tr>
<td>Bounded Below</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
</tr>
<tr>
<td>Free</td>
</tr>
<tr>
<td>Fixed</td>
</tr>
<tr>
<td>Number of Constraints</td>
</tr>
<tr>
<td>Linear LE (&lt;=)</td>
</tr>
<tr>
<td>Linear EQ (=)</td>
</tr>
<tr>
<td>Linear GE (&gt;=)</td>
</tr>
<tr>
<td>Linear Range</td>
</tr>
<tr>
<td>Constraint Coefficients</td>
</tr>
</tbody>
</table>
**Output 12.2.1 continued**

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Primal Infeasibility</th>
<th>1.776357E-15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dual Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
<td>0</td>
</tr>
</tbody>
</table>

| Iterations | 5 |
| Presolve Time | 0.00 |
| Solution Time | 0.01 |

<table>
<thead>
<tr>
<th>[1]</th>
<th>diet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bread</td>
</tr>
<tr>
<td></td>
<td>Cheese</td>
</tr>
<tr>
<td></td>
<td>Fish</td>
</tr>
<tr>
<td></td>
<td>Milk</td>
</tr>
<tr>
<td></td>
<td>Potato</td>
</tr>
<tr>
<td></td>
<td>Yogurt</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
</tbody>
</table>

| Number of Variables | 6 |
| Bounded Above      | 0 |
| Bounded Below      | 5 |
| Bounded Below and Above | 1 |
| Free               | 0 |
| Fixed              | 0 |

| Number of Constraints | 4 |
| Linear LE (<=)        | 1 |
| Linear EQ (=)         | 0 |
| Linear GE (>=)        | 3 |
| Linear Range          | 0 |

| Constraint Coefficients | 23 |
Output 12.2.1 continued

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
</tr>
<tr>
<td>Dual Infeasibility</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

The following iteration log indicates that it takes the LP solver no more iterations to solve the modified problem by using BASIS=WARMSTART, since the optimal solution to the original problem remains optimal after the objective function is changed.
Next, restore the original coefficients of the objective function and consider the case that you need a diet that supplies at least 150 calories. You can change the parameters and solve the modified problem by submitting the following code:

```r
cost['Cheese']=8; cost['Fish']=11; cal_con.lb=150;
solve with lp/presolver=none
  basis=warmstart
  algorithm=ds
  logfreq=1;
print diet;
```

Note that the dual simplex algorithm is preferred because the dual solution to the last-solved LP is still feasible for the modified problem in this case. The solution is shown in Output 12.2.3.
Output 12.2.3 Optimal Solution to the Diet Problem with Modified RHS

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
</tbody>
</table>

| Number of Variables | 6 |
| Bounded Above | 0 |
| Bounded Below | 5 |
| Bounded Below and Above | 1 |
| Free | 0 |
| Fixed | 0 |

| Number of Constraints | 4 |
| Linear LE (<=) | 1 |
| Linear EQ (=) | 0 |
| Linear GE (>=) | 3 |
| Linear Range | 0 |

Constraint Coefficients  23

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
</tbody>
</table>

Primal Infeasibility  0
Dual Infeasibility  0
Bound Infeasibility  0

Iterations  2
Presolve Time  0.00
Solution Time  0.00

[1] diet
   Bread  0.00000
   Cheese  0.18481
   Fish  0.50000
   Milk  0.56440
   Potato  0.14702
   Yogurt  0.00000

The following iteration log indicates that it takes the LP solver just one more phase II iteration to solve the modified problem by using BASIS=WARMSTART.
Output 12.2.4 Log

NOTE: There were 6 observations read from the data set WORK.FOODDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 linear constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 linear constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Iteration</th>
<th>Objective</th>
<th>Entering Variable</th>
<th>Leaving Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>8.813205E+00</td>
<td>cal_con ($)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>9.174413E+00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Optimal.
NOTE: Objective = 9.1744131985.
NOTE: The Dual Simplex solve time is 0.00 seconds.
Next, restore the original constraint on calories and consider the case that you need a diet that supplies no more than 550 mg of sodium per day. The following row is appended to Table 12.2.

<table>
<thead>
<tr>
<th>sodium, mg</th>
<th>Bread</th>
<th>Milk</th>
<th>Cheese</th>
<th>Potato</th>
<th>Fish</th>
<th>Yogurt</th>
</tr>
</thead>
<tbody>
<tr>
<td>148</td>
<td>122</td>
<td>337</td>
<td>186</td>
<td>56</td>
<td>132</td>
<td></td>
</tr>
</tbody>
</table>

You can change the parameters, add the new constraint, and solve the modified problem by submitting the following code:

```plaintext
cal_con.lb=300;
num sod(FOOD)=[148 122 337 186 56 132];
con sodium: sum{i in FOOD}sod[i]*diet[i] <= 550;
solve with lp/presolver=none
   basis=warmstart
   logfreq=1;
print diet;
```

The solution is shown in Output 12.2.5.

**Output 12.2.5**  Optimal Solution to the Diet Problem with Additional Constraint

**The OPTMODEL Procedure**

<table>
<thead>
<tr>
<th>Problem Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
<td>Minimization</td>
</tr>
<tr>
<td>Objective Function</td>
<td>f</td>
</tr>
<tr>
<td>Objective Type</td>
<td>Linear</td>
</tr>
<tr>
<td>Number of Variables</td>
<td>6</td>
</tr>
<tr>
<td>Bounded Above</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below</td>
<td>5</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
<td>1</td>
</tr>
<tr>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>Fixed</td>
<td>0</td>
</tr>
<tr>
<td>Number of Constraints</td>
<td>5</td>
</tr>
<tr>
<td>Linear LE (&lt;=)</td>
<td>2</td>
</tr>
<tr>
<td>Linear EQ (=)</td>
<td>0</td>
</tr>
<tr>
<td>Linear GE (&gt;=)</td>
<td>3</td>
</tr>
<tr>
<td>Linear Range</td>
<td>0</td>
</tr>
</tbody>
</table>

**Constraint Coefficients**  29
### Output 12.2.5 continued

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solver</strong></td>
</tr>
<tr>
<td><strong>Algorithm</strong></td>
</tr>
<tr>
<td><strong>Objective Function</strong></td>
</tr>
<tr>
<td><strong>Solution Status</strong></td>
</tr>
<tr>
<td><strong>Objective Value</strong></td>
</tr>
</tbody>
</table>

Primal Infeasibility 1.776357E-15  
Dual Infeasibility 0  
Bound Infeasibility 0

| Iterations | 1 |
| Presolve Time | 0.00 |
| Solution Time | 0.00 |

<table>
<thead>
<tr>
<th>[1] diet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bread</td>
</tr>
<tr>
<td>Cheese</td>
</tr>
<tr>
<td>Fish</td>
</tr>
<tr>
<td>Milk</td>
</tr>
<tr>
<td>Potato</td>
</tr>
<tr>
<td>Yogurt</td>
</tr>
</tbody>
</table>
Example 12.3: Two-Person Zero-Sum Game

Consider a two-person zero-sum game (where one person wins what the other person loses). The players make moves simultaneously, and each has a choice of actions. There is a payoff matrix that indicates the amount one player gives to the other under each combination of actions:

Player II plays $j$

\[
\begin{array}{cccc}
1 & 2 & 3 & 4 \\
1 & -5 & 3 & 1 & 8 \\
2 & 5 & 5 & 4 & 6 \\
3 & -4 & 6 & 0 & 5 \\
\end{array}
\]

If player I makes move $i$ and player II makes move $j$, then player I wins (and player II loses) $a_{ij}$. What is the best strategy for the two players to adopt? This example is simple enough to be analyzed from observation. Suppose player I plays 1 or 3; the best response of player II is to play 1. In both cases, player I loses and player II wins. So the best action for player I is to play 2. In this case, the best response for player II is to play 3, which minimizes the loss. In this case, $(2, 3)$ is a pure-strategy Nash equilibrium in this game.

For illustration, consider the following mixed strategy case. Assume that player I selects $i$ with probability $p_i$, $i = 1, 2, 3$, and player II selects $j$ with probability $q_j$, $j = 1, 2, 3, 4$. Consider player II’s problem of minimizing the maximum expected payout:

The following iteration log indicates that it takes the LP solver no more iterations to solve the modified problem by using the BASIS=WARMSTART option, since the optimal solution to the original problem remains optimal after one more constraint is added.

Output 12.2.6 Log

NOTE: There were 6 observations read from the data set WORK.FOODDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 4 linear constraints (1 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 23 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 5 linear constraints (2 LE, 0 EQ, 3 GE, 0 range).
NOTE: The problem has 29 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Iteration</th>
<th>Objective Value</th>
<th>Time</th>
<th>Entering Variable</th>
<th>Leaving Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>2</td>
<td>1.208134E+01</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Optimal.
NOTE: Objective = 12.081337881.
NOTE: The Dual Simplex solve time is 0.00 seconds.
\[
\min_{q} \left\{ \max_{i} \sum_{j=1}^{4} a_{ij} q_{j} \right\} \quad \text{subject to} \quad \sum_{j=1}^{4} q_{ij} = 1, \quad q \geq 0
\]

This is equivalent to

\[
\min_{q,v} v \quad \text{subject to} \quad \sum_{j=1}^{4} a_{ij} q_{j} \leq v \quad \forall i
\]
\[
\sum_{j=1}^{4} q_{j} = 1
\]
\[
q \geq 0
\]

The problem can be transformed into a more standard format by making a simple change of variables: \( x_j = q_j / v \). The preceding LP formulation now becomes

\[
\min_{x,v} v \quad \text{subject to} \quad \sum_{j=1}^{4} a_{ij} x_j \leq 1 \quad \forall i
\]
\[
\sum_{j=1}^{4} x_j = 1 / v
\]
\[
q \geq 0
\]

which is equivalent to

\[
\max_{x} \sum_{j=1}^{4} x_j \quad \text{subject to} \quad A x \leq 1, \quad x \geq 0
\]

where \( A \) is the payoff matrix and \( 1 \) is a vector of 1’s. It turns out that the corresponding optimization problem from player I’s perspective can be obtained by solving the dual problem, which can be written as

\[
\min_{y} \sum_{i=1}^{3} y_i \quad \text{subject to} \quad A^T y \geq 1, \quad y \geq 0
\]

You can model the problem and solve it by using PROC OPTMODEL as follows:

```plaintext
proc optmodel;
  num a{1..3, 1..4}=[-5 3 1 8
  5 5 4 6
  -4 6 0 5];
  var x{1..4} >= 0;
  max f = sum{i in 1..4}x[i];
  con c{i in 1..3}: sum{j in 1..4}a[i,j]*x[j] <= 1;
  solve with lp / algorithm = ps presolver = none logfreq = 1;
  print x;
  print c.dual;
quit;
```

The optimal solution is displayed in Output 12.3.1.
**Output 12.3.1** Optimal Solutions to the Two-Person Zero-Sum Game

**The OPTMODEL Procedure**

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
<tr>
<td>Number of Variables</td>
</tr>
<tr>
<td>Bounded Above</td>
</tr>
<tr>
<td>Bounded Below</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
</tr>
<tr>
<td>Free</td>
</tr>
<tr>
<td>Fixed</td>
</tr>
<tr>
<td>Number of Constraints</td>
</tr>
<tr>
<td>Linear LE (&lt;=)</td>
</tr>
<tr>
<td>Linear EQ (=)</td>
</tr>
<tr>
<td>Linear GE (&gt;=)</td>
</tr>
<tr>
<td>Linear Range</td>
</tr>
</tbody>
</table>

**Constraint Coefficients**

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
</tr>
<tr>
<td>Dual Infeasibility</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c}
[1] \quad x \\
1 \quad 0.00 \\
2 \quad 0.00 \\
3 \quad 0.25 \\
4 \quad 0.00 \\
\end{array}
\]

\[
\begin{array}{c}
[1] \quad c.DUAL \\
1 \quad 0.00 \\
2 \quad 0.25 \\
3 \quad 0.00 \\
\end{array}
\]
The optimal solution $x^* = (0, 0, 0.25, 0)$ with an optimal value of 0.25. Therefore the optimal strategy for player II is $q^* = x^*/0.25 = (0, 0, 1, 0)$. You can check the optimal solution of the dual problem by using the constraint suffix “.dual”. So $y^* = (0, 0.25, 0)$ and player I’s optimal strategy is $(0, 1, 0)$. The solution is consistent with our intuition from observation.

**Example 12.4: Finding an Irreducible Infeasible Set**

This example demonstrates the use of the IIS= option to locate an irreducible infeasible set. Suppose you want to solve a linear program that has the following simple formulation:

\[
\begin{align*}
\text{min} & \quad x_1 + x_2 + x_3 \quad & \text{(cost)} \\
\text{subject to} & \quad x_1 + x_2 \quad \geq 10 \quad & \text{(con1)} \\
& \quad x_1 + x_3 \quad \leq 4 \quad & \text{(con2)} \\
& \quad 4 \leq x_2 + x_3 \quad \leq 5 \quad & \text{(con3)} \\
& \quad x_1, x_2 \quad \geq 0 \\
& \quad 0 \leq x_3 \quad \leq 3
\end{align*}
\]

It is easy to verify that the following three constraints (or rows) and one variable (or column) bound form an IIS for this problem:

\[
\begin{align*}
& \quad x_1 + x_2 \quad \geq 10 \quad \text{(con1)} \\
& \quad x_1 \quad \leq 4 \quad \text{(con2)} \\
& \quad x_2 + x_3 \quad \leq 5 \quad \text{(con3)} \\
& \quad x_3 \quad \geq 0
\end{align*}
\]

You can formulate the problem and call the LP solver by using the following statements:

```plaintext
proc optmodel presolver=none;
  /* declare variables */
  var x{1..3} >=0;

  /* upper bound on variable x[3] */
  x[3].ub = 3;

  /* objective function */
  min obj = x[1] + x[2] + x[3];

  /* constraints */

  solve with lp / iis=true;

  print x.status;
  print c1.status c2.status c3.status;
  expand / IIS;
```
Example 12.4: Finding an Irreducible Infeasible Set

The notes printed in the log appear in Output 12.4.1.

Output 12.4.1 Finding an IIS: Log

NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 3 linear constraints (1 LE, 0 EQ, 1 GE, 1 range).
NOTE: The problem has 6 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP solver is called.
NOTE: The IIS option is enabled.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Iteration</th>
<th>Value</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P 1</td>
<td>1</td>
<td>6.0000000E+00</td>
<td>0</td>
</tr>
<tr>
<td>P 1</td>
<td>3</td>
<td>1.0000000E+00</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Applying the IIS sensitivity filter.
NOTE: The sensitivity filter removed 1 constraints and 3 variable bounds.
NOTE: Applying the IIS deletion filter.
NOTE: Processing constraints.

<table>
<thead>
<tr>
<th>Processed</th>
<th>Removed</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Processing variable bounds.

<table>
<thead>
<tr>
<th>Processed</th>
<th>Removed</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The deletion filter removed 0 constraints and 0 variable bounds.
NOTE: The IIS= option found this problem to be infeasible.
NOTE: The IIS= option found an irreducible infeasible set with 1 variables and 3 constraints.
NOTE: The IIS solve time is 0.01 seconds.

There are two ways to display the rows and columns that are included in the IIS. One way is to use the PRINT statement to print the value of the .status suffix for each variable and constraint. The more straightforward approach is to use the EXPAND statement with the IIS option.

The output of both approaches appears in Output 12.4.2. The value of the .status suffix for the variables x[1] and x[2] is missing, so the variable bounds for x[1] and x[2] are not in the IIS.
Chapter 12: The Linear Programming Solver

Output 12.4.2 Solution Summary, Variable Status, and Constraint Status

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
</tbody>
</table>

| Iterations | 10 |
| Presolve Time | 0.00 |
| Solution Time | 0.01 |

[1] x.STATUS
1
2
3 I_L

<table>
<thead>
<tr>
<th>c1.STATUS</th>
<th>c2.STATUS</th>
<th>c3.STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_L</td>
<td>I_U</td>
<td>I_U</td>
</tr>
</tbody>
</table>

The OPTMODEL Procedure

Var x[3] >= 0 <= 3

The value of c3.status is I_U, which indicates that \( x_2 + x_3 \leq 5 \) is an element of the IIS. The original constraint is c3, a range constraint with a lower bound of 4. If you choose to remove the constraint \( x_2 + x_3 \leq 5 \), you can change the value of c3.ub to the largest positive number representable in your operating environment. You can specify this number by using the CONSTANT function.

The modified LP problem is specified and solved by adding the following lines to the original PROC OPTMODEL call.

```latex
/* relax upper bound on constraint c3 */
c3.ub = constant('BIG');

solve with lp / iis=true;
```

Because one element of the IIS has been removed, the modified LP problem should no longer contain the infeasible set. Due to the size of this problem, there should be no additional irreducible infeasible sets.

The notes shown in Output 12.4.3 are printed to the log.
Example 12.5: Using the Network Simplex Algorithm

This example demonstrates how you can use the network simplex algorithm to find the minimum-cost flow in a directed graph. Consider the directed graph in Figure 12.4, which appears in Ahuja, Magnanti, and Orlin (1993).
You can use the following SAS statements to create the input data sets `nodedata` and `arcdata`:

```sas
data nodedata;
    input _node_ $ _sd_;
data lines;
    1 10
    2 20
    3 0
    4 -5
    5 0
    6 0
    7 -15
    8 -10;
;

data arcdata;
    input _tail_ $ _head_ $ _lo_ _capac_ _cost_;
data lines;
    1 4 0 15 2
    2 1 0 10 1
    2 3 0 10 0
    2 6 0 10 6
    3 4 0 5 1
    3 5 0 10 4
    4 7 0 10 5
    5 6 0 20 2
    5 7 0 15 7
    6 8 0 10 8
    7 8 0 15 9;
;
You can use the following call to PROC OPTMODEL to find the minimum-cost flow:
Example 12.5: Using the Network Simplex Algorithm

```
proc optmodel;
    set <str> NODES;
        num supply_demand {NODES};

    set <str,str> ARCS;
        num arcLower {ARCS};
        num arcUpper {ARCS};
        num arcCost {ARCS};

    read data arcdata into ARCS=tail head
        arcLower=lo arcUpper=capac arcCost=cost;
    read data nodedata into NODES=node
        supply_demand=sd;

    var flow {<i,j> in ARCS} >= arcLower[i,j] <= arcUpper[i,j];
    min obj = sum {<i,j> in ARCS} arcCost[i,j] * flow[i,j];
    con balance {i in NODES}:
        sum {<(i),j> in ARCS} flow[i,j] - sum {<j,(i)> in ARCS} flow[j,i] = supply_demand[i];
    solve with lp / algorithm=ns scale=none logfreq=1;
    print flow;
quit;
%put &_OROPTMODEL_;
```

The optimal solution is displayed in Output 12.5.1.

**Output 12.5.1**  Network Simplex Algorithm: Primal Solution Output

**The OPTMODEL Procedure**

<table>
<thead>
<tr>
<th>Problem Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
<td>Minimization</td>
</tr>
<tr>
<td>Objective Function</td>
<td>obj</td>
</tr>
<tr>
<td>Objective Type</td>
<td>Linear</td>
</tr>
</tbody>
</table>

| Number of Variables | 11 |
| Bounded Above | 0 |
| Bounded Below | 0 |
| Bounded Below and Above | 11 |
| Free | 0 |
| Fixed | 0 |

| Number of Constraints | 8 |
| Linear LE (<=) | 0 |
| Linear EQ (=) | 8 |
| Linear GE (>=) | 0 |
| Linear Range | 0 |

| Constraint Coefficients | 22 |
Output 12.5.1 continued

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
</tr>
<tr>
<td>Dual Infeasibility</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Iterations2</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[1] [2] flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4 10</td>
</tr>
<tr>
<td>2 1 0</td>
</tr>
<tr>
<td>2 3 10</td>
</tr>
<tr>
<td>2 6 10</td>
</tr>
<tr>
<td>3 4 5</td>
</tr>
<tr>
<td>3 5 5</td>
</tr>
<tr>
<td>4 7 10</td>
</tr>
<tr>
<td>5 6 0</td>
</tr>
<tr>
<td>5 7 5</td>
</tr>
<tr>
<td>6 8 10</td>
</tr>
<tr>
<td>7 8 0</td>
</tr>
</tbody>
</table>

The optimal solution is represented graphically in Figure 12.5.

Figure 12.5 Minimum-Cost Network Flow Problem: Optimal Solution
Example 12.5: Using the Network Simplex Algorithm

The iteration log is displayed in Output 12.5.2.

Output 12.5.2 Log: Solution Progress

```
NOTE: There were 11 observations read from the data set WORK.ARCDATA.
NOTE: There were 8 observations read from the data set WORK.NODEDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 11 variables (0 free, 0 fixed).
NOTE: The problem has 8 linear constraints (0 LE, 8 EQ, 0 GE, 0 range).
NOTE: The problem has 22 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem is a pure network instance; PRESOLVER=None is used.
NOTE: The LP presolver value NONE is applied.
NOTE: The LP solver is called.
NOTE: The Network Simplex algorithm is used.
NOTE: The network has 8 rows (100.00%), 11 columns (100.00%), and 1 component.
NOTE: The network extraction and setup time is 0.00 seconds.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Primal Objective</th>
<th>Primal Infeasibility</th>
<th>Dual Infeasibility</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.000000E+00</td>
<td>2.000000E+01</td>
<td>8.900000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>0.000000E+00</td>
<td>2.000000E+01</td>
<td>8.900000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td>5.000000E+00</td>
<td>1.500000E+01</td>
<td>8.400000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>5.000000E+00</td>
<td>1.500000E+01</td>
<td>8.300000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>7.500000E+01</td>
<td>1.500000E+01</td>
<td>8.300000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>7.500000E+01</td>
<td>1.500000E+01</td>
<td>7.900000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>1.300000E+02</td>
<td>1.000000E+01</td>
<td>7.600000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>2.700000E+02</td>
<td>0.000000E+00</td>
<td>0.000000E+00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: The total Network Simplex solve time is 0.00 seconds.
NOTE: Optimal.
NOTE: Objective = 270.
STATUS=OK ALGORITHM=NS SOLUTION_STATUS=OPTIMAL OBJECTIVE=270
PRIMAL_INFEASIBILITY=0 DUAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0 ITERATIONS=8
ITERATIONS2=0 PRESOLVE_TIME=0.00 SOLUTION_TIME=0.00
```

Now, suppose there is a budget on the flow that comes out of arc 2: the total arc cost of flow that comes out of arc 2 cannot exceed 50. You can use the following call to PROC OPTMODEL to find the minimum-cost flow:

```
proc optmodel;
  set <str> NODES;
  num supply_demand {NODES};

  set <str,str> ARCS;
  num arcLower {ARCS};
  num arcUpper {ARCS};
  num arcCost {ARCS};

  read data arcdata into ARCS=[_tail_ _head_]
  arcLower=lo_ arcUpper=capac_ arcCost=cost_;
```
read data nodedata into NODES=[_node_] supply_demand=_sd_;

var flow {<i,j> in ARCS} >= arcLower[i,j] <= arcUpper[i,j];
min obj = sum {<i,j> in ARCS} arcCost[i,j] * flow[i,j];
con balance {i in NODES}:
    sum {<(i),j> in ARCS} flow[i,j] - sum {<j,(i)> in ARCS} flow[j,i]
    = supply_demand[i];
con budgetOn2:
    sum {<i,j> in ARCS: i='2'} arcCost[i,j] * flow[i,j] <= 50;
solve with lp / algorithm=ns scale=none logfreq=1;
print flow;
quit;
%put &_OROPTMODEL_;

The optimal solution is displayed in Output 12.5.3.

Output 12.5.3 Network Simplex Algorithm: Primal Solution Output

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
</tbody>
</table>

| Number of Variables          | 11                    |
| Bounded Above                | 0                     |
| Bounded Below                | 0                     |
| Bounded Below and Above      | 11                    |
| Free                         | 0                     |
| Fixed                        | 0                     |

| Number of Constraints        | 9                     |
| Linear LE (<=)               | 1                     |
| Linear EQ (=)                | 8                     |
| Linear GE (>=)               | 0                     |
| Linear Range                 | 0                     |

| Constraint Coefficients      | 24                    |
Example 12.5: Using the Network Simplex Algorithm

Output 12.5.3 continued

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
</tr>
<tr>
<td>Dual Infeasibility</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Iterations2</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[1] [2] flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 4 12</td>
</tr>
<tr>
<td>2 1 2</td>
</tr>
<tr>
<td>2 3 10</td>
</tr>
<tr>
<td>2 6 8</td>
</tr>
<tr>
<td>3 4 3</td>
</tr>
<tr>
<td>3 5 7</td>
</tr>
<tr>
<td>4 7 10</td>
</tr>
<tr>
<td>5 6 2</td>
</tr>
<tr>
<td>5 7 5</td>
</tr>
<tr>
<td>6 8 10</td>
</tr>
<tr>
<td>7 8 0</td>
</tr>
</tbody>
</table>

The optimal solution is represented graphically in Figure 12.6.

**Figure 12.6** Minimum-Cost Network Flow Problem: Optimal Solution (with Budget Constraint)

The iteration log is displayed in Output 12.5.4. Note that the network simplex algorithm extracts a subnetwork in this case.
Output 12.5.4 Log: Solution Progress

NOTE: There were 11 observations read from the data set WORK.ARCDATA.
NOTE: There were 8 observations read from the data set WORK.NODEDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 11 variables (0 free, 0 fixed).
NOTE: The problem has 9 linear constraints (1 LE, 8 EQ, 0 GE, 0 range).
NOTE: The problem has 24 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 4 variables and 4 constraints.
NOTE: The LP presolver removed 7 constraint coefficients.
NOTE: The presolved problem has 7 variables, 5 constraints, and 17 constraint coefficients.
NOTE: The LP solver is called.
NOTE: The Network Simplex algorithm is used.
NOTE: The network has 4 rows (80.00%), 7 columns (100.00%), and 1 component.
NOTE: The network extraction and setup time is 0.00 seconds.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Primal Objective</th>
<th>Primal Infeasibility</th>
<th>Primal Infeasibility</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>8.015000E+01</td>
<td>1.006000E+01</td>
<td>5.500000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>1.053000E+02</td>
<td>5.030000E+00</td>
<td>5.400000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td>1.053000E+02</td>
<td>5.030000E+00</td>
<td>5.400000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>1.353000E+02</td>
<td>3.000000E-02</td>
<td>4.900000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>1.356300E+02</td>
<td>0.000000E+00</td>
<td>4.700000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>1.356300E+02</td>
<td>0.000000E+00</td>
<td>0.000000E+00</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>2.700000E+02</td>
<td>0.000000E+00</td>
<td>0.000000E+00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: The total Network Simplex solve time is 0.00 seconds.
NOTE: The Dual Simplex algorithm is used.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Iteration</th>
<th>Objective Value</th>
<th>Time</th>
<th>Entering Variable</th>
<th>Leaving Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>D 2</td>
<td>1</td>
<td>2.700000E+02</td>
<td>0</td>
<td>flow['5','6']</td>
<td></td>
</tr>
<tr>
<td>budgetOn2  (S)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D 2</td>
<td>2</td>
<td>2.740000E+02</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Optimal.
NOTE: Objective = 274.
NOTE: The Simplex solve time is 0.01 seconds.

STATUS=OK ALGORITHM=NS SOLUTION_STATUS=OPTIMAL OBJECTIVE=274
PRIMAL_INFEASIBILITY=0 DUAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0 ITERATIONS=7
ITERATIONS2=2 PRESOLVE_TIME=0.00 SOLUTION_TIME=0.01
Example 12.6: Migration to OPTMODEL: Generalized Networks

The following example shows how to use PROC OPTMODEL to solve the example “Generalized Networks: Using the EXCESS= Option” in Chapter 5, “The NETFLOW Procedure” (SAS/OR User’s Guide: Mathematical Programming Legacy Procedures). The input data sets are the same as in the PROC NETFLOW example.

```sas
title 'Generalized Networks';

data garcs;
   input _from_ $ _to_ $ _cost_ _mult_;
datalines;
s1 d1 1 .
s1 d2 8 .
s2 d1 4 2
s2 d2 2 2
s2 d3 1 2
s3 d2 5 0.5
s3 d3 4 0.5 ;

data gnodes;
   input _node_ $ _sd_ ;
datalines;
s1 5
s2 20
s3 10
d1 -5
d2 -10
d3 -20 ;
```

The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to a SAS data set called GNETOUT:

```sas
proc optmodel;
   set <str> NODES;
      num _sd_ {NODES} init 0;
   read data gnodes into NODES=[_node_] _sd_;

   set <str,str> ARCS;
      num _lo_ {ARCS} init 0;
      num _capac_ {ARCS} init .;
      num _cost_ {ARCS};
      num _mult_ {ARCS} init 1;
   read data garcs nomiss into ARCS=[_from_ _to_] _cost_ _mult_;
   NODES = NODES union (union {<i,j> in ARCS} {i,j});
```
**Chapter 12: The Linear Programming Solver**

```plaintext
var Flow {<i,j> in ARCS} >= _lo_[i,j];
min obj = sum {<i,j> in ARCS} _cost_[i,j] * Flow[i,j];
con balance {i in NODES}: sum {<(i),j> in ARCS} Flow[i,j] - sum {<j,(i)> in ARCS} _mult_[j,i] * Flow[j,i] = _sd_[i];

num infinity = constant('BIG');
/* change equality constraint to le constraint for supply nodes */
for {i in NODES: _sd_[i] > 0} balance[i].lb = -infinity;
solve;

num _supply_ {<i,j> in ARCS} = (if _sd_[i] ne 0 then _sd_[i] else .);
um _demand_ {<i,j> in ARCS} = (if _sd_[j] ne 0 then -_sd_[j] else .);
um _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;
create data gnetout from [_from_ _to_] _cost_ _capac_ _lo_ _mult_ _supply_ _demand_ _flow_=Flow _fcost_;
quit;
```

To solve a generalized network flow problem, the usual balance constraint is altered to include the arc multiplier "mult[i,j]" in the second sum. The balance constraint is initially declared as an equality, but to mimic the EXCESS=SUPPLY option in PROC NETFLOW, the sense of this constraint is changed to "≤" by relaxing the constraint’s lower bound for supply nodes. The output data set is displayed in **Output 12.6.1**.

**Output 12.6.1** Optimal Solution with Excess Supply

<table>
<thead>
<tr>
<th>Obs</th>
<th><em>from</em></th>
<th><em>to</em></th>
<th><em>cost</em></th>
<th><em>capac</em></th>
<th><em>lo</em></th>
<th><em>mult</em></th>
<th><em>supply</em></th>
<th><em>demand</em></th>
<th><em>flow</em></th>
<th><em>fcost</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s1</td>
<td>d1</td>
<td>1</td>
<td>0</td>
<td>1.0</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>s1</td>
<td>d2</td>
<td>8</td>
<td>0</td>
<td>1.0</td>
<td>5</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>s2</td>
<td>d1</td>
<td>4</td>
<td>0</td>
<td>2.0</td>
<td>20</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>s2</td>
<td>d2</td>
<td>2</td>
<td>0</td>
<td>2.0</td>
<td>20</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>s2</td>
<td>d3</td>
<td>1</td>
<td>0</td>
<td>2.0</td>
<td>20</td>
<td>20</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>6</td>
<td>s3</td>
<td>d2</td>
<td>5</td>
<td>0</td>
<td>0.5</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>s3</td>
<td>d3</td>
<td>4</td>
<td>0</td>
<td>0.5</td>
<td>10</td>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The log is displayed in **Output 12.6.2**.
Output 12.6.2  OPTMODEL Log

NOTE: There were 6 observations read from the data set WORK.GNODES.
NOTE: There were 7 observations read from the data set WORK.GARCS.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 7 variables (0 free, 0 fixed).
NOTE: The problem has 6 linear constraints (3 LE, 3 EQ, 0 GE, 0 range).
NOTE: The problem has 14 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 2 variables and 2 constraints.
NOTE: The LP presolver removed 4 constraint coefficients.
NOTE: The presolved problem has 5 variables, 4 constraints, and 10 constraint coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.

<table>
<thead>
<tr>
<th>Phase</th>
<th>Iteration</th>
<th>Value</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>2</td>
<td>1.500000E+01</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>2.500000E+01</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Optimal.
NOTE: Objective = 25.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: The data set WORK.GNETOUT has 7 observations and 10 variables.

Now consider the previous example but with a slight modification to the arc multipliers, as in the PROC NETFLOW example:

```plaintext
data garcs1;
  input _from_ $ _to_ $ _cost_ _mult_;
datalines;
s1 d1 1 0.5
s1 d2 8 0.5
s2 d1 4 .
s2 d2 2 .
s2 d3 1 .
s3 d2 5 0.5
s3 d3 4 0.5
;
```

The following PROC OPTMODEL statements are identical to the preceding example, except for the balance constraint. The balance constraint is still initially declared as an equality, but to mimic the PROC NETFLOW EXCESS=DEMAND option, the sense of this constraint is changed to “≥” by relaxing the constraint’s upper bound for demand nodes.
proc optmodel;
set <str> NODES;
num _sd_ {NODES} init 0;
read data gnodes into NODES=[_node_] _sd_;
set <str,str> ARCS;
num _lo_ {ARCS} init 0;
num _capac_ {ARCS} init .;
num _cost_ {ARCS};
um _mult_ {ARCS} init 1;
read data garcs1 nomiss into ARCS=[_from_ _to_] _cost_ _mult_; NODES = NODES union (union {<i,j> in ARCS} {i,j});
var Flow {<i,j> in ARCS} >= _lo_[i,j];
for {<i,j> in ARCS: _capac_[i,j] ne .} Flow[i,j].ub = _capac_[i,j];
min obj = sum {<i,j> in ARCS} _cost_[i,j] * Flow[i,j];
con balance {i in NODES}: sum {<(i),j> in ARCS} Flow[i,j] - sum {<j,(i)> in ARCS} _mult_[j,i] * Flow[j,i] = _sd_[i];
num infinity = constant('BIG');
/* change equality constraint to ge constraint */
for {i in NODES: _sd_[i] < 0} balance[i].ub = infinity;
solve;
num _supply_ {<i,j> in ARCS} = (if _sd_[i] ne 0 then _sd_[i] else .);
num _demand_ {<i,j> in ARCS} = (if _sd_[j] ne 0 then -_sd_[j] else .);
num _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;
create data gnetout1 from [<_from_ _to_] _cost_ _capac_ _lo_ _mult_ _supply_ _demand_ _flow_=Flow _fcost_; quit;
The output data set is displayed in Output 12.6.3.

**Output 12.6.3** Optimal Solution with Excess Demand

<table>
<thead>
<tr>
<th>Obs</th>
<th><em>from</em></th>
<th><em>to</em></th>
<th><em>cost</em></th>
<th><em>capac</em></th>
<th><em>lo</em></th>
<th><em>mult</em></th>
<th><em>supply</em></th>
<th><em>demand</em></th>
<th><em>flow</em></th>
<th><em>fcost</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s1</td>
<td>d1</td>
<td>1</td>
<td>0</td>
<td>0.5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>s1</td>
<td>d2</td>
<td>8</td>
<td>0</td>
<td>0.5</td>
<td>5</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>s2</td>
<td>d1</td>
<td>4</td>
<td>0</td>
<td>1.0</td>
<td>20</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>s2</td>
<td>d2</td>
<td>2</td>
<td>0</td>
<td>1.0</td>
<td>20</td>
<td>10</td>
<td>5</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>s2</td>
<td>d3</td>
<td>1</td>
<td>0</td>
<td>1.0</td>
<td>20</td>
<td>20</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>s3</td>
<td>d2</td>
<td>5</td>
<td>0</td>
<td>0.5</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>s3</td>
<td>d3</td>
<td>4</td>
<td>0</td>
<td>0.5</td>
<td>10</td>
<td>20</td>
<td>10</td>
<td>40</td>
<td>40</td>
</tr>
</tbody>
</table>

The log is displayed in Output 12.6.4.
Example 12.7: Migration to OPTMODEL: Maximum Flow

The following example shows how to use PROC OPTMODEL to solve the example “Maximum Flow Problem” in Chapter 5, “The NETFLOW Procedure” *(SAS/OR User’s Guide: Mathematical Programming Legacy Procedures)*. The input data set is the same as in that example.

```plaintext
title 'Maximum Flow Problem';

data arcs;
  input _from_ $ _to_ $ _cost_ _capac_;
  datalines;
  S a . .
  S b . .
  a c 1 7
  b c 2 9
  a d 3 5
  b d 4 8
  c e 5 15
  d f 6 20
  e g 7 11
  f g 8 6
  e h 9 12
  f h 10 4
  g T . .
  h T . .
;```
The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to a SAS data set called GOUT3:

```sas
proc optmodel;
  str source = 'S';
  str sink = 'T';

  set <str> NODES;
  num _supdem_ {i in NODES} = (if i in {source, sink} then . else 0);

  set <str,str> ARCS;
  num _lo_ (ARCS) init 0;
  num _capac_ (ARCS) init .;
  num _cost_ (ARCS) init 0;
  read data arcs nomiss into ARCS=[_from_ _to_] _cost_ _capac_;
  NODES = (union {<i,j> in ARCS} {i,j});

  var Flow {<i,j> in ARCS} >= _lo_[i,j];
  for {<i,j> in ARCS: _capac_[i,j] ne .} Flow[i,j].ub = _capac_[i,j];
  max obj = sum {<i,j> in ARCS: j = sink} Flow[i,j];
  con balance {i in NODES diff {source, sink}}:
    sum {<(i),j> in ARCS} Flow[i,j]
    - sum {<j,(i)> in ARCS} Flow[j,i] = _supdem_[i];

  solve;

  num _supply_ {<i,j> in ARCS} =
    (if _supdem_[i] ne 0 then _supdem_[i] else .);
  num _demand_ {<i,j> in ARCS} =
    (if _supdem_[j] ne 0 then -_supdem_[j] else .);
  num _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;

  create data gout3 from [_from_ _to_] _cost_ _capac_ _lo_ _supply_ _demand_ _flow_=Flow _fcost_;
quit;
```

To solve a maximum flow problem, you solve a network flow problem that has a zero supply or demand at all nodes other than the source and sink nodes, as specified in the declaration of the _SUPDEM_ numeric parameter and the balance constraint. The objective declaration uses the logical condition J = SINK to maximize the flow into the sink node. The output data set is displayed in Output 12.7.1.
**Example 12.7: Migration to OPTMODEL: Maximum Flow**

Output 12.7.1 Optimal Solution

<table>
<thead>
<tr>
<th>Obs</th>
<th>from</th>
<th>to</th>
<th>cost</th>
<th>capac</th>
<th>lo</th>
<th>supply</th>
<th>demand</th>
<th>flow</th>
<th>fcost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>S</td>
<td>a</td>
<td>0</td>
<td>.</td>
<td>0</td>
<td>.</td>
<td>12</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>S</td>
<td>b</td>
<td>0</td>
<td>.</td>
<td>0</td>
<td>.</td>
<td>13</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>c</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>.</td>
<td>7</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>c</td>
<td>2</td>
<td>9</td>
<td>0</td>
<td>.</td>
<td>8</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>a</td>
<td>d</td>
<td>3</td>
<td>5</td>
<td>0</td>
<td>.</td>
<td>5</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>b</td>
<td>d</td>
<td>4</td>
<td>8</td>
<td>0</td>
<td>.</td>
<td>5</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>c</td>
<td>e</td>
<td>5</td>
<td>15</td>
<td>0</td>
<td>.</td>
<td>15</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>d</td>
<td>f</td>
<td>6</td>
<td>20</td>
<td>0</td>
<td>.</td>
<td>10</td>
<td>60</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>e</td>
<td>g</td>
<td>7</td>
<td>11</td>
<td>0</td>
<td>.</td>
<td>3</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>f</td>
<td>g</td>
<td>8</td>
<td>6</td>
<td>0</td>
<td>.</td>
<td>6</td>
<td>48</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>e</td>
<td>h</td>
<td>9</td>
<td>12</td>
<td>0</td>
<td>.</td>
<td>12</td>
<td>108</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>f</td>
<td>h</td>
<td>10</td>
<td>4</td>
<td>0</td>
<td>.</td>
<td>4</td>
<td>40</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>g</td>
<td>T</td>
<td>0</td>
<td>.</td>
<td>0</td>
<td>.</td>
<td>9</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>h</td>
<td>T</td>
<td>0</td>
<td>.</td>
<td>0</td>
<td>.</td>
<td>16</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

The log is displayed in Output 12.7.2.

Output 12.7.2 OPTMODEL Log

NOTE: There were 14 observations read from the data set WORK.ARCS.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 14 variables (0 free, 0 fixed).
NOTE: The problem has 8 linear constraints (0 LE, 8 EQ, 0 GE, 0 range).
NOTE: The problem has 24 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The problem is a pure network instance. The ALGORITHM=NETWORK option is recommended for solving problems with this structure.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed 10 variables and 6 constraints.
NOTE: The LP presolver removed 20 constraint coefficients.
NOTE: The presolved problem has 4 variables, 2 constraints, and 4 constraint coefficients.
NOTE: The LP solver is called.
NOTE: The Dual Simplex algorithm is used.

<table>
<thead>
<tr>
<th>Objective</th>
<th>Phase</th>
<th>Iteration</th>
<th>Value</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>2</td>
<td>1</td>
<td>2.500000E+01</td>
<td>0</td>
</tr>
<tr>
<td>P</td>
<td>2</td>
<td>4</td>
<td>2.500000E+01</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Optimal.
NOTE: Objective = 25.
NOTE: The Dual Simplex solve time is 0.01 seconds.
NOTE: The data set WORK.GOUT3 has 14 observations and 9 variables.
Example 12.8: Migration to OPTMODEL: Production, Inventory, Distribution

The following example shows how to use PROC OPTMODEL to solve the example “Production, Inventory, Distribution Problem” in Chapter 5, “The NETFLOW Procedure” (SAS/OR User’s Guide: Mathematical Programming Legacy Procedures). The input data sets are the same as in that example.

```plaintext
title 'Minimum-Cost Flow Problem';
title2 'Production Planning/Inventory/Distribution';

data node0;
    input _node_ $ _supdem_ ;
datalines;
fact1_1 1000
fact2_1 850
fact1_2 1000
fact2_2 1500
shop1_1 -900
shop2_1 -900
shop1_2 -900
shop2_2 -1450
;

data arc0;
    input _tail_ $ _head_ $ _cost_ _capac_ _lo_
        diagonal factory key_id $10. mth_made $ _name_&$17.;
datalines;
fact1_1 f1_mar_1 127.9 500 50 19 1 production March prod f1 19 mar
fact1_1 f1_apr_1 78.6 600 50 19 1 production April prod f1 19 apl
fact1_1 f1_may_1 95.1 400 50 19 1 production May .
f1_mar_1 f1_apr_1 15 50 . 19 1 storage March .
f1_apr_1 f1_may_1 12 50 . 19 1 storage April .
f1_apr_1 f1_mar_1 28 20 . 19 1 backorder April back f1 19 apl
f1_may_1 f1_apr_1 28 20 . 19 1 backorder May back f1 19 may
f1_mar_1 f2_mar_1 11 . . 19 . f1_to_2 March .
f1_apr_1 f2_apr_1 11 . . 19 . f1_to_2 April .
f1_may_1 f2_may_1 16 . . 19 . f1_to_2 May .
f1_mar_1 shop1_1 -327.65 250 . 19 1 sales March .
f1_apr_1 shop1_1 -300 250 . 19 1 sales April .
f1_may_1 shop1_1 -285 250 . 19 1 sales May .
f1_mar_1 shop2_1 -362.74 250 . 19 1 sales March .
f1_apr_1 shop2_1 -300 250 . 19 1 sales April .
f1_may_1 shop2_1 -245 250 . 19 1 sales May .
fact2_1 f2_mar_1 88.0 450 35 19 2 production March prod f2 19 mar
fact2_1 f2_apr_1 62.4 480 35 19 2 production April prod f2 19 apl
fact2_1 f2_may_1 133.8 250 35 19 2 production May .
f2_mar_1 f2_apr_1 18 30 . 19 2 storage March .
f2_apr_1 f2_may_1 20 30 . 19 2 storage April .
f2_apr_1 f2_mar_1 17 15 . 19 2 backorder April back f2 19 apl
f2_may_1 f2_apr_1 25 15 . 19 2 backorder May back f2 19 may
f2_mar_1 f1_mar_1 10 40 . 19 . f2_to_1 March .
f2_apr_1 f1_apr_1 11 40 . 19 . f2_to_1 April .
f2_may_1 f1_may_1 13 40 . 19 . f2_to_1 May .
```
Example 12.8: Migration to OPTMODEL: Production, Inventory, Distribution

The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to SAS data sets called ARC1 and NODE2:

```sas
proc optmodel;
  set <str> NODES;
  num _supdem_ {NODES} init 0;
  read data node0 into NODES=[_node_] _supdem_;

  set <str,str> ARCS;
  num _lo_ {ARCS} init 0;
  num _capac_ {ARCS} init .;
  num _cost_ {ARCS};
  num diagonal {ARCS};
  num factory {ARCS};
```

```sas
f2_mar_1 shop1_1 -297.4 250 . 19 2 sales March .
f2_apr_1 shop1_1 -290 250 . 19 2 sales April .
f2_mar_1 shop1_1 -292 250 . 19 2 sales May .
f2_apr_1 shop2_1 -272.7 250 . 19 2 sales March .
f2_apr_1 shop2_1 -312 250 . 19 2 sales April .
f2_mar_1 shop2_1 -299 250 . 19 2 sales May .
fact1_2 f1_mar_2 217.9 400 40 25 1 production March prod f1 25 mar
fact1_2 f1_apr_2 174.5 550 50 25 1 production April prod f1 25 apl
fact1_2 f1_may_2 133.3 350 40 25 1 production May .
f1_mar_2 f1_apr_2 20 40 . 25 1 storage March .
f1_apr_2 f1_may_2 18 40 . 25 1 storage April .

f1_apr_2 f1_mar_2 32 30 . 25 1 backorder April back f1 25 apl
f1_may_2 f1_apr_2 41 15 . 25 1 backorder May back f1 25 may
f1_mar_2 f2_mar_2 23 . . 25 . f1_to_2 March .
f1_apr_2 f2_apr_2 23 . . 25 . f1_to_2 April .
f1_may_2 f2_may_2 26 . . 25 . f1_to_2 May .
f1_mar_2 shop1_2 -559.76 . . 25 1 sales March .
f1_apr_2 shop1_2 -475.02 . . 25 1 sales April .
f1_may_2 shop1_2 -623.89 . . 25 1 sales May .
f1_apr_2 shop2_2 -549.68 . . 25 1 sales April .
f1_may_2 shop2_2 -460.00 . . 25 1 sales May .
fact2_2 f2_mar_2 182.0 650 35 25 2 production March prod f2 25 mar
fact2_2 f2_apr_2 196.7 680 35 25 2 production April prod f2 25 apl
fact2_2 f2_may_2 201.4 550 35 25 2 production May .
f2_mar_2 f2_apr_2 28 50 . 25 2 storage March .
f2_apr_2 f2_may_2 38 50 . 25 2 storage April .
f2_apr_2 f2_mar_2 31 15 . 25 2 backorder April back f2 25 apl
f2_may_2 f2_apr_2 54 15 . 25 2 backorder May back f2 25 may
f2_mar_2 f2_mar_2 20 25 . 25 . f2_to_1 March .
f2_apr_2 f1_apr_2 21 25 . 25 . f2_to_1 April .
f2_may_2 f1_may_2 43 25 . 25 . f2_to_1 May .
f2_mar_2 shop1_2 -567.83 500 . 25 2 sales March .
f2_apr_2 shop1_2 -489.06 500 . 25 2 sales May .

```

The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to SAS data sets called ARC1 and NODE2:

```sas
proc optmodel;
  set <str> NODES;
  num _supdem_ {NODES} init 0;
  read data node0 into NODES=[_node_] _supdem_;

  set <str,str> ARCS;
  num _lo_ {ARCS} init 0;
  num _capac_ {ARCS} init .;
  num _cost_ {ARCS};
  num diagonal {ARCS};
  num factory {ARCS};
```
str key_id {ARCS};
str mth_made {ARCS};
str _name_ {ARCS};
read data arc0 nomiss into ARCS=[_tail_ _head_] _lo_ _capac_ _cost_
    diagonal factory key_id mth_made _name_; 
NODES = NODES union (union {<i,j> in ARCS} {i,j});

var Flow {<i,j> in ARCS} >= _lo_[i,j];
for {<i,j> in ARCS: _capac_[i,j] ne .} Flow[i,j].ub = _capac_[i,j];
min obj = sum {<i,j> in ARCS} _cost_[i,j] * Flow[i,j];
con balance {i in NODES}: sum {<(i),j> in ARCS} Flow[i,j]
    - sum {<j,(i)> in ARCS} Flow[j,i] = _supdem_[i];

num infinity = constant('BIG');
um num excess = sum {i in NODES} _supdem_[i];
if (excess > 0) then do;
    /* change equality constraint to le constraint for supply nodes */
    for {i in NODES: _supdem_[i] > 0} balance[i].lb = -infinity;
end;
else if (excess < 0) then do;
    /* change equality constraint to ge constraint for demand nodes */
    for {i in NODES: _supdem_[i] < 0} balance[i].ub = infinity;
end;
solve;

num _supply_ {<i,j> in ARCS} =
    (if _supdem_[i] ne 0 then _supdem_[i] else .);
um _demand_ {<i,j> in ARCS} =
    (if _supdem_[j] ne 0 then -_supdem_[j] else .);
um _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;

create data arc1 from [._tail_._head_]
    _cost_ _capac_ _lo_ _name_ _supply_ _demand_ _flow_=Flow _fcost_
    _rcost_ =
        (if Flow[_tail_,_head_].rc ne 0 then Flow[_tail_,_head_].rc else .)
    _status_ = Flow.status diagonal factory key_id mth_made;
create data node2 from [._node_]
    _supdem_ = (if _supdem_[._node_] ne 0 then _supdem_[._node_] else .)
    _dual_ = balance.dual;
quit;
Example 12.8: Migration to OPTMODEL: Production, Inventory, Distribution

The PROC OPTMODEL statements use both single-dimensional (NODES) and multiple-dimensional (ARCS) index sets, which are populated from the corresponding data set variables in the READ DATA statements. The _SUPDEM_, _LO_, and _CAPAC_ parameters are given initial values, and the NOMISS option in the READ DATA statement tells PROC OPTMODEL to read only the nonmissing values from the input data set. The balance constraint is initially declared as an equality, but depending on the total supply or demand, the sense of this constraint is changed to “≤” or “≥” by relaxing the constraint’s lower or upper bound, respectively. The ARC1 output data set contains most of the same information as in the NETFLOW example, including reduced cost, basis status, and dual values. The _ANUMB_ and _TNUMB_ values do not apply here.

The PROC PRINT statements are similar to the PROC NETFLOW example:

```plaintext
options ls=80 ps=54;
proc print data=arc1 heading=h width=min;
   var _tail_ _head_ _cost_ _capac_ _lo_ _name_ _supply_ _demand_ _flow_ _fcost_
   sum _fcost_; run;
proc print data=arc1 heading=h width=min;
   var _rcost_ _status_ diagonal factory key_id mth_made;
run;
proc print data=node2;
run;
```

The output data sets are displayed in Output 12.8.1.
### Output 12.8.1 Output Data Sets

<table>
<thead>
<tr>
<th>Obs</th>
<th>tail</th>
<th>head</th>
<th>cost</th>
<th>capac</th>
<th>lo</th>
<th>name</th>
<th>supply</th>
<th>demand</th>
<th>flow</th>
<th>fcost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fact1</td>
<td>f1_mar</td>
<td>127.90</td>
<td>500</td>
<td>50</td>
<td>prod f1 19 mar</td>
<td>1000</td>
<td>.</td>
<td>345</td>
<td>44125.50</td>
</tr>
<tr>
<td>2</td>
<td>fact1</td>
<td>f1_apr</td>
<td>78.60</td>
<td>600</td>
<td>50</td>
<td>prod f1 19 apr</td>
<td>1000</td>
<td>.</td>
<td>600</td>
<td>47160.00</td>
</tr>
<tr>
<td>3</td>
<td>fact1</td>
<td>f1_may</td>
<td>95.10</td>
<td>400</td>
<td>50</td>
<td>prod f1 19 may</td>
<td>1000</td>
<td>.</td>
<td>50</td>
<td>4755.00</td>
</tr>
<tr>
<td>4</td>
<td>f1_mar</td>
<td>f1_apr</td>
<td>12.00</td>
<td>50</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>f1_apr</td>
<td>f1_may</td>
<td>8.00</td>
<td>50</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>f1_apr</td>
<td>f1_mar</td>
<td>28.00</td>
<td>20</td>
<td>0</td>
<td>back f1 19 apr</td>
<td>.</td>
<td>20</td>
<td>560.00</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>f1_may</td>
<td>f1_apr</td>
<td>0.00</td>
<td>20</td>
<td>0</td>
<td>back f1 19 may</td>
<td>.</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>f1_mar</td>
<td>f2_mar</td>
<td>11.00</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>f1_apr</td>
<td>f2_apr</td>
<td>11.00</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>30</td>
<td>330.00</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>f1_may</td>
<td>f2_may</td>
<td>16.00</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>100</td>
<td>1600.00</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>f1_mar</td>
<td>shop1</td>
<td>-327.65</td>
<td>250</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>155</td>
<td>-50785.75</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>f1_apr</td>
<td>shop1</td>
<td>-300.00</td>
<td>250</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>250</td>
<td>-75000.00</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>f1_may</td>
<td>shop1</td>
<td>-285.00</td>
<td>250</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>f1_mar</td>
<td>shop2</td>
<td>-362.74</td>
<td>250</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>250</td>
<td>-90685.00</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>f1_apr</td>
<td>shop2</td>
<td>-300.00</td>
<td>250</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>250</td>
<td>-75000.00</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>f1_may</td>
<td>shop2</td>
<td>-245.00</td>
<td>250</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>fact2</td>
<td>f2_mar</td>
<td>88.00</td>
<td>450</td>
<td>35</td>
<td>prod f2 19 mar</td>
<td>850</td>
<td>.</td>
<td>290</td>
<td>25520.00</td>
</tr>
<tr>
<td>18</td>
<td>fact2</td>
<td>f2_apr</td>
<td>62.40</td>
<td>480</td>
<td>35</td>
<td>prod f2 19 apr</td>
<td>850</td>
<td>.</td>
<td>480</td>
<td>29952.00</td>
</tr>
<tr>
<td>19</td>
<td>fact2</td>
<td>f2_may</td>
<td>133.80</td>
<td>250</td>
<td>35</td>
<td>prod f2 19 may</td>
<td>850</td>
<td>.</td>
<td>35</td>
<td>4683.00</td>
</tr>
<tr>
<td>20</td>
<td>f2_mar</td>
<td>f2_apr</td>
<td>18.00</td>
<td>30</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>f2_apr</td>
<td>f2_may</td>
<td>20.00</td>
<td>30</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>15</td>
<td>300.00</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>f2_apr</td>
<td>f2_mar</td>
<td>17.00</td>
<td>15</td>
<td>0</td>
<td>back f2 19 apr</td>
<td>.</td>
<td>.</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>f2_may</td>
<td>f2_apr</td>
<td>25.00</td>
<td>15</td>
<td>0</td>
<td>back f2 19 may</td>
<td>.</td>
<td>.</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>f2_mar</td>
<td>f1_mar</td>
<td>10.00</td>
<td>40</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>40</td>
<td>400.00</td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>f2_apr</td>
<td>f1_apr</td>
<td>11.00</td>
<td>40</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>f2_may</td>
<td>f1_may</td>
<td>13.00</td>
<td>40</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>f2_mar</td>
<td>shop1</td>
<td>-297.40</td>
<td>250</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>250</td>
<td>-74350.00</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>f2_apr</td>
<td>shop1</td>
<td>-290.00</td>
<td>250</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>245</td>
<td>-71050.00</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>f2_may</td>
<td>shop1</td>
<td>-292.00</td>
<td>250</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>f2_mar</td>
<td>shop2</td>
<td>-272.70</td>
<td>250</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>f2_apr</td>
<td>shop2</td>
<td>-312.00</td>
<td>250</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>250</td>
<td>-78000.00</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>f2_may</td>
<td>shop2</td>
<td>-299.00</td>
<td>250</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>150</td>
<td>-44850.00</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>fact2</td>
<td>f1_mar</td>
<td>217.90</td>
<td>400</td>
<td>40</td>
<td>prod f1 25 mar</td>
<td>1000</td>
<td>.</td>
<td>400</td>
<td>87160.00</td>
</tr>
<tr>
<td>34</td>
<td>fact2</td>
<td>f1_apr</td>
<td>174.50</td>
<td>550</td>
<td>50</td>
<td>prod f1 25 apr</td>
<td>1000</td>
<td>.</td>
<td>550</td>
<td>95975.00</td>
</tr>
<tr>
<td>35</td>
<td>fact2</td>
<td>f1_may</td>
<td>133.30</td>
<td>350</td>
<td>40</td>
<td>.</td>
<td>1000</td>
<td>.</td>
<td>40</td>
<td>5332.00</td>
</tr>
<tr>
<td>36</td>
<td>f1_mar</td>
<td>f1_apr</td>
<td>0.00</td>
<td>40</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>f1_apr</td>
<td>f1_may</td>
<td>0.00</td>
<td>40</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>38</td>
<td>f1_apr</td>
<td>f1_mar</td>
<td>0.00</td>
<td>30</td>
<td>0</td>
<td>back f1 25 apr</td>
<td>.</td>
<td>30</td>
<td>960.00</td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>f1_may</td>
<td>f1_apr</td>
<td>41.00</td>
<td>15</td>
<td>0</td>
<td>back f1 25 may</td>
<td>.</td>
<td>15</td>
<td>615.00</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>f1_mar</td>
<td>f2_mar</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>f1_apr</td>
<td>f2_apr</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>f1_may</td>
<td>f2_may</td>
<td>0.00</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>f1_mar</td>
<td>shop1</td>
<td>-559.76</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>f1_apr</td>
<td>shop1</td>
<td>-524.28</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>45</td>
<td>f1_may</td>
<td>shop1</td>
<td>-475.02</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>25</td>
<td>-11875.50</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>f1_mar</td>
<td>shop2</td>
<td>-623.89</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>1450</td>
<td>455</td>
<td>-283869.95</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>f1_apr</td>
<td>shop2</td>
<td>-549.68</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>1450</td>
<td>535</td>
<td>-294078.80</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>f1_may</td>
<td>shop2</td>
<td>-460.00</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>1450</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
</tbody>
</table>
Output 12.8.1 continued

<table>
<thead>
<tr>
<th>Obs</th>
<th>tail</th>
<th>head</th>
<th>cost</th>
<th>capac</th>
<th>lo</th>
<th>name</th>
<th>supply</th>
<th>demand</th>
<th>flow</th>
<th>fcost</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>fact2_2</td>
<td>f2_mar_2</td>
<td>182.00</td>
<td>650</td>
<td>35</td>
<td>prod f2 25 mar</td>
<td>1500</td>
<td>.</td>
<td>645</td>
<td>117390.00</td>
</tr>
<tr>
<td>50</td>
<td>fact2_2</td>
<td>f2_apr_2</td>
<td>196.70</td>
<td>680</td>
<td>35</td>
<td>prod f2 25 aprl</td>
<td>1500</td>
<td>.</td>
<td>680</td>
<td>133756.00</td>
</tr>
<tr>
<td>51</td>
<td>fact2_2</td>
<td>f2_may_2</td>
<td>201.40</td>
<td>550</td>
<td>35</td>
<td>Prod f2 25 may</td>
<td>1500</td>
<td>.</td>
<td>35</td>
<td>7049.00</td>
</tr>
<tr>
<td>52</td>
<td>f2_may_2</td>
<td>f2_mar_2</td>
<td>28.00</td>
<td>50</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>53</td>
<td>f2_apr_2</td>
<td>f2_may_2</td>
<td>38.00</td>
<td>50</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>54</td>
<td>f2_apr_2</td>
<td>f2_may_2</td>
<td>31.00</td>
<td>15</td>
<td>0</td>
<td>back f2 25 aprl</td>
<td>.</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>55</td>
<td>f2_may_2</td>
<td>f2_apr_2</td>
<td>54.00</td>
<td>15</td>
<td>0</td>
<td>back f2 25 may</td>
<td>.</td>
<td>15</td>
<td>810</td>
<td>810.00</td>
</tr>
<tr>
<td>56</td>
<td>f2_may_2</td>
<td>f1_mar_2</td>
<td>20.00</td>
<td>25</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>25</td>
<td>500</td>
<td>500.00</td>
</tr>
<tr>
<td>57</td>
<td>f2_apr_2</td>
<td>f1_apr_2</td>
<td>21.00</td>
<td>25</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>58</td>
<td>f2_may_2</td>
<td>f1_may_2</td>
<td>43.00</td>
<td>25</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0</td>
<td>0.00</td>
</tr>
<tr>
<td>59</td>
<td>f2_may_2</td>
<td>shop1_2</td>
<td>-567.83</td>
<td>500</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>500</td>
<td>-283915.00</td>
<td></td>
</tr>
<tr>
<td>60</td>
<td>f2_apr_2</td>
<td>shop1_2</td>
<td>-542.19</td>
<td>500</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>375</td>
<td>-203321.25</td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>f2_may_2</td>
<td>shop1_2</td>
<td>-461.56</td>
<td>500</td>
<td>0</td>
<td>.</td>
<td>900</td>
<td>0</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>f2_may_2</td>
<td>shop2_2</td>
<td>-542.83</td>
<td>500</td>
<td>0</td>
<td>.</td>
<td>1450</td>
<td>120</td>
<td>-65139.60</td>
<td></td>
</tr>
<tr>
<td>63</td>
<td>f2_apr_2</td>
<td>shop2_2</td>
<td>-559.19</td>
<td>500</td>
<td>0</td>
<td>.</td>
<td>1450</td>
<td>320</td>
<td>-178940.80</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>f2_may_2</td>
<td>shop2_2</td>
<td>-489.06</td>
<td>500</td>
<td>0</td>
<td>.</td>
<td>1450</td>
<td>20</td>
<td>-9781.20</td>
<td></td>
</tr>
</tbody>
</table>

-1281110.35
Chapter 12: The Linear Programming Solver

Output 12.8.1  continued

<table>
<thead>
<tr>
<th>Obs</th>
<th><em>rcost</em></th>
<th><em>status</em></th>
<th>diagonal</th>
<th>factory</th>
<th>key_id</th>
<th>mth_made</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>. B</td>
<td>19</td>
<td>1</td>
<td>production</td>
<td>March</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-0.65</td>
<td>U</td>
<td>19</td>
<td>1</td>
<td>production</td>
<td>April</td>
</tr>
<tr>
<td>3</td>
<td>0.85</td>
<td>L</td>
<td>19</td>
<td>1</td>
<td>production</td>
<td>May</td>
</tr>
<tr>
<td>4</td>
<td>63.65</td>
<td>L</td>
<td>19</td>
<td>1</td>
<td>storage</td>
<td>March</td>
</tr>
<tr>
<td>5</td>
<td>-3.00</td>
<td>U</td>
<td>19</td>
<td>1</td>
<td>storage</td>
<td>April</td>
</tr>
<tr>
<td>6</td>
<td>-20.65</td>
<td>U</td>
<td>19</td>
<td>1</td>
<td>backorder</td>
<td>April</td>
</tr>
<tr>
<td>7</td>
<td>43.00</td>
<td>L</td>
<td>19</td>
<td>1</td>
<td>backorder</td>
<td>May</td>
</tr>
<tr>
<td>8</td>
<td>50.90</td>
<td>L</td>
<td>19</td>
<td>.</td>
<td>f1_to_2</td>
<td>March</td>
</tr>
<tr>
<td>9</td>
<td>. B</td>
<td>19</td>
<td>.</td>
<td>f1_to_2</td>
<td>April</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>. B</td>
<td>19</td>
<td>.</td>
<td>f1_to_2</td>
<td>May</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>. B</td>
<td>19</td>
<td>1</td>
<td>sales</td>
<td>March</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>-21.00</td>
<td>U</td>
<td>19</td>
<td>1</td>
<td>sales</td>
<td>April</td>
</tr>
<tr>
<td>13</td>
<td>9.00</td>
<td>L</td>
<td>19</td>
<td>1</td>
<td>sales</td>
<td>May</td>
</tr>
<tr>
<td>14</td>
<td>-46.09</td>
<td>U</td>
<td>19</td>
<td>1</td>
<td>sales</td>
<td>March</td>
</tr>
<tr>
<td>15</td>
<td>-32.00</td>
<td>U</td>
<td>19</td>
<td>1</td>
<td>sales</td>
<td>April</td>
</tr>
<tr>
<td>16</td>
<td>38.00</td>
<td>L</td>
<td>19</td>
<td>1</td>
<td>sales</td>
<td>May</td>
</tr>
<tr>
<td>17</td>
<td>. B</td>
<td>19</td>
<td>1</td>
<td>2</td>
<td>production</td>
<td>March</td>
</tr>
<tr>
<td>18</td>
<td>-27.85</td>
<td>U</td>
<td>19</td>
<td>2</td>
<td>production</td>
<td>April</td>
</tr>
<tr>
<td>19</td>
<td>23.55</td>
<td>L</td>
<td>19</td>
<td>2</td>
<td>production</td>
<td>May</td>
</tr>
<tr>
<td>20</td>
<td>15.75</td>
<td>L</td>
<td>19</td>
<td>2</td>
<td>storage</td>
<td>March</td>
</tr>
<tr>
<td>21</td>
<td>. B</td>
<td>19</td>
<td>2</td>
<td>storage</td>
<td>April</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>19.25</td>
<td>L</td>
<td>19</td>
<td>2</td>
<td>backorder</td>
<td>April</td>
</tr>
<tr>
<td>23</td>
<td>45.00</td>
<td>L</td>
<td>19</td>
<td>2</td>
<td>backorder</td>
<td>May</td>
</tr>
<tr>
<td>24</td>
<td>-29.90</td>
<td>U</td>
<td>19</td>
<td>.</td>
<td>f2_to_1</td>
<td>March</td>
</tr>
<tr>
<td>25</td>
<td>22.00</td>
<td>L</td>
<td>19</td>
<td>.</td>
<td>f2_to_1</td>
<td>April</td>
</tr>
<tr>
<td>26</td>
<td>29.00</td>
<td>L</td>
<td>19</td>
<td>.</td>
<td>f2_to_1</td>
<td>May</td>
</tr>
<tr>
<td>27</td>
<td>-9.65</td>
<td>U</td>
<td>19</td>
<td>2</td>
<td>sales</td>
<td>March</td>
</tr>
<tr>
<td>28</td>
<td>. B</td>
<td>19</td>
<td>2</td>
<td>sales</td>
<td>April</td>
<td></td>
</tr>
<tr>
<td>29</td>
<td>18.00</td>
<td>L</td>
<td>19</td>
<td>2</td>
<td>sales</td>
<td>May</td>
</tr>
<tr>
<td>30</td>
<td>4.05</td>
<td>L</td>
<td>19</td>
<td>2</td>
<td>sales</td>
<td>March</td>
</tr>
<tr>
<td>31</td>
<td>-33.00</td>
<td>U</td>
<td>19</td>
<td>2</td>
<td>sales</td>
<td>April</td>
</tr>
<tr>
<td>32</td>
<td>. B</td>
<td>19</td>
<td>2</td>
<td>sales</td>
<td>May</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>-45.16</td>
<td>U</td>
<td>25</td>
<td>1</td>
<td>production</td>
<td>March</td>
</tr>
<tr>
<td>34</td>
<td>-14.35</td>
<td>U</td>
<td>25</td>
<td>1</td>
<td>production</td>
<td>April</td>
</tr>
<tr>
<td>35</td>
<td>2.11</td>
<td>L</td>
<td>25</td>
<td>1</td>
<td>production</td>
<td>May</td>
</tr>
<tr>
<td>36</td>
<td>94.21</td>
<td>L</td>
<td>25</td>
<td>1</td>
<td>storage</td>
<td>March</td>
</tr>
<tr>
<td>37</td>
<td>75.66</td>
<td>L</td>
<td>25</td>
<td>1</td>
<td>storage</td>
<td>April</td>
</tr>
<tr>
<td>38</td>
<td>-42.21</td>
<td>U</td>
<td>25</td>
<td>1</td>
<td>backorder</td>
<td>April</td>
</tr>
<tr>
<td>39</td>
<td>-16.66</td>
<td>U</td>
<td>25</td>
<td>1</td>
<td>backorder</td>
<td>May</td>
</tr>
<tr>
<td>40</td>
<td>104.06</td>
<td>L</td>
<td>25</td>
<td>.</td>
<td>f1_to_2</td>
<td>March</td>
</tr>
<tr>
<td>41</td>
<td>13.49</td>
<td>L</td>
<td>25</td>
<td>.</td>
<td>f1_to_2</td>
<td>April</td>
</tr>
<tr>
<td>42</td>
<td>28.96</td>
<td>L</td>
<td>25</td>
<td>.</td>
<td>f1_to_2</td>
<td>May</td>
</tr>
<tr>
<td>43</td>
<td>47.13</td>
<td>L</td>
<td>25</td>
<td>1</td>
<td>sales</td>
<td>March</td>
</tr>
<tr>
<td>44</td>
<td>8.40</td>
<td>L</td>
<td>25</td>
<td>1</td>
<td>sales</td>
<td>April</td>
</tr>
<tr>
<td>45</td>
<td>. B</td>
<td>25</td>
<td>1</td>
<td>sales</td>
<td>May</td>
<td></td>
</tr>
<tr>
<td>46</td>
<td>. B</td>
<td>25</td>
<td>1</td>
<td>sales</td>
<td>March</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>. B</td>
<td>25</td>
<td>1</td>
<td>sales</td>
<td>April</td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>32.02</td>
<td>L</td>
<td>25</td>
<td>1</td>
<td>sales</td>
<td>May</td>
</tr>
</tbody>
</table>
Example 12.8: Migration to OPTMODEL: Production, Inventory, Distribution

Output 12.8.1 continued

<table>
<thead>
<tr>
<th>Obs</th>
<th><em>rcost</em></th>
<th><em>status</em></th>
<th>diagonal</th>
<th>factory</th>
<th>key_id</th>
<th>mth_made</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>.</td>
<td>B</td>
<td>25</td>
<td>2</td>
<td>production</td>
<td>March</td>
</tr>
<tr>
<td>50</td>
<td>-1.66</td>
<td>U</td>
<td>25</td>
<td>2</td>
<td>production</td>
<td>April</td>
</tr>
<tr>
<td>51</td>
<td>73.17</td>
<td>L</td>
<td>25</td>
<td>2</td>
<td>production</td>
<td>May</td>
</tr>
<tr>
<td>52</td>
<td>11.64</td>
<td>L</td>
<td>25</td>
<td>2</td>
<td>storage</td>
<td>March</td>
</tr>
<tr>
<td>53</td>
<td>108.13</td>
<td>L</td>
<td>25</td>
<td>2</td>
<td>storage</td>
<td>April</td>
</tr>
<tr>
<td>54</td>
<td>47.36</td>
<td>L</td>
<td>25</td>
<td>2</td>
<td>backorder</td>
<td>April</td>
</tr>
<tr>
<td>55</td>
<td>-16.13</td>
<td>U</td>
<td>25</td>
<td>2</td>
<td>backorder</td>
<td>May</td>
</tr>
<tr>
<td>56</td>
<td>-61.06</td>
<td>U</td>
<td>25</td>
<td>.</td>
<td>f2_to_1</td>
<td>March</td>
</tr>
<tr>
<td>57</td>
<td>30.51</td>
<td>L</td>
<td>25</td>
<td>.</td>
<td>f2_to_1</td>
<td>April</td>
</tr>
<tr>
<td>58</td>
<td>40.04</td>
<td>L</td>
<td>25</td>
<td>.</td>
<td>f2_to_1</td>
<td>May</td>
</tr>
<tr>
<td>59</td>
<td>-42.00</td>
<td>U</td>
<td>25</td>
<td>2</td>
<td>sales</td>
<td>March</td>
</tr>
<tr>
<td>60</td>
<td>.</td>
<td>B</td>
<td>25</td>
<td>2</td>
<td>sales</td>
<td>April</td>
</tr>
<tr>
<td>61</td>
<td>10.50</td>
<td>L</td>
<td>25</td>
<td>2</td>
<td>sales</td>
<td>May</td>
</tr>
<tr>
<td>62</td>
<td>.</td>
<td>B</td>
<td>25</td>
<td>2</td>
<td>sales</td>
<td>March</td>
</tr>
<tr>
<td>63</td>
<td>.</td>
<td>B</td>
<td>25</td>
<td>2</td>
<td>sales</td>
<td>April</td>
</tr>
<tr>
<td>64</td>
<td>.</td>
<td>B</td>
<td>25</td>
<td>2</td>
<td>sales</td>
<td>May</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obs</th>
<th><em>node</em></th>
<th><em>supdem</em></th>
<th><em>dual</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>fact1_1</td>
<td>1000</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>fact2_1</td>
<td>850</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td>fact1_2</td>
<td>1000</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>fact2_2</td>
<td>1500</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>shop1_1</td>
<td>-900</td>
<td>199.75</td>
</tr>
<tr>
<td>6</td>
<td>shop2_1</td>
<td>-900</td>
<td>188.75</td>
</tr>
<tr>
<td>7</td>
<td>shop1_2</td>
<td>-900</td>
<td>343.83</td>
</tr>
<tr>
<td>8</td>
<td>shop2_2</td>
<td>-1450</td>
<td>360.83</td>
</tr>
<tr>
<td>9</td>
<td>f1_mar_1</td>
<td>.</td>
<td>-127.90</td>
</tr>
<tr>
<td>10</td>
<td>f1_apr_1</td>
<td>.</td>
<td>-79.25</td>
</tr>
<tr>
<td>11</td>
<td>f1_may_1</td>
<td>.</td>
<td>-94.25</td>
</tr>
<tr>
<td>12</td>
<td>f2_mar_1</td>
<td>.</td>
<td>-88.00</td>
</tr>
<tr>
<td>13</td>
<td>f2_apr_1</td>
<td>.</td>
<td>-90.25</td>
</tr>
<tr>
<td>14</td>
<td>f2_may_1</td>
<td>.</td>
<td>-110.25</td>
</tr>
<tr>
<td>15</td>
<td>f1_mar_2</td>
<td>.</td>
<td>-263.06</td>
</tr>
<tr>
<td>16</td>
<td>f1_apr_2</td>
<td>.</td>
<td>-188.85</td>
</tr>
<tr>
<td>17</td>
<td>f1_may_2</td>
<td>.</td>
<td>-131.19</td>
</tr>
<tr>
<td>18</td>
<td>f2_mar_2</td>
<td>.</td>
<td>-182.00</td>
</tr>
<tr>
<td>19</td>
<td>f2_apr_2</td>
<td>.</td>
<td>-198.36</td>
</tr>
<tr>
<td>20</td>
<td>f2_may_2</td>
<td>.</td>
<td>-128.23</td>
</tr>
</tbody>
</table>

The log is displayed in Output 12.8.2.
Example 12.9: Migration to OPTMODEL: Shortest Path

The following example shows how to use PROC OPTMODEL to solve the example “Shortest Path Problem” in Chapter 5, “The NETFLOW Procedure” (SAS/OR User’s Guide: Mathematical Programming Legacy Procedures). The input data set is the same as in that example.

```
title 'Shortest Path Problem';
title2 'How to get Hawaiian Pineapples to a London Restaurant';

data aircost1;
  input ffrom&$13. tto&$15. _cost_;
datalines;
Honolulu  Chicago 105
Honolulu  San Francisco 75
Honolulu  Los Angeles  68
Chicago    Boston  45
Chicago    New York  56
San Francisco Boston  71
San Francisco New York 48
San Francisco Atlanta 63
Los Angeles New York  44
Los Angeles Atlanta  57
Boston     Heathrow London 88
New York   Heathrow London 65
Atlanta    Heathrow London 76
;```
The following PROC OPTMODEL statements read the data sets, build the linear programming model, solve the model, and output the optimal solution to a SAS data set called SPATH:

```sas
proc optmodel;
  str sourcenode = 'Honolulu';
  str sinknode = 'Heathrow London';

  set <str> NODES;
  num _supdem_ {i in NODES} = (if i = sourcenode then 1
    else if i = sinknode then -1 else 0);

  set <str,str> ARCS;
  num _lo_ {ARCS} init 0;
  num _capac_ {ARCS} init .;
  num _cost_ {ARCS};
  read data aircost1 into ARCS=\(\text{ffrom tto}\) _cost_;
  NODES = (union {<i,j> in ARCS} {i,j});

  var Flow {<i,j> in ARCS} >= _lo_[i,j];
  min obj = sum {<i,j> in ARCS} _cost_[i,j] * Flow[i,j];
  con balance {i in NODES}: sum {<(i),j> in ARCS} Flow[i,j]
    - sum {<j,(i)> in ARCS} Flow[j,i] = _supdem_[i];
  solve;

  num _supply_ {<i,j> in ARCS} =
    (if _supdem_[i] ne 0 then _supdem_[i] else .);
  num _demand_ {<i,j> in ARCS} =
    (if _supdem_[j] ne 0 then -_supdem_[j] else .);
  num _fcost_ {<i,j> in ARCS} = _cost_[i,j] * Flow[i,j].sol;

  create data spath from \(\text{ffrom tto}\)
    _cost_ _capac_ _lo_ _supply_ _demand_ _flow_=Flow _fcost_ _rcost_=(if Flow[ffrom,tto].rc ne 0 then Flow[ffrom,tto].rc else .)
    _status_=Flow.status;
quit;
```

The statements use both single-dimensional (NODES) and multiple-dimensional (ARCS) index sets. The ARCS index set is populated from the ffrom and tto data set variables in the READ DATA statement. To solve a shortest path problem, you solve a minimum-cost network flow problem that has a supply of one unit at the source node, a demand of one unit at the sink node, and zero supply or demand at all other nodes, as specified in the declaration of the _SUPDEM_ numeric parameter. The SPATH output data set contains most of the same information as in the PROC NETFLOW example, including reduced cost and basis status. The _ANUMB_ and _TNUMB_ values do not apply here.
The PROC PRINT statements are similar to the PROC NETFLOW example:

```plaintext
proc print data=spath;
  sum _fcost_;
run;
```

The output is displayed in Output 12.9.1.

**Output 12.9.1 Output Data Set**

<table>
<thead>
<tr>
<th>Obs</th>
<th>from</th>
<th>to</th>
<th><em>cost</em></th>
<th><em>capac</em></th>
<th><em>lo</em></th>
<th><em>supply</em></th>
<th><em>demand</em></th>
<th><em>flow</em></th>
<th><em>fcost</em></th>
<th><em>rcost</em></th>
<th><em>status</em></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Honolulu</td>
<td>Chicago</td>
<td>105</td>
<td>.</td>
<td>0</td>
<td>1</td>
<td>.</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>Honolulu</td>
<td>San Francisco</td>
<td>75</td>
<td>.</td>
<td>0</td>
<td>1</td>
<td>.</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>B</td>
</tr>
<tr>
<td>3</td>
<td>Honolulu</td>
<td>Los Angeles</td>
<td>68</td>
<td>.</td>
<td>0</td>
<td>1</td>
<td>.</td>
<td>1</td>
<td>68</td>
<td>.</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>Chicago</td>
<td>Boston</td>
<td>45</td>
<td>.</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0</td>
<td>61</td>
<td>L</td>
</tr>
<tr>
<td>5</td>
<td>Chicago</td>
<td>New York</td>
<td>56</td>
<td>.</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0</td>
<td>49</td>
<td>L</td>
</tr>
<tr>
<td>6</td>
<td>San Francisco</td>
<td>Boston</td>
<td>71</td>
<td>.</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0</td>
<td>57</td>
<td>L</td>
</tr>
<tr>
<td>7</td>
<td>San Francisco</td>
<td>New York</td>
<td>48</td>
<td>.</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>L</td>
</tr>
<tr>
<td>8</td>
<td>San Francisco</td>
<td>Atlanta</td>
<td>63</td>
<td>.</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0</td>
<td>37</td>
<td>L</td>
</tr>
<tr>
<td>9</td>
<td>Los Angeles</td>
<td>New York</td>
<td>44</td>
<td>.</td>
<td>0</td>
<td>.</td>
<td>1</td>
<td>44</td>
<td>.</td>
<td></td>
<td>B</td>
</tr>
<tr>
<td>10</td>
<td>Los Angeles</td>
<td>Atlanta</td>
<td>57</td>
<td>.</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>0</td>
<td>0</td>
<td>24</td>
<td>L</td>
</tr>
<tr>
<td>11</td>
<td>Boston</td>
<td>Heathrow London</td>
<td>88</td>
<td>.</td>
<td>0</td>
<td>.</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>B</td>
</tr>
<tr>
<td>12</td>
<td>New York</td>
<td>Heathrow London</td>
<td>65</td>
<td>.</td>
<td>0</td>
<td>.</td>
<td>1</td>
<td>1</td>
<td>65</td>
<td>.</td>
<td>B</td>
</tr>
<tr>
<td>13</td>
<td>Atlanta</td>
<td>Heathrow London</td>
<td>76</td>
<td>.</td>
<td>0</td>
<td>.</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>B</td>
</tr>
</tbody>
</table>

The log is displayed in Output 12.9.2.

**Output 12.9.2 OPTMODEL Log**

NOTE: There were 13 observations read from the data set WORK.AIRCOST1.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 13 variables (0 free, 0 fixed).
NOTE: The problem has 8 linear constraints (0 LE, 8 EQ, 0 GE, 0 range).
NOTE: The problem has 26 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The problem is a pure network instance. The ALGORITHM=NETWORK option is recommended for solving problems with this structure.
NOTE: The LP presolver value AUTOMATIC is applied.
NOTE: The LP presolver removed all variables and constraints.
NOTE: Optimal.
NOTE: Objective = 177.
NOTE: The data set WORK.SPATH has 13 observations and 11 variables.
References


Chapter 13
The Local Search Optimization Solver

Overview: LSO Solver

The local search optimization (LSO) solver, which you can access through PROC OPTMODEL, provides a framework for specifying optimization problems and solving them by using derivative-free methods. It does this by optimizing general nonlinear functions over both continuous and integer variables. You do not need to express these functions in analytic closed form, and they can be non-smooth, discontinuous, and computationally expensive to evaluate. Problem types can be single-objective or multiobjective. The LSO solver runs in either single-machine mode or distributed mode.
The general problem formulation is given by

\[
\min_x f(x) \\
\text{subject to} \\
x_l \leq x \leq x_u \\
b_l \leq Ax \leq b_u \\
c_l \leq c(x) \leq c_u \\
x_i \in \mathbb{Z}, \; i \in I
\]

where \( x \in \mathbb{R}^n \) is the vector of the decision variables; \( f(x) : \mathbb{R}^n \rightarrow \mathbb{R} \) is the objective function; \( A \) is an \( m \times n \) linear coefficient matrix; \( c(x) : \mathbb{R}^n \rightarrow \mathbb{R}^p \) is the vector of general nonlinear constraint functions—that is, \( c = (c_1, \ldots, c_p) \); \( x_l \) and \( x_u \) are the vectors of the lower and upper bounds, respectively, on the decision variables; \( b_l \) and \( b_u \) are the vectors of the lower and upper bounds, respectively, on the linear constraints; and \( c_l \) and \( c_u \) are the vectors of the lower and upper bounds, respectively, on the nonlinear constraint functions. Equality constraints can be represented by equating the lower and upper bounds of the desired variable or constraint.

Because of the limited assumptions that are made about the objective function \( f(x) \) and constraint functions \( c(x) \), the LSO solver uses a parallel hybrid derivative-free approach similar to approaches that are used in Taddy et al. (2009), Plantenga (2009), Gray, Fowler, and Griffin (2010), and Griffin and Kolda (2010a). Derivative-free methods are effective whether or not derivatives are available, provided that the dimension of \( x \) is not too large (Gray and Fowler 2011). As a rule of thumb, derivative-free algorithms are rarely applied to black-box optimization problems that have more than 100 variables. The term black-box emphasizes that the function is used only as a mapping operator and makes no implicit assumption or requirement about the structure of the functions themselves. In contrast, derivative-based algorithms commonly require the nonlinear objectives and constraints to be continuous and smooth and to have an exploitable analytic representation.

The LSO solver optimizes general nonlinear problems by simultaneously applying multiple instances of global and local search algorithms in parallel. This hybrid approach streamlines the process of needing to first apply a global algorithm in order to determine a good starting point to initialize a local algorithm. For example, if the problem is convex, a local algorithm should be sufficient, and using a global algorithm would create unnecessary overhead. If the problem instead has many local minima, failing to run a global search algorithm first could result in an inferior solution. Rather than attempting to guess which paradigm is best, the LSO solver simultaneously performs global and local searches while continuously sharing computational resources and function evaluations. Given a suitable number of threads and processors, the resulting run time and solution quality should be similar to those from having automatically selected the best global and local search combination. Moreover, because information is shared, the robustness of this hybrid approach can exceed that of hybrid combinations that simply use the output of one algorithm to hot-start another algorithm.

The LSO solver uses different strategies to handle different types of constraints. It handles linear constraints by using both linear programming and strategies similar to those in Griffin, Kolda, and Lewis (2008), where tangent directions to nearby constraints are constructed and used as search directions. It handles nonlinear constraints by using smooth merit functions (Griffin and Kolda 2010b) and handles integer and categorical variables by using strategies and concepts similar to those in Griffin et al. (2011). Its approach can be viewed
as a genetic algorithm that includes an additional “growth” step, in which selected points from the population are allotted a small fraction of the total evaluation budget to improve their fitness score (that is, the objective function value) by using local optimization over the continuous variables.

Because the LSO solver is a parallel/distributed solver, it also does the following:

- enables you to run the solver in a session on a CAS server; this can be a single machine or a distributed computing environment, depending on the number of computing nodes that the CAS session is running on
- enables you to run in single-machine mode on the server where SAS is installed
- exploits all the available cores and concurrent threads, regardless of execution mode

For more information, see Chapter 4, “Shared Concepts,” for the options available for running in a distributed environment.

## Getting Started: LSO Solver

### Introductory Examples

The following introductory examples illustrate how to get started using the LSO solver.

### A Bound-Constrained Problem

Consider the simple example of minimizing the Branin function,

\[
f(x) = \left( x_2 - \frac{5.1}{4\pi^2} x_1^2 + \frac{5}{\pi} x_1 - 6 \right)^2 + 10 \left( 1 - \frac{1}{8\pi} \right) \cos(x_1) + 10
\]

subject to \(-5 \leq x_1 \leq 10\) and \(0 \leq x_2 \leq 15\) (Jones, Perttunen, and Stuckman 1993). The minimum function value is \(f(x^*) = 0.397887\) at \(x^* = (-\pi, 12.275), (\pi, 2.275), (9.42478, 2.475)\). You can use the following statements to solve this bound-constrained problem:

```plaintext
proc optmodel;
    var x {1..2};
    x[1].lb = -5;
    x[1].ub = 10;
    x[2].lb = 0;
    x[2].ub = 15;
    num pi = constant('PI');
    min f = (x[2]-(5.1/(4*pi^2))*x[1]^2+5*pi/x[1]-6)^2 + 10*(1-1/(8*pi))*cos(x[1]) + 10;
    solve with lso;
    print x;
quit;
```
The VAR declaration in the PROC OPTMODEL statement declares an array of two decision variables, \(x[1]\) and \(x[2]\). Also, lower and upper bounds are set for the variables. The MIN declaration defines the BRANIN function as the objective function to be minimized.

Figure 13.1 shows the ODS tables that are produced by default.

![Figure 13.1](https://example.com/fig13.1)

**The OPTMODEL Procedure**

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Variables</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded Above</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
<td>2</td>
</tr>
<tr>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>Fixed</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Constraints</th>
<th>0</th>
</tr>
</thead>
</table>

| Constraint Coefficients | 0 |

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Infeasibility</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random Seed Used</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Evaluations</th>
<th>1541</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cached Evaluations</td>
<td>48</td>
</tr>
<tr>
<td>Iterations</td>
<td>28</td>
</tr>
<tr>
<td>Presolve Time</td>
<td>0.00</td>
</tr>
<tr>
<td>Solution Time</td>
<td>0.67</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|c}
   \text{[1]} & x \\
   1 & 3.1416 \\
   2 & 2.2750 \\
\end{array}
\]

**Adding a Linear Constraint**

You can use a CONSTRAINT declaration to specify general equality or inequality constraints. For example, suppose that in addition to the bound constraints on the decision variables, you need to guarantee that the sum \(x_1 + x_2\) is less than or equal to 0.6. To enforce this, you can add a CON declaration to the PROC OPTMODEL statements, as follows:
Introductory Examples

/* Linear Constraint on the Decision Variables */
proc optmodel;
  var x {1..2};
x[1].lb = -5;
x[1].ub = 10;
x[2].lb = 0;
x[2].ub = 15;
  num pi = constant('PI');
  min f = (x[2]-(5.1/(4*pi^2))*x[1]^2+5*x[1]/pi-6)^2
       + 10*(1-1/(8*pi))*cos(x[1]) + 10;
  con a1: x[1] + x[2] <= 0.6;
solve with lso;
quit;

Here the symbol a1 denotes the name of the given linear constraint.

Nonlinear Constraints on the Decision Variables

You can specify general nonlinear equality or inequality constraints by using the same CON declaration as in the preceding code to add the linear constraint a1. Consider the previous problem with the following additional constraint:

\[ x_1^2 - 2x_2 \geq 14 \]

You can specify this constraint by adding a new CON declaration, as in the following statements:

/* Nonlinear Constraints on the Decision Variables */
proc optmodel;
  var x {1..2};
x[1].lb = -5;
x[1].ub = 10;
x[2].lb = 0;
x[2].ub = 15;
  num pi = constant('PI');
  min f = (x[2]-(5.1/(4*pi^2))*x[1]^2+5*x[1]/pi-6)^2
       + 10*(1-1/(8*pi))*cos(x[1]) + 10;
  con a1: x[1] + x[2] <= 0.6;
solve with lso;
quit;

Here the symbol c1 has been added to denote the name of the given nonlinear constraint.

A Simple Maximum Likelihood Example

The following is a very simple example of a maximum likelihood estimation problem that uses the log-likelihood function:

\[ l(\mu, \sigma) = -\log(\sigma) - \frac{1}{2} \left( \frac{x - \mu}{\sigma} \right)^2 \]

The maximum likelihood estimates of the parameters \( \mu \) and \( \sigma \) form the solution to

\[ \max_{\mu, \sigma} \sum_i l_i(\mu, \sigma) \]
where $\sigma > 0$ and

$$l_i(\mu, \sigma) = -\log(\sigma) - \frac{1}{2} \left( \frac{x_i - \mu}{\sigma} \right)^2$$

In addition to specifying objective and constraint functions directly with PROC OPTMODEL syntax as shown in the first example, you can also specify these functions by using the FCMP procedure. In PROC FCMP, you specify the objective and constraint functions by using SAS statements that are similar to syntax in the SAS DATA step; these functions are then compiled into function libraries for subsequent use. The SAS CMPLIB= system option specifies where to look for previously compiled functions and subroutines. All procedures (including PROC FCMP) that support the use of FCMP functions and subroutines use this system option. After you specify your objective and constraint functions in a library, PROC OPTMODEL requires the names and context of the functions within this library that are relevant to the current optimization problem. You can provide this information to PROC OPTMODEL by using the same function names that you used in the PROC FCMP statements.

The following sets of statements use PROC FCMP functions and demonstrate two ways to formulate this example problem. In the first example, the FCMP function is “stand-alone” because all the necessary data are defined within the function itself. Notice how the values of (1, 3, 4, 5, 7) are assigned directly to X within the FCMP function.

```sas
title3 'Maximum Likelihood Example';
/* Maximum Likelihood Example */
proc fcmp outlib=sasuser.myfuncs.mypkg;
    function loglkh1(mu, sigma);
      array x[5] / nosym (1 3 4 5 7);
      s=0;
      do j=1 to 5;
        s = s - log(sigma) - 0.5*((x[j]-mu)/sigma)**2;
      end;
      return (s);
    endsub;
run;

options cmplib = sasuser.myfuncs;
proc optmodel;
    var mu, sigma >= 0;
    max f = loglkh1(mu, sigma);
    solve with lso;
quit;
```

The second example shows how you can use an external data set to store the necessary observations and run PROC OPTMODEL to feed each observation to an FCMP function that processes a single line of data. In this case, the objective definition in PROC OPTMODEL is the sum of the individual observation results. This mode is particularly useful when the data set is large and therefore cumbersome to enter directly into a function definition. The following statements demonstrate how to store the necessary observations in an external data set for use with PROC FCMP:

```sas
data logdata;
    input x @@;
datalines;
1 3 4 5 7
;
```
proc fcmp outlib=sasuser.myfuncs.mypkg;
  function loglkh2(mu, sigma, x);
    return (-log(sigma) -0.5*((x-mu)/sigma)**2);
  endsub;
run;

options cmplib = sasuser.myfuncs;
proc optmodel;
  var mu, sigma >= 0;
  set OBS;
  num x {OBS};
  read data logdata into OBS= [_N_] x;
  max f = sum {j in OBS} loglkh2(mu, sigma, x[j]);
  solve with lso;
quit;

In this case, for each line of data in the data set LogData, the FCMP function LOGLKH2 is called.

Another variation of this same problem involves bypassing the use of PROC FCMP altogether and defining your objective function directly in PROC OPTMODEL as a function of the LogData data read into X:

proc optmodel;
  var mu, sigma >= 0;
  set OBS;
  num x {OBS};
  read data logdata into OBS= [_N_] x;
  max f = sum {j in OBS} (-log(sigma) - 0.5*((x[j]-mu)/sigma)**2);
  solve with lso;
quit;

Syntax: LSO Solver

The following statement is available in the OPTMODEL procedure:

SOLVE WITH LSO < /options > ;

Functional Summary

Table 13.1 outlines the options available in the SOLVE WITH LSO statement.

<table>
<thead>
<tr>
<th>Description</th>
<th>option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Data Option</td>
<td>PRIMALIN</td>
</tr>
<tr>
<td>Specifies whether to use the initial trial points</td>
<td></td>
</tr>
<tr>
<td>Stopping Condition Options</td>
<td>ABSFCONV=</td>
</tr>
<tr>
<td>Specifies the absolute function convergence criterion</td>
<td>NABSFCONV=</td>
</tr>
<tr>
<td>Specifies the number of iterations without improvement for convergence</td>
<td></td>
</tr>
</tbody>
</table>
Table 13.1  continued

<table>
<thead>
<tr>
<th>Description</th>
<th>option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies the maximum number of function evaluations MAXFUNCE=</td>
<td></td>
</tr>
<tr>
<td>Specifies the maximum number of genetic algorithm iterations MAXGENE=</td>
<td></td>
</tr>
<tr>
<td>Specifies the upper limit on real time MAXTIME=</td>
<td></td>
</tr>
</tbody>
</table>

**Optimization Control Options**

- Specifies the feasibility tolerance FEASTOL=
- Specifies the number of global searches NGLOBAL=
- Specifies the number of local searches NLOCAL=
- Specifies the genetic algorithm population size POPSIZE=

**Technical Options**

- Specifies the cache tolerance CACHETOL=
- Specifies the maximum cache size CACHEMAX=
- Specifies the maximum size of the Pareto-optimal set PARETOMAX=
- Specifies how frequently to print the solution progress LOGFREQ=
- Specifies how much detail of solution progress to print in the log LOGLEVEL=
- Initializes the seed for sampling routines SEED=
- Specifies the number of threads for the LSO solver to use per node NTHREADS=

**LSO Solver Options**

This section describes the options that the LSO solver recognizes in PROC OPTMODEL. You can specify these options after a forward slash (/) in the SOLVE statement, provided that you explicitly specify the LSO solver by using a WITH clause. For example, the following line could appear in PROC OPTMODEL statements:

```
solve with lso / maxtime=600 nthreads=4 primalin;
```

**Input Data Options**

You can specify the following options that are related to input data sets:

**PRIMALIN**

requests that an initial sample set be evaluated by the LSO solver. Adding this option to the SOLVE statement requests that the LSO solver use the current variable values as a starting point. This option might be useful over a sequence of runs when you want to ensure that the LSO solver generates points that are at least as good as your current best solution. You can also specify other starting points by setting the _NSOL_ parameter to the number of points that you want to specify and setting the variable .sol[n] values (for n in 1.._NSOL_).
Stopping Condition Options

You can specify the following options to determine when to stop optimization:

\[
\text{ABSFCONV} = r
\]

specifies an absolute function convergence criterion. The LSO solver stops when the changes in the objective function and constraint violation values in successive iterations meet the criterion

\[
|f(x^{(k-1)}) - f(x^{(k)})| + |\theta(x^{(k-1)}) - \theta(x^{(k)})| \leq r
\]

where \( \theta(x) \) denotes the maximum constraint violation at point \( x \). By default, \( r = 1E^{-6} \).

\[
\text{MAXFUNC} = i
\]

specifies the maximum number of function calls in the optimization process. The actual number of function evaluations can exceed this number in order to ensure deterministic results.

\[
\text{MAXGEN} = i
\]

specifies the maximum number of genetic algorithm iterations. By default, MAXGEN=500.

\[
\text{MAXTIME} = r
\]

specifies an upper limit (in seconds) on the real time that the optimization process uses. The actual run time of the LSO solver might be longer, because the actual run time includes the time needed to finish current function evaluations and the time needed to output the (temporary) results. The value of \( r \) can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

\[
\text{NABSFCONV} = i
\]

specifies the number of successive iterations for which the criterion that you specify in the ABSFCONV= option must be satisfied before the optimization process is terminated. By default, NABSFCONV=10. To cause an early exit, you must specify a value for \( i \) that is less than the value of the MAXGEN= option.

Optimization Control Options

You can specify the following options to tailor the LSO solver to your specific optimization problem:

\[
\text{FEASTOL} = r
\]

specifies a feasibility tolerance for the provided constraints. Specify \( r \geq 1E^{-9} \). By default, \( r = 1E^{-3} \).

\[
\text{NGLOBAL} = i
\]

specifies the number of genetic algorithms to create; each algorithm works on a separate population of the size that you specify in the POPSIZE= option. Specify \( i \) as an integer greater than 0. By default, NGLOBAL=1.

\[
\text{NLOCAL} = i
\]

specifies the number of local solvers to create. Specify \( i \) as an integer greater than 0. The default is twice the number of variables in the problem.

\[
\text{POPSIZE} = i
\]

specifies the population size for the genetic algorithm to use. The default is \( 40 \times \text{ceil}(\log(n) + 1) \), where \( n \) denotes the number of variables.
Technical Options

You can specify the following technical options:

**CACHEMAX=i**
specifies the maximum number of points that can be cached. By default, the LSO solver automatically calculates the maximum number of points.

**CACHETOL=r**
specifies the cache tolerance to be used for caching and referencing previously evaluated points. For more information about this tolerance, see the section “Function Value Caching” on page 580. The value of \( r \) can be any number in the interval \([0, 1]\). By default, \( r=1E^{-9} \).

**LOGFREQ=i**
requests that the solution progress be printed to the iteration log after every \( i \) iterations if the value of the LOGLEVEL= option is equal to 1. LOGFREQ=0 disables the printing of the solution progress. The solution progress of the final iteration is always printed if \( i \geq 1 \) and the value of the LOGLEVEL= option is nonzero. By default, LOGFREQ=1.

**LOGLEVEL=0 | 1**
controls how much information is printed to the log. If LOGLEVEL=0, nothing is printed. If LOGLEVEL=1, a short summary of the problem description and final solution status is printed. If LOGLEVEL=0, this option overrides the LOGFREQ= option. By default, LOGLEVEL=1.

**NTHREADS=i**
specifies the maximum number of threads for the LSO solver to use for multithreaded processing. When running in a distributed computing environment, the LSO solver uses up to \( i \) threads per worker node. The default is the value of the NTHREADS= option in PROC OPTMODEL.

**PARETOMAX=i**
specifies the maximum number of points in the Pareto-optimal set, which is produced for multiobjective optimization. By default, PARETOMAX=5000.

**SEED=i**
specifies a nonnegative integer as a seed value for the pseudorandom number generator. Pseudorandom numbers are used within the genetic algorithm. By default, SEED=1.

Details: LSO Solver

The LSO solver uses a hybrid of genetic algorithms (Goldberg 1989; Holland 1975), which optimize integer and continuous variables, and the generating set search (Kolda, Lewis, and Torczon 2003), which performs a local search on the continuous variables to improve the robustness of both algorithms. Both genetic algorithms (GAs) and the generating set search (GSS) have proven to be effective algorithms for many classes of derivative-free optimization problems. When only continuous variables are present, a GA usually requires more function evaluations than a GSS to converge to a minimum. This is partly due to the GA’s need to simultaneously perform a global search of the solution space. In a hybrid setting, the requirement for the GA to find accurate local minima can be relaxed, and internal parameters can be tuned to find promising starting points for the GSS.
The LSO Algorithm

The LSO algorithm is based on a genetic algorithm (GA). GAs are a family of local search algorithms that seek optimal solutions to problems by applying the principles of natural selection and evolution. You can apply genetic algorithms to almost any optimization problem. They are especially useful for problems where calculus-based techniques do not work, such as when the objective function has many local optima, when the objective function is not differentiable or continuous, or when solution elements are constrained to be integers or sequences. In most cases, genetic algorithms require more computation than specialized techniques that take advantage of specific problem structures or characteristics. However, for optimization problems for which no such techniques are available, genetic algorithms provide a robust general method of solution.

In general, genetic algorithms use some variation of the following process to search for an optimal solution:

- **Initialization**: An initial population of solutions is randomly generated, and the objective function is evaluated for each member of this initial iteration.

- **Selection**: Individual members of the current iteration are chosen stochastically either to parent the next iteration or to be passed on to it such that the fittest members are more likely to be selected. A solution’s fitness is based on its objective value; a better objective value reflects greater fitness.

- **Crossover**: Some of the selected solutions are passed to a crossover operator. The crossover operator combines two or more parents to produce new offspring solutions for the next iteration. The crossover operator tends to produce new offspring that retain the common characteristics of the parent solutions while combining the other traits in new ways. In this way, new areas of the search space are explored while the operator attempts to retain optimal solution characteristics.

- **Mutation**: Some of the next-iteration solutions are passed to a mutation operator, which introduces random variations in the solutions. The purpose of the mutation operator is to ensure that the solution space is adequately searched in order to prevent premature convergence to a local optimum.

- **Repeat**: The current iteration of solutions is replaced by the new iteration. If the stopping criterion is not satisfied, the process returns to the selection phase.

The crossover and mutation operators are commonly called genetic operators. Selection and crossover distinguish genetic algorithms from a purely random search and direct the algorithm toward finding an optimum.

There are many ways to implement the general strategy just outlined, and it is also possible to combine the genetic algorithm approach with other heuristic solution improvement techniques. In the traditional genetic algorithm, the solution space is composed of bit strings, which are mapped to an objective function, and the genetic operators are modeled after biological processes. Although there is a theoretical foundation for the convergence of genetic algorithms that are formulated in this way, in practice most problems do not
fit naturally into this paradigm. Modern research has shown that optimizations can be set up by using the natural solution domain (for example, a real vector or integer sequence) and by applying crossover and mutation operators that are analogous to the traditional genetic operators but are more appropriate to the natural formulation of the problem.

Although genetic algorithms have been demonstrated to work well for a variety of problems, there is no guarantee of convergence to a global optimum. Also, the convergence of genetic algorithms can be sensitive to the choice of genetic operators, mutation probability, and selection criteria, so some initial experimentation and fine-tuning of these parameters are often required.

The LSO solver seeks to automate this process by using hybrid optimization in which several genetic algorithms can run in parallel and each algorithm is initialized with different default option settings. The LSO solver also adds a step to the GA; this step permits generating set search (GSS) algorithms to be used on a selected subset of the current population. GSS algorithms are designed for problems that have continuous variables and have the advantage that, in practice, they often require significantly fewer evaluations than GAs in order to converge. Furthermore, GSS can provide a measure of local optimality that is very useful in performing multimodal optimization.

The following additional “growth steps” are used whenever continuous variables are present:

- **local search selection:** A small subset of points are selected based on their fitness score and distance to (1) other points and (2) pre-existing locally optimal points.

- **local search optimization:** Local search optimization begins concurrently for each selected point. The only change that is made to the original optimization problem is that the variables’ lower and upper bounds are modified to temporarily fix integer variables at their current setting.

The LSO solver performs these additional growth steps at each iteration; they permit selected members of the population (based on diversity and fitness) to benefit from local optimization over the continuous variables. If only integer variables are present, these additional steps are not used.

---

**Multiobjective Optimization**

Many practical optimization problems involve more than one objective criterion, so the decision maker needs to examine trade-offs between conflicting objectives. For example, for a particular financial model you might want to maximize profit while minimizing risk, or for a structural model you might want to minimize weight while maximizing strength. The desired result for such problems is usually not a single solution but rather a range of solutions that you can use to identify an acceptable compromise. Ideally each point represents a necessary compromise in the sense that no single objective can be improved without causing at least one remaining objective to deteriorate. The goal of the LSO solver in the multiobjective case is thus to provide to the decision maker a set of points that represent the continuum of best-case scenarios. Multiobjective optimization is performed in the LSO solver whenever more than one objective function exists. For an example, see “Example 13.7: Multiobjective Optimization” on page 599.
Mathematically, you can define multiobjective optimization in terms of dominance and Pareto optimality. For a $k$-objective minimizing optimization problem, a point $x$ is dominated by a point $y$ if $f_i(x) \geq f_i(y)$ for all $i = 1, \ldots, k$ and $f_j(x) > f_j(y)$ for some $j = 1, \ldots, k$.

A Pareto-optimal set contains only nondominated solutions. In Figure 13.2, a Pareto-optimal frontier is plotted with respect to minimization objectives $f_1(x)$ and $f_2(x)$ along with a corresponding population of 10 points that are plotted in the objective space. In this example, point $a$ dominates $\{e, f, k\}$, $b$ dominates $\{e, f, g, k\}$, $c$ dominates $\{g, h, k\}$, and $d$ dominates $\{k, i\}$. Although no other point in the population dominates point $c$, it has not yet converged to the true Pareto-optimal frontier. Thus there are points in a neighborhood of $c$ that have smaller values of $f_1$ and $f_2$.

![Figure 13.2 Pareto-Optimal Set](image)

In the constrained case, a point $x$ is dominated by a point $y$ if $\theta(x) > \epsilon$ and $\theta(y) < \theta(x)$, where $\theta(x)$ denotes the maximum constraint violation at point $x$ and the feasibility tolerance is $\epsilon$; thus feasibility takes precedence over objective function values.

Genetic algorithms enable you to attack multiobjective problems directly in order to evolve a set of Pareto-optimal solutions in one run of the optimization process instead of solving multiple separate problems. In addition, the LSO solver conducts local searches in neighborhoods around nondominated points to improve objective function values and reduce crowding. Because the number of nondominated points that are encountered might be greater than the total population size, the LSO solver stores nondominated points in an archived set $\mathcal{N}$; you can specify the PARETOMAX= option in the SOLVE WITH LSO statement to control the size of this set.

Although it is difficult to verify directly that a point lies on the true Pareto-optimal frontier without using derivatives, you can measure convergence indirectly by monitoring movement of the population with respect to $\mathcal{N}$, the current set of nondominated points. A number of metrics for measuring convergence in multiobjective evolutionary algorithms have been suggested, such as the generational distance by Van Veldhuizen (1999), the inverted generational distance by Coello Coello and Cruz Cortes (2005), and the averaged Hausdorff distance by Schütze et al. (2012). The LSO solver uses a variation of the averaged Hausdorff distance that is extended for general constraints.

Distance between sets is computed in terms of the distance between a point and a set, which in turn is defined in terms of the distance between two points. The distance measure that the LSO solver uses for two points $x$ and $y$ is
and $y$ is calculated as

$$d(x, y) = \|\theta(x) - \theta(y)\| + \sum_{i=1}^{k} |f_i(x) - f_i(y)|$$

where $\theta(x)$ denotes the maximum constraint violation at point $x$. Then the distance between a point $x$ and a set $\mathcal{A}$ is defined as

$$d(x, \mathcal{A}) = \min_{y \in \mathcal{A}} d(x, y)$$

Let $\mathcal{F}_j$ denote the set of all nondominated points within the current population at the start of generation $j$. Let $\mathcal{N}_j$ denote the set of all nondominated points at the start of generation $j$. At the beginning of each generation, $\mathcal{F}_j \subseteq \mathcal{N}_j$ is always satisfied. Then progress that the solver makes during iteration $j+1$ is defined as

$$\text{Progress}(j + 1) = \frac{1}{|\mathcal{F}_j|} \sum_{x \in \mathcal{F}_j} d(x, \mathcal{N}_{j+1})$$

Because $d(x, \mathcal{N}_{j+1}) = 0$ whenever $x \in \mathcal{F}_j \cap \mathcal{F}_{j+1}$, the preceding sum is over the set of points in the population that move from a status of nondominated to dominated. In this case, the progress that the solver makes is measured as the distance to the nearest dominating point.

**Function Value Caching**

To improve performance, the LSO solver implements a caching system. This system helps reduce the overall workload of the solver by eliminating duplicate function evaluations. As the individual optimizers submit new trial points for evaluation, the points are saved in the cache along with their function evaluation values. Then, before beginning the potentially expensive function evaluations for a new trial point, the LSO solver determines whether a similar point already exists within the cache. If a similar point is found in the cache, its function values are immediately returned for the newly submitted point. By returning function values from the cache instead of duplicating the effort of computing new function values, the solver can achieve significant performance improvements in the overall procedure.

The cache is implemented as a splay tree, similar to that used in Gray and Kolda (2006) and Hough, Kolda, and Patrick (2000). The splay tree rebalances itself as new points are added to the cache. This rebalancing ensures that recently added or accessed points are maintained at the root of the tree and are therefore available for quick retrieval. This splay tree data structure lends itself nicely to the needs of a caching system.

When determining whether a newly submitted trial point has a similar point already in the cache, the LSO solver compares the new trial point to all previous trial points by using the value of the CACHETOL= option in the SOLVE WITH LSO statement. This value specifies a tolerance to use in comparing two points and determining whether the two points are similar enough.

Two points $x$ and $y$ are considered *cache equivalent* if

$$|x_i - y_i| \leq s_i \epsilon, \text{ for } i = 1, \ldots, n$$

where $s$ denotes the corresponding scaling vector. Thus, if a point $x$ has been evaluated (or is being evaluated) and a point $y$ is submitted for evaluation and satisfies the preceding criteria, then $y$ is “cache evaluated” instead of being evaluated directly. In this case, $y$ is associated with $f(x)$ and $c(x)$. Although the splay tree
is very efficient, the cache lookup time for quick evaluation might eventually dominate the evaluation process time. You can use the CACHEMAX= option in the SOLVE WITH LSO statement to limit the size of the cache.

Even when nonlinear objective functions are not defined, the cache system helps avoid redundant computations when the linear constraints are explicitly handled.

---

**Iteration Log**

The iteration log provides detailed information about the progress of the LSO solver. The log appears by default or if you specify LOGLEVEL=1 in the SOLVE WITH LSO statement. The iteration log provides this information in the following columns:

- **Iteration**: Displays the number of completed genetic algorithm iterations.
- **Best Objective**: Displays the best objective that is found at each iteration with respect to the current value of the FEASTOL= option. **NOTE**: This column is included only for single-objective optimization.
- **Nondom**: Displays the number of nondominated solutions in the current Pareto-optimal set. **NOTE**: This column is included only for multiobjective optimization.
- **Progress**: Displays a numerical indication of the progress that the solver is making from one iteration to the next. For a description of how this value is computed, see the section “Multiobjective Optimization” on page 578. **NOTE**: This column is included only for multiobjective optimization.
- **Infeasibility**: Displays the aggregate constraint violation of the current best point.
- **Evals**: Displays the cumulative number of function evaluations.
- **Time**: Displays the time needed to achieve the current best solution.

---

**Solver Termination Messages**

After the LSO solver terminates, one of the following messages is displayed:

**Interrupted by user.**

The solver’s execution was interrupted by the user.

**Function convergence criteria reached.**

The best objective value and feasibility have not sufficiently improved in the specified number of iterations. This condition is a typical exit state if the solver reaches the global optimum early in the process.

**Maximum function evaluations reached.**

The LSO solver has attempted to exceed the maximum number of function evaluations.
Generations complete.
The maximum number of GA generations (iterations) has been reached. In this situation, no new points are generated, and the solver exits.

Maximum time reached.
The LSO solver has exceeded the maximum allotted time.

Problem might be unbounded.
The objective value appears to take on arbitrarily large values at points that satisfy the feasibility tolerance.

Infeasible.
The problem is infeasible.

Failed.
The solver exited for a reason other than the preceding reasons.

Macro Variable _OROPTMODEL_
The OPTMODEL procedure defines a macro variable named _OROPTMODEL_. This variable contains a character string that indicates the status of the procedure upon termination. The contents of the macro variable depend on which solver was invoked. For the LSO solver, the various terms of _OROPTMODEL_ are interpreted as follows.

STATUS
indicates the solver status at termination. It can take one of the following values:

OK The solver terminated normally.
SYNTAX_ERROR Syntax was used incorrectly.
DATA_ERROR The input data are inconsistent.
OUT_OF_MEMORY Insufficient memory was allocated to the solver.
IO_ERROR A problem in reading or writing of data has occurred.
SEMANTIC_ERROR An evaluation error, such as an invalid operand type, has occurred.
ERROR The status cannot be classified into any of the preceding categories.

SOLUTION_STATUS
indicates the status of the solution at termination. It can take one of the following values:

ABORTED The user signaled that the solver should terminate.
ABSFCONV The solver terminated on the absolute function convergence criterion.
INFEASIBLE Problem is globally infeasible. This value is returned only if all constraints are linear.
MAXFUNC The solver terminated on the maximum number of function evaluations.
MAXGEN The maximum number of genetic algorithm iterations was completed.
MAXTIME The maximum time limit was reached.
UNBOUNDED The problem might be unbounded.
FAILED The status cannot be classified into any of the preceding categories.

OBJECTIVE indicates the objective value that the solver obtains at termination. **NOTE:** This value is missing for multiobjective optimization.

NONDOMINATED indicates the number of nondominated solutions in the Pareto-optimal set. **NOTE:** This value is included only for multiobjective optimization.

PROGRESS indicates the progress that the solver made during the last iteration. For a description of how this value is computed, see the section “Multiobjective Optimization” on page 578. **NOTE:** This value is included only for multiobjective optimization.

INFEASIBILITY indicates the level of infeasibility of the constraints at the solution.

EVALUATIONS indicates the total number of evaluations that were not cached.

ITERATIONS indicates the number of iterations that were required in order to solve the problem.

PRESOLVE_TIME indicates the real time (in seconds) that is used to set up the optimization problem and distribute data.

SOLUTION_TIME indicates the real time (in seconds) that is used by the solver to perform iterations for solving the problem.

---

**Examples: LSO Solver**

**Example 13.1: Linear Constraints and a Quadratic Objective**

This example uses data from Floudas and Pardalos (1992) and illustrates how to use the SOLVE WITH LSO statement in the OPTMODEL procedure. The problem has nine linear constraints and a quadratic objective function. Suppose you want to minimize

\[
f(x) = 5 \sum_{i=1}^{4} x_i - 5 \sum_{i=1}^{4} x_i^2 - \sum_{i=5}^{13} x_i
\]
subject to
\[
\begin{align*}
2x_1 + 2x_2 + x_{10} + x_{11} & \leq 10 \\
2x_1 + 2x_3 + x_{10} + x_{12} & \leq 10 \\
2x_1 + 2x_3 + x_{11} + x_{12} & \leq 10 \\
-8x_1 + x_{10} & \leq 0 \\
-8x_2 + x_{11} & \leq 0 \\
-8x_3 + x_{12} & \leq 0 \\
-2x_4 - x_{5} + x_{10} & \leq 0 \\
-2x_6 - x_{7} + x_{11} & \leq 0 \\
-2x_8 - x_{9} + x_{12} & \leq 0
\end{align*}
\]

and
\[
\begin{align*}
0 \leq x_i & \leq 1, \quad i = 1, 2, \ldots, 9 \\
0 \leq x_i & \leq 100, \quad i = 10, 11, 12 \\
0 \leq x_{13} & \leq 1
\end{align*}
\]
The following statements use the LSO solver within PROC OPTMODEL:

```plaintext
proc optmodel;
  var x {i in 1..13} >= 0 <= (if i in 10..12 then 100 else 1);
  min f = 5 * sum {i in 1..4} x[i] - 5 * sum {i in 1..4} x[i]^2 - sum {i in 5..13} x[i];
  con a4: -8*x[1] + x[10] <= 0;
  con a6: -8*x[3] + x[12] <= 0;
  solve with lso / nthreads=2;
  print x;
quit;
```

Output 13.1.1 shows the output from running these statements.
### Output 13.1.1 Linear Constraints and a Quadratic Objective

#### The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
<td>Minimization</td>
</tr>
<tr>
<td>Objective Function</td>
<td>f</td>
</tr>
<tr>
<td>Objective Type</td>
<td>Quadratic</td>
</tr>
</tbody>
</table>

| Number of Variables | 13       |
| Bounded Above       | 0        |
| Bounded Below       | 0        |
| Bounded Below and Above | 13   |
| Free                | 0        |
| Fixed               | 0        |

| Number of Constraints | 9         |
| Linear LE (<=)        | 9         |
| Linear EQ (=)         | 0         |
| Linear GE (>=)        | 0         |
| Linear Range          | 0         |

**Constraint Coefficients** 27

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>LSO</td>
</tr>
<tr>
<td>Objective Function</td>
<td>f</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Function Convergence</td>
</tr>
<tr>
<td>Objective Value</td>
<td>-15.00106969</td>
</tr>
</tbody>
</table>

| Infeasibility         | 0.0009765677 |
| Random Seed Used      | 1         |

| Evaluations           | 3372     |
| Cached Evaluations    | 532      |
| Iterations            | 29       |
| Presolve Time         | 0.00     |
| Solution Time         | 0.34     |
Example 13.2: Linear Constraints and a Nonlinear Objective

The problem in this example is to minimize the six-hump camelback function (Michalewicz 1996, Appendix B). Minimize

\[ f(x) = \left( 4 - 2.1x_1^2 + \frac{x_1^4}{3} \right)x_1^2 + x_1x_2 + (-4 + 4x_2^2)x_2^2 \]

subject to

\[
\begin{align*}
2x_1 + x_2 & \leq 2 \\
x_1 - x_2 & \geq -2 \\
x_1 + 2x_2 & \geq -2
\end{align*}
\]

Providing derivative-free algorithms with good estimates for lower and upper bounds often greatly improves performance because it prevents the algorithm from unnecessarily sampling in regions that you do not want to explore. For this problem, the following statements add the explicit variable bounds \(-2 \leq x_1 \leq 2\) and \(-2 \leq x_2 \leq 2\):

```plaintext
proc optmodel;
var x {1..2} >= -2 <= 2;
con a1: 2*x[1] + x[2] <= 2;
       + (-4 + 4*x[2]^2)*x[2]^2;
solve with lso / nthreads=2;
print x;
quit;
```

Output 13.2.1 shows the output from running these steps.
Output 13.2.1 Linear Constraints and a Nonlinear Objective

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objective Sense</strong></td>
</tr>
<tr>
<td><strong>Objective Function</strong></td>
</tr>
<tr>
<td><strong>Objective Type</strong></td>
</tr>
<tr>
<td><strong>Number of Variables</strong></td>
</tr>
<tr>
<td><strong>Bounded Above</strong></td>
</tr>
<tr>
<td><strong>Bounded Below</strong></td>
</tr>
<tr>
<td><strong>Bounded Below and Above</strong></td>
</tr>
<tr>
<td><strong>Free</strong></td>
</tr>
<tr>
<td><strong>Fixed</strong></td>
</tr>
<tr>
<td><strong>Number of Constraints</strong></td>
</tr>
<tr>
<td><strong>Linear LE (&lt;=)</strong></td>
</tr>
<tr>
<td><strong>Linear EQ (=)</strong></td>
</tr>
<tr>
<td><strong>Linear GE (&gt;=)</strong></td>
</tr>
<tr>
<td><strong>Linear Range</strong></td>
</tr>
<tr>
<td><strong>Constraint Coefficients</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solver</strong></td>
</tr>
<tr>
<td><strong>Objective Function</strong></td>
</tr>
<tr>
<td><strong>Solution Status</strong></td>
</tr>
<tr>
<td><strong>Objective Value</strong></td>
</tr>
<tr>
<td><strong>Infeasibility</strong></td>
</tr>
<tr>
<td><strong>Random Seed Used</strong></td>
</tr>
<tr>
<td><strong>Evaluations</strong></td>
</tr>
<tr>
<td><strong>Cached Evaluations</strong></td>
</tr>
<tr>
<td><strong>Iterations</strong></td>
</tr>
<tr>
<td><strong>Presolve Time</strong></td>
</tr>
<tr>
<td><strong>Solution Time</strong></td>
</tr>
</tbody>
</table>

\[
\begin{array}{ll}
[1] & x \\
1 & 0.089842 \\
2 & -0.712658 \\
\end{array}
\]
Example 13.3: Using Nonlinear Constraints

The following optimization problem is discussed in Haverly (1978) and Liebman et al. (1986). This example illustrates a problem that has both linear and nonlinear constraints and a linear objective. Maximize

$$f(x) = 9x_1 + 15x_2 - 6x_3 - 16x_4 - 10x_5$$

subject to

$$x_3 + x_4 = x_6 + x_7$$
$$x_6 + x_8 = x_1$$
$$x_7 + x_9 = x_2$$
$$x_8 + x_9 = x_5$$
$$2.5x_1 - x_{10}x_6 - 2x_8 \geq 0$$
$$1.5x_2 - x_{10}x_7 - 2x_9 \geq 0$$
$$3x_3 + x_4 - x_{10}(x_3 + x_4) = 0$$

and

$$0 \leq x_1 \leq 100$$
$$0 \leq x_2 \leq 200$$
$$1 \leq x_{10} \leq 3$$
$$0 \leq x_i, \text{ for } i = 3, \ldots, 9$$

In the following steps, the problem definition is described in PROC OPTMODEL. This example also demonstrates the distributed computing capabilities of the LSO solver. By simply passing a reference to a running CAS session via the SESSREF= option, you can direct the LSO solver to run in the distributed CAS environment.

```plaintext
proc optmodel sessref=mysess;
    var x{1..10} >= 0;
    x[1].ub = 100;
    x[2].ub = 200;
    x[10].lb = 1;
    x[10].ub = 3;
        x[6] + x[8] = x[1],
        x[8] + x[9] = x[5],
        2.5*x[1] - x[10]*x[6] - 2*x[8] >= 0,
    solve with lso / logfreq=10 nthreads=8 popsize=400;
    print x;
quit;
```

Output 13.3.1 shows the ODS tables that are produced by running these steps.
**Output 13.3.1** Using Nonlinear Constraints: ODS Tables

**The OPTMODEL Procedure**

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objective Sense</strong></td>
</tr>
<tr>
<td><strong>Objective Function</strong></td>
</tr>
<tr>
<td><strong>Objective Type</strong></td>
</tr>
<tr>
<td><strong>Number of Variables</strong></td>
</tr>
<tr>
<td><strong>Bounded Above</strong></td>
</tr>
<tr>
<td><strong>Bounded Below</strong></td>
</tr>
<tr>
<td><strong>Bounded Below and Above</strong></td>
</tr>
<tr>
<td><strong>Free</strong></td>
</tr>
<tr>
<td><strong>Fixed</strong></td>
</tr>
<tr>
<td><strong>Number of Constraints</strong></td>
</tr>
<tr>
<td><strong>Linear LE (&lt;=)</strong></td>
</tr>
<tr>
<td><strong>Linear EQ (=)</strong></td>
</tr>
<tr>
<td><strong>Linear GE (&gt;=)</strong></td>
</tr>
<tr>
<td><strong>Linear Range</strong></td>
</tr>
<tr>
<td><strong>Nonlinear LE (&lt;=)</strong></td>
</tr>
<tr>
<td><strong>Nonlinear EQ (=)</strong></td>
</tr>
<tr>
<td><strong>Nonlinear GE (&gt;=)</strong></td>
</tr>
<tr>
<td><strong>Nonlinear Range</strong></td>
</tr>
<tr>
<td><strong>Constraint Coefficients</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solver</strong></td>
</tr>
<tr>
<td><strong>Objective Function</strong></td>
</tr>
<tr>
<td><strong>Solution Status</strong></td>
</tr>
<tr>
<td><strong>Objective Value</strong></td>
</tr>
<tr>
<td><strong>Infeasibility</strong></td>
</tr>
<tr>
<td><strong>Random Seed Used</strong></td>
</tr>
<tr>
<td><strong>Evaluations</strong></td>
</tr>
<tr>
<td><strong>Cached Evaluations</strong></td>
</tr>
<tr>
<td><strong>Iterations</strong></td>
</tr>
<tr>
<td><strong>Presolve Time</strong></td>
</tr>
<tr>
<td><strong>Solution Time</strong></td>
</tr>
</tbody>
</table>
Output 13.3.1 continued

<table>
<thead>
<tr>
<th></th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00115763</td>
</tr>
<tr>
<td>2</td>
<td>200.00000000</td>
</tr>
<tr>
<td>3</td>
<td>0.00012520</td>
</tr>
<tr>
<td>4</td>
<td>99.99659990</td>
</tr>
<tr>
<td>5</td>
<td>100.00086613</td>
</tr>
<tr>
<td>6</td>
<td>0.00000338</td>
</tr>
<tr>
<td>7</td>
<td>99.99771166</td>
</tr>
<tr>
<td>8</td>
<td>0.00017295</td>
</tr>
<tr>
<td>9</td>
<td>100.00139776</td>
</tr>
<tr>
<td>10</td>
<td>1.00000000</td>
</tr>
</tbody>
</table>

Output 13.3.2 shows the iteration log from running these steps.
### Output 13.3.2 Using Nonlinear Constraints: Log

NOTE: Problem generation will use 4 threads.

NOTE: The problem has 10 variables (0 free, 0 fixed).

NOTE: The problem has 4 linear constraints (0 LE, 4 EQ, 0 GE, 0 range).

NOTE: The problem has 13 linear constraint coefficients.

NOTE: The problem has 3 nonlinear constraints (0 LE, 1 EQ, 2 GE, 0 range).

NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0 nonlinear constraints.

NOTE: The SOLVE statement is executing in the distributed computing environment with 4 worker nodes.

NOTE: The LSO solver is using up to 32 total threads across 4 machines.

NOTE: The LSO solver is using the EAGLS optimizer algorithm.

NOTE: The problem has 10 variables (0 integer, 10 continuous).

NOTE: The problem has 7 constraints (4 linear, 3 nonlinear).

NOTE: The problem has 3 user-defined functions.

NOTE: The deterministic parallel mode is enabled.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Objective</th>
<th>Infeasibility</th>
<th>Evals</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>399.49904945</td>
<td>0.00000007104373</td>
<td>3171</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>399.99620302</td>
<td>0.0000000344593</td>
<td>6020</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td>400.01128609</td>
<td>0.00086688</td>
<td>8702</td>
<td>1</td>
</tr>
<tr>
<td>41</td>
<td>400.02979991</td>
<td>0.00089062</td>
<td>11330</td>
<td>1</td>
</tr>
<tr>
<td>51</td>
<td>400.04028270</td>
<td>0.00096977</td>
<td>14044</td>
<td>1</td>
</tr>
<tr>
<td>61</td>
<td>400.04359244</td>
<td>0.00099703</td>
<td>16667</td>
<td>1</td>
</tr>
<tr>
<td>71</td>
<td>400.05034379</td>
<td>0.00097599</td>
<td>19364</td>
<td>2</td>
</tr>
<tr>
<td>81</td>
<td>400.05149884</td>
<td>0.00092924</td>
<td>21930</td>
<td>2</td>
</tr>
<tr>
<td>91</td>
<td>400.05276261</td>
<td>0.00098551</td>
<td>24592</td>
<td>2</td>
</tr>
<tr>
<td>101</td>
<td>400.05453441</td>
<td>0.00099273</td>
<td>27243</td>
<td>2</td>
</tr>
<tr>
<td>111</td>
<td>400.05472984</td>
<td>0.00095487</td>
<td>29993</td>
<td>3</td>
</tr>
<tr>
<td>121</td>
<td>400.05540781</td>
<td>0.00098994</td>
<td>32592</td>
<td>3</td>
</tr>
<tr>
<td>122</td>
<td>400.05540781</td>
<td>0.00098994</td>
<td>32855</td>
<td>3</td>
</tr>
</tbody>
</table>

NOTE: Function convergence criteria reached.

STATUS=OK SOLUTION_STATUS=ABSFCNV OBJECTIVE=400.05540781
INFEASIBILITY=0.0009899432 EVALUATIONS=32855 ITERATIONS=122 PRESOLVE_TIME=0.00
SOLUTION_TIME=2.94
Example 13.4: Using External Data Sets

This example illustrates the use of external data that are required in order to evaluate the objective function. The Bard function (Moré, Garbow, and Hillstrom 1981) is a least squares problem that has \( n = 3 \) parameters and \( m = 15 \) functions \( f_k \).

\[
f(x) = \frac{1}{2} \sum_{k=1}^{15} f_k^2(x), \quad x = (x_1, x_2, x_3)
\]

where

\[
f_k(x) = y_k - \left( x_1 + \frac{k}{v_k x_2 + w_k x_3} \right)
\]

with \( v_k = 16 - k \), \( w_k = \min(u_k, v_k) \), and

\[
y = (0.14, 0.18, 0.22, 0.25, 0.29, 0.32, 0.35, 0.39, 0.37, 0.58, 0.73, 0.96, 1.34, 2.10, 4.39)
\]

The minimum function value \( f(x^*) = 4.107 \times 10^{-3} \) occurs at the point \((0.08, 1.13, 2.34)\). In this example, the additional variable bounds \(-1,000 \leq x_i \leq 1,000\) for \( i = 1, 2, 3 \) are added.

The following example code shows two approaches to specifying the objective function. The first approach assumes that the necessary data are stored within the FCMP function that defines the objective. In this case, you can specify the objective function without using an external data set, as follows:

```sas
proc fcmp outlib=sasuser.myfuncs.mypkg;
  function bard(x1, x2, x3);
    array y[15] /nosym (0.14 0.18 0.22 0.25 0.29
                          0.32 0.35 0.39 0.37 0.58
                          0.73 0.96 1.34 2.10 4.39);
    fx = 0;
    do k=1 to 15;
      vk = 16 - k;
      wk = min(k,vk);
      fxk = y[k] - (x1 + k/(vk*x2 + wk*x3));
      fx = fx + fxk**2;
    end;
    return (0.5*fx);
  endsub;
run;
```

```sas
options cmplib = sasuser.myfuncs;
proc optmodel;
  var x {1..3} >= -1000 <= 1000;
  min f = bard(of x[*]);
  solve with lso / nthreads=2;
  print x;
quit;
```

Output 13.4.1 shows the output from running these steps.
Output 13.4.1 Using External Data Sets

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense: Minimization</td>
</tr>
<tr>
<td>Objective Function: f</td>
</tr>
<tr>
<td>Objective Type: Nonlinear</td>
</tr>
</tbody>
</table>

| Number of Variables: 3          |
| Bounded Above: 0               |
| Bounded Below: 0               |
| Bounded Below and Above: 3      |
| Free: 0                        |
| Fixed: 0                       |

| Constraint Coefficients: 0      |

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver: LSO</td>
</tr>
<tr>
<td>Objective Function: f</td>
</tr>
<tr>
<td>Solution Status: Function Convergence</td>
</tr>
<tr>
<td>Objective Value: 0.0041074408</td>
</tr>
</tbody>
</table>

| Infeasibility: 0                |
| Random Seed Used: 1             |

Evaluations: 4589  
Cached Evaluations: 55  
Iterations: 53  
Presolve Time: 0.00  
Solution Time: 0.29

\[ \begin{array}{c}
[1] \quad x \\
1 \quad 0.082419 \\
2 \quad 1.133250 \\
3 \quad 2.343536
\end{array} \]

The preceding approach can be prohibitive if the data set is large. As a second approach to specifying the objective function, you can define data in a SAS data set, read the data set into PROC OPTMODEL, and then use the values from that data set as part of the objective computation. This approach is shown in the following statements:

```sas
data barddata;
  k = _n_;  
  input y @@;  
  datalines;  
0.14 0.18 0.22 0.25 0.29  
0.32 0.35 0.39 0.37 0.58
```

Chapter 13: The Local Search Optimization Solver

0.73 0.96 1.34 2.10 4.39

;  
proc fcmp outlib=sasuser.myfuncs.mypkg;
function bard(x1, x2, x3, k, y);
  vk = 16 - k;
  wk = min(k, vk);
  fxk = y - (x1 + k/(vk*x2 + wk*x3));
  return (0.5*fxk**2);
endsub;
run;

options cmplib = sasuser.myfuncs;
proc optmodel;
  set KSET;
  num y {KSET};
  read data barddata into KSET=[k] y;
  var x {1..3} >= -1000 <= 1000;
  min f = sum {k in KSET} bard(of x[*], k, y[k]);
  solve with lso / nthreads=2;
  print x;
quit;

Output 13.4.2 shows the output from running these statements.

Output 13.4.2 Using External Data Sets (II)

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objective Sense</strong></td>
</tr>
<tr>
<td><strong>Objective Function</strong></td>
</tr>
<tr>
<td><strong>Objective Type</strong></td>
</tr>
<tr>
<td><strong>Number of Variables</strong></td>
</tr>
<tr>
<td><strong>Bounded Above</strong></td>
</tr>
<tr>
<td><strong>Bounded Below</strong></td>
</tr>
<tr>
<td><strong>Bounded Below and Above</strong></td>
</tr>
<tr>
<td><strong>Free</strong></td>
</tr>
<tr>
<td><strong>Fixed</strong></td>
</tr>
<tr>
<td><strong>Number of Constraints</strong></td>
</tr>
<tr>
<td><strong>Constraint Coefficients</strong></td>
</tr>
</tbody>
</table>
Example 13.5: Johnson’s Systems of Distributions

This example further illustrates the use of external data that are required in order to evaluate the objective function. For this example, a data set that contains \( n = 20,000 \) randomly generated observations is used to estimate the parameters for the Johnson \( SU \) family of distributions (Bowman and Shenton 1983). The objective is the log likelihood for the family, which involves four variables, \( x = (x_1, x_2, x_3, x_4) \),

\[
f(x) = n \log x_4 - n \log x_2 - \frac{1}{2} \sum_{k=1}^{n} \left( (x_3 + x_4 \log z_k)^2 + \log(1 + y_k^2) \right)
\]

where

\[
z_k = y_k + \sqrt{1 + y_k^2} \text{ with } y_k = \frac{d_k - x_1}{x_2}
\]

Here, \( d_k \) denotes the value of \( d \) in the \( k \)th observation of the data set that is generated by the following DATA step:

```plaintext
data sudata;
  n = 20000;
  theta = -1;
  sigma = 1;
  delta = 3;
  gamma = 5;
  rngSeed=12345;
call streaminit(rngSeed);
do i = 1 to n;
  z = rand('NORMAL');
  a = exp( (z - gamma)/delta );
  d = sigma * ( (a**2 - 1)/(2*a) ) + theta;
  output;
end;
keep d;
run;
```
This generates a data set called suData that contains \( n = 20,000 \) observations. You can modify \( n \) to increase or decrease the computational work per function evaluation, as follows:

```plaintext
proc optmodel;
    set OBS;
    num d {OBS};
    read data sudata into OBS=[_N_] d;
    num n = card(OBS);
    var x {1..4};
    x[2].lb = 1e-12;
    x[4].lb = 1e-12;
    impvar y {k in OBS} = (d[k] - x[1]) / x[2];
    impvar z {k in OBS} = y[k] + sqrt(1 + y[k]^2);
    max f = n * (log(x[4]) - log(x[2]))
            - 0.5 * sum {k in OBS} ((x[3] + x[4]*log(z[k]))^2 + log(1 + y[k]^2));
    solve with lso / maxgen=1000 nthreads=8;
    print x;
quit;
```

Output 13.5.1 shows the output from running these steps.

**Output 13.5.1** Estimation for Johnson \( SU \) Family of Distributions

**The OPTMODEL Procedure**

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
<tr>
<td>Number of Variables</td>
</tr>
<tr>
<td>Bounded Above</td>
</tr>
<tr>
<td>Bounded Below</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
</tr>
<tr>
<td>Free</td>
</tr>
<tr>
<td>Fixed</td>
</tr>
<tr>
<td>Number of Constraints</td>
</tr>
<tr>
<td>Constraint Coefficients</td>
</tr>
</tbody>
</table>
Example 13.6: Discontinuous Function with a Lookup Table

This example illustrates the ability of the LSO solver to optimize a discontinuous function. The example generates arrays of discrete values that approximate a given smooth nonlinear function. The function that is optimized is simply using those data as a lookup table to find the appropriate discretized value. The underlying smooth function is defined as follows:

\[ f(x) = x \sin(x) + x^2 \cos(x) \]

This first PROC OPTMODEL call shows how to optimize this function by using the discretized lookup approach:

```plaintext
%let N = 100;
%let L = 0;
%let U = 10;

proc optmodel;
  set OBS = 1..&N;
  num xdata {i in OBS} = &L + (i/&N)*(&U-&L);
  num ydata {i in OBS} = xdata[i]*sin(xdata[i]) + xdata[i]^2*cos(xdata[i]);
  var x >= &L <= &U;
  min f = sum {i in 1..&N-1} ydata[i] * (xdata[i] <= x < xdata[i+1]);
  solve with lso;
  print x;
quit;
```

The previous statements first create the lookup arrays \((xdata\) and \(ydata\)) for \(x\) values at 100 \((N)\) uniformly spaced points between 0 \((L)\) and 10 \((U)\). The objective function \(f\) is simply returning the discretized function value when the given \(x\) value is between two neighboring \(x\) values in the array. When the solver finds the appropriate location in the arrays, the smaller of the two nearest points is returned as the function value.

Output 13.6.1 shows the ODS tables that are produced.
Output 13.6.1 Discontinuous Function: ODS Tables

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
</tbody>
</table>

| Number of Variables     | 1                     |
| Bounded Above           | 0                     |
| Bounded Below           | 0                     |
| Bounded Below and Above | 1                     |
| Free                    | 0                     |
| Fixed                   | 0                     |

| Number of Constraints   | 0                     |
| Constraint Coefficients | 0                     |

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
</tbody>
</table>

Infeasibility 0
Random Seed Used 1

Evaluations 284
Cached Evaluations 51
Iterations 11
Presolve Time 0.00
Solution Time 0.04

\[
\begin{align*}
x &= 9.7436
\end{align*}
\]

The following statements optimize the smooth version of the same function to demonstrate that virtually the same result is achieved in both cases:

```plaintext
proc optmodel;
  var x >= &L <= &U;
  min f = x*sin(x) + x^2*cos(x);
  solve with lso;
  print x;
quit;
```

Output 13.6.2 shows the ODS tables that are produced.
Example 13.7: Multiobjective Optimization

The following optimization problem is discussed in Huband et al. (2006) and Custódio et al. (2011). This example illustrates how to optimize a problem that contains multiple nonlinear objectives. This problem minimizes

\[ f_1(x) = (x_1 - 1)^2 + (x_1 - x_2)^2 \quad \text{and} \quad f_2(x) = (x_1 - x_2)^2 + (x_2 - 3)^2 \]

subject to \( 0 \leq x_1, x_2 \leq 5 \). The variable information and the objective functions are defined using the standard PROC OPTMODEL syntax. This example shows how you can specify the objective names as part of the SOLVE WITH LSO statement. By default, all currently defined objectives are used, so specifying the objective names is optional unless you want to exclude some previously defined objectives from your problem.
The following PROC OPTMODEL code shows how to define the two objective functions in the multiobjective optimization problem. This code also demonstrates how to use the OBJ keyword in the SOLVE WITH LSO statement.

```plaintext
proc optmodel;
  var x {1..2} >= 0 <= 5;
  min f1 = (x[1]-1)^2 + (x[1]-x[2])^2;
  min f2 = (x[1]-x[2])^2 + (x[2]-3)^2;
  solve with lso obj (f1 f2) / logfreq=50;
  num f1sol {1.._NSOL_};
  num f2sol {1.._NSOL_};
  for {s in 1.._NSOL_} do;
    for {i in 1..2} x[i] = x[i].sol[s];
    f1sol[s] = f1;
    f2sol[s] = f2;
  end;
  create data pareto from [s]=(1.._NSOL_) {i in 1..2} <col('x'||i)=x[i].sol[s]>
    f1=f1sol[s] f2=f2sol[s];
quit;
```

%put &_OROPTMODEL_;
proc sgplot data=pareto;
  scatter x=f1 y=f2;
run;

Output 13.7.1 shows the ODS tables that are produced.

**Output 13.7.1** Multiobjective: ODS Tables

**The OPTMODEL Procedure**

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Objectives</td>
</tr>
<tr>
<td>Number of Variables</td>
</tr>
<tr>
<td>Bounded Above</td>
</tr>
<tr>
<td>Bounded Below</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
</tr>
<tr>
<td>Free</td>
</tr>
<tr>
<td>Fixed</td>
</tr>
<tr>
<td>Number of Constraints</td>
</tr>
<tr>
<td>Constraint Coefficients</td>
</tr>
</tbody>
</table>
### Output 13.7.1 continued

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>LSO</td>
</tr>
<tr>
<td>Objective Function</td>
<td>(MULTIPLE)</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Function Convergence</td>
</tr>
<tr>
<td>Nondominated Solutions</td>
<td>4932</td>
</tr>
<tr>
<td>Progress</td>
<td>7.0494933E-7</td>
</tr>
<tr>
<td>Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Random Seed Used</td>
<td>1</td>
</tr>
<tr>
<td>Evaluations</td>
<td>22734</td>
</tr>
<tr>
<td>Cached Evaluations</td>
<td>139</td>
</tr>
<tr>
<td>Iterations</td>
<td>428</td>
</tr>
<tr>
<td>Presolve Time</td>
<td>0.00</td>
</tr>
<tr>
<td>Solution Time</td>
<td>3.60</td>
</tr>
</tbody>
</table>

Output 13.7.2 shows the iteration log.
### Output 13.7.2  Multiobjective: Log

NOTE: Problem generation will use 4 threads.

NOTE: The problem has 2 variables (0 free, 0 fixed).

NOTE: The problem has 0 linear constraints (0 LE, 0 EQ, 0 GE, 0 range).

NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).

NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0 nonlinear constraints.

NOTE: The LSO solver is using up to 4 threads.

NOTE: The LSO solver is using the EAGLS optimizer algorithm.

NOTE: The problem has 2 variables (0 integer, 2 continuous).

NOTE: The problem has 0 constraints (0 linear, 0 nonlinear).

NOTE: The problem has 2 user-defined functions.

NOTE: The deterministic parallel mode is enabled.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Nondom</th>
<th>Progress</th>
<th>Infeasibility</th>
<th>Evals</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td>0.0000811</td>
<td>0</td>
<td>84</td>
<td>0</td>
</tr>
<tr>
<td>51</td>
<td>877</td>
<td>0.000123</td>
<td>0</td>
<td>2876</td>
<td>0</td>
</tr>
<tr>
<td>101</td>
<td>1654</td>
<td>0.000182</td>
<td>0</td>
<td>5576</td>
<td>0</td>
</tr>
<tr>
<td>151</td>
<td>2290</td>
<td>0.00032330545</td>
<td>0</td>
<td>8181</td>
<td>1</td>
</tr>
<tr>
<td>201</td>
<td>2874</td>
<td>0.000182</td>
<td>0</td>
<td>10796</td>
<td>1</td>
</tr>
<tr>
<td>251</td>
<td>3422</td>
<td>0.0003148003</td>
<td>0</td>
<td>13447</td>
<td>2</td>
</tr>
<tr>
<td>301</td>
<td>3847</td>
<td>0.00001559509</td>
<td>0</td>
<td>16046</td>
<td>2</td>
</tr>
<tr>
<td>351</td>
<td>4282</td>
<td>0.00001159917</td>
<td>0</td>
<td>18733</td>
<td>3</td>
</tr>
<tr>
<td>401</td>
<td>4712</td>
<td>0.00002423149</td>
<td>0</td>
<td>21315</td>
<td>3</td>
</tr>
<tr>
<td>428</td>
<td>4932</td>
<td>0.000000704949</td>
<td>0</td>
<td>22734</td>
<td>4</td>
</tr>
</tbody>
</table>

NOTE: Function convergence criteria reached.

NOTE: The data set WORK.PARETO has 4932 observations and 5 variables.

STATUS=OK  SOLUTION_STATUS=ABSFCNV OBJECTIVE=.  NONDOMINATED=4932
PROGRESS=7.0494933E-7  INFEASIBILITY=0  EVALUATIONS=22734  ITERATIONS=428
PRESOLVE_TIME=0.00  SOLUTION_TIME=3.60

When you are solving a multiobjective problem that contains two objectives, it can be helpful to use PROC SGPLOT to create a plot of the Pareto-optimal set that the LSO solver finds.

Output 13.7.3 shows a plot of the Pareto-optimal set that the LSO solver finds.
Output 13.7.3 Multiobjective: Plot of Pareto-Optimal Set

References


Chapter 14
The Mixed Integer Linear Programming Solver

Overview: MILP Solver

The OPTMODEL procedure provides a framework for specifying and solving mixed integer linear programs (MILPs). A standard mixed integer linear program has the formulation

\[ \min \quad c^T x \]

subject to

\[ Ax \{ \geq, =, \leq \} b \]

\[ 1 \leq x \leq u \]

\[ x_i \in \mathbb{Z} \quad \forall i \in S \]

(MathLP)
Chapter 14: The Mixed Integer Linear Programming Solver

where

- \( x \in \mathbb{R}^n \) is the vector of structural variables
- \( A \in \mathbb{R}^{m \times n} \) is the matrix of technological coefficients
- \( c \in \mathbb{R}^n \) is the vector of objective function coefficients
- \( b \in \mathbb{R}^m \) is the vector of constraints’ right-hand sides (RHS)
- \( l \in \mathbb{R}^n \) is the vector of lower bounds on variables
- \( u \in \mathbb{R}^n \) is the vector of upper bounds on variables
- \( S \) is a nonempty subset of the set \{1, \ldots, n\} of indices

The MILP solver, available in the OPTMODEL procedure, implements a linear-programming-based branch-and-cut algorithm. This divide-and-conquer approach attempts to solve the original problem by solving linear programming relaxations of a sequence of smaller subproblems. The MILP solver also implements advanced techniques such as presolving, generating cutting planes, and applying primal heuristics to improve the efficiency of the overall algorithm.

The MILP solver provides various control options and solution strategies. In particular, you can enable, disable, or set levels for the advanced techniques previously mentioned. It is also possible to input an incumbent solution; see the section “Warm Start Option” on page 612 for details.

---

**Getting Started: MILP Solver**

The following example illustrates how you can use the OPTMODEL procedure to solve mixed integer linear programs. For more examples, see the section “Examples: MILP Solver” on page 632. Suppose you want to solve the following problem:

\[
\begin{align*}
\text{min} & \quad 2x_1 - 3x_2 - 4x_3 \\
\text{s.t.} & \quad -2x_2 - 3x_3 \geq -5 \quad (R1) \\
& \quad x_1 + x_2 + 2x_3 \leq 4 \quad (R2) \\
& \quad x_1 + 2x_2 + 3x_3 \leq 7 \quad (R3) \\
& \quad x_1, x_2, x_3 \geq 0 \\
& \quad x_1, x_2, x_3 \in \mathbb{Z}
\end{align*}
\]

You can use the following statements to call the OPTMODEL procedure for solving mixed integer linear programs:

```plaintext
proc optmodel;
var x{1..3} >= 0 integer;
min f = 2*x[1] - 3*x[2] - 4*x[3];
solve with milp / presolver = automatic heuristics = automatic;
print x;
quit;
```
The `PRESOLVER=` and `HEURISTICS=` options specify the levels for presolving and applying heuristics, respectively. In this example, each option is set to its default value, `AUTOMATIC`, meaning that the solver automatically determines the appropriate levels for presolve and heuristics.

The optimal value of $x$ is shown in Figure 14.1.

**Figure 14.1** Solution Output

<table>
<thead>
<tr>
<th>The OPTMODEL Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1] $x$</td>
</tr>
<tr>
<td>1 0</td>
</tr>
<tr>
<td>2 1</td>
</tr>
<tr>
<td>3 1</td>
</tr>
</tbody>
</table>

The solution summary stored in the macro variable `_OROPTMODEL_` can be viewed by issuing the following statement:

```plaintext
%put &_OROPTMODEL_;
```

This statement produces the output shown in Figure 14.2.

**Figure 14.2** Macro Output

```
STATUS=OK   ALGORITHM=BAC   SOLUTION_STATUS=OPTIMAL   OBJECTIVE=-7   RELATIVE_GAP=0
ABSOLUTE_GAP=0  PRIMAL_INFEASIBILITY=0  BOUND_INFEASIBILITY=0
INTEGER_INFEASIBILITY=0  BEST_BOUND=-7  NODES=1  SOLUTIONS_FOUND=3  ITERATIONS=3
PRESOLVE_TIME=0.06  SOLUTION_TIME=0.13
```

## Syntax: MILP Solver

The following statement is available in the OPTMODEL procedure:

```plaintext
SOLVE WITH MILP < / options > ;
```

## Functional Summary

Table 14.1 summarizes the options available for the `SOLVE WITH MILP` statement, classified by function.

**Table 14.1** Options for the MILP Solver

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Presolve Option</td>
<td><code>PRESOLVER=</code></td>
</tr>
<tr>
<td>Warm Start Option</td>
<td><code>PRIMALIN</code></td>
</tr>
<tr>
<td>Control Options</td>
<td><code>ABSOBJGAP=</code></td>
</tr>
</tbody>
</table>
### Table 14.1 (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies the cutoff value for node removal</td>
<td>CUTOFF=</td>
</tr>
<tr>
<td>Emphasizes feasibility or optimality</td>
<td>EMPHASIS=</td>
</tr>
<tr>
<td>Specifies the maximum violation on variables and constraints</td>
<td>FEASTOL=</td>
</tr>
<tr>
<td>Specifies the maximum allowed difference between an integer variable’s</td>
<td>INTTOL=</td>
</tr>
<tr>
<td>value and an integer</td>
<td></td>
</tr>
<tr>
<td>Specifies the frequency of printing the node log</td>
<td>LOGFREQ=</td>
</tr>
<tr>
<td>Specifies the detail of solution progress printed in log</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the maximum number of nodes to be processed</td>
<td>MAXNODES=</td>
</tr>
<tr>
<td>Specifies the maximum number of solutions to return from the pool</td>
<td>MAXPOOLSOLS=</td>
</tr>
<tr>
<td>Specifies the maximum number of solutions to be found</td>
<td>MAXSOLS=</td>
</tr>
<tr>
<td>Specifies the time limit for the optimization process</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Specifies the tolerance used in determining the optimality of nodes in the</td>
<td>OPTTOL=</td>
</tr>
<tr>
<td>branch-and-bound tree</td>
<td></td>
</tr>
<tr>
<td>Specifies the probing level</td>
<td>PROBE=</td>
</tr>
<tr>
<td>Specifies the stopping criterion based on relative objective gap</td>
<td>RELOBJGAP=</td>
</tr>
<tr>
<td>Enables the use of scaling on the problem matrix</td>
<td>SCALE=</td>
</tr>
<tr>
<td>Specifies the initial seed for the random number generator</td>
<td>SEED=</td>
</tr>
<tr>
<td>Specifies the stopping criterion based on target objective value</td>
<td>TARGET=</td>
</tr>
<tr>
<td>Specifies whether time units are CPU time or real time</td>
<td>TIMETYPE=</td>
</tr>
<tr>
<td><strong>Heuristics Option</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the primal heuristics level</td>
<td>HEURISTICS=</td>
</tr>
<tr>
<td><strong>Search Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the level of conflict search</td>
<td>CONFLICTSEARCH=</td>
</tr>
<tr>
<td>Specifies the node selection strategy</td>
<td>NODESEL=</td>
</tr>
<tr>
<td>Enables use of variable priorities</td>
<td>PRIORITY=</td>
</tr>
<tr>
<td>Specifies the restarting strategy</td>
<td>RESTARTS=</td>
</tr>
<tr>
<td>Specifies the number of simplex iterations to perform on each variable</td>
<td>STRONGITER=</td>
</tr>
<tr>
<td>in the strong branching variable selection strategy</td>
<td></td>
</tr>
<tr>
<td>Specifies the number of candidates for the strong branching</td>
<td>STRONGLEN=</td>
</tr>
<tr>
<td>variable selection strategy</td>
<td></td>
</tr>
<tr>
<td>Specifies the level of symmetry detection</td>
<td>SYMMETRY=</td>
</tr>
<tr>
<td>Specifies the rule for selecting the branching variable</td>
<td>VAREL=</td>
</tr>
<tr>
<td><strong>Cut Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the cut level for all cuts</td>
<td>ALLCUTS=</td>
</tr>
<tr>
<td>Specifies the clique cut level</td>
<td>CUTCLIQUE=</td>
</tr>
<tr>
<td>Specifies the flow cover cut level</td>
<td>CUTFLOWCOVER=</td>
</tr>
<tr>
<td>Specifies the flow path cut level</td>
<td>CUTFLOWPATH=</td>
</tr>
<tr>
<td>Specifies the Gomory cut level</td>
<td>CUTGOMORY=</td>
</tr>
<tr>
<td>Specifies the generalized upper bound (GUB) cut level</td>
<td>CUTGUB=</td>
</tr>
<tr>
<td>Specifies the implied bounds cut level</td>
<td>CUTIMPLIED=</td>
</tr>
<tr>
<td>Specifies the knapsack cover cut level</td>
<td>CUTKNAPSACK=</td>
</tr>
<tr>
<td>Specifies the lift-and-project cut level</td>
<td>CUTLAP=</td>
</tr>
<tr>
<td>Specifies the mixed lifted 0-1 cut level</td>
<td>CUTMILIFTED=</td>
</tr>
<tr>
<td>Specifies the mixed integer rounding (MIR) cut level</td>
<td>CUTMIR=</td>
</tr>
</tbody>
</table>
### MILP Solver Options

This section describes the options that are recognized by the MILP solver in PROC OPTMODEL. These options can be specified after a forward slash (/) in the SOLVE statement, provided that the MILP solver is explicitly specified using a WITH clause. For example, the following line could appear in PROC OPTMODEL statements:

```plaintext
solve with milp / allcuts=aggressive maxnodes=10000 primalin;
```

### Presolve Option

**PRESOLVER=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE**

specifies the level of presolve processing. You can specify the following values:

- **AUTOMATIC** applies the default level of presolve processing.
- **NONE** disables the presolver.
- **BASIC** performs minimal presolve processing.
- **MODERATE** applies a higher level of presolve processing.
- **AGGRESSIVE** applies the highest level of presolve processing.

By default, PRESOLVER=AUTOMATIC.

### Table 14.1 (continued)

| Description                                                                 | Option                              |
|                                                                           |                                     |
| Specifies the multicommodity network flow cut level                       | CUTMULTICOMMODITY=                  |
| Specifies the row multiplier factor for cuts                              | CUTSFACOR=                          |
| Specifies the overall cut aggressiveness                                  | CUTSTRATEGY=                        |
| Specifies the zero-half cut level                                        | CUTZEROHALF=                        |

**Decomposition Algorithm Options**

Enables decomposition algorithm and specifies general control options

- `DECOMP=()` specifies options for the master problem
- `DECOMPMASTER=()` specifies options for the master problem solved as a MILP
- `DECOMPSUBPROB=()` specifies options for the subproblem

**Parallel Options**

Enables the MILP solver to run in concurrent mode

- `CONCURRENT=`
- `DETERMINISTIC=`
- `DISTRIBUTED=`

Enables the MILP solver to run deterministically

- `NTHREADS=` specifies the number of threads for the parallel MILP solver to use
Warm Start Option

PRIMALIN enables you to input a starting solution in PROC OPTMODEL before invoking the MILP solver. Adding the PRIMALIN option to the SOLVE statement requests that the MILP solver use the current variable values as a starting solution (warm start). If the MILP solver finds that the input solution is feasible, then the input solution provides an incumbent solution and a bound for the branch-and-bound algorithm. If the solution is not feasible, the MILP solver tries to repair it. It is possible to set a variable value to the missing value ‘.’ to mark a variable for repair. When it is difficult to find a good integer feasible solution for a problem, warm start can reduce solution time significantly.

NOTE: If the MILP solver produces a feasible solution, the variable values from that run can be used as the warm start solution for a subsequent run. If the warm start solution is not feasible for the subsequent run, the solver automatically tries to repair it.

Control Options

ABSOBJGAP=number

ABSOLUTEOBJECTIVEGAP=number
specifies a stopping criterion. When the absolute difference between the best integer objective and the best bound on the objective function value falls below the value of number, the MILP solver stops. The value of number can be any nonnegative number; the default value is 1E–6.

CUTOFF=number
cuts off any nodes in a minimization (maximization) problem that have an objective value at or above (below) number. The value of number can be any number; the default value is the largest (smallest) number that can be represented by a double.

EMPHASIS=BALANCE | OPTIMAL | FEASIBLE
specifies a search emphasis string as listed below.

BALANCE performs a balanced search.
OPTIMAL emphasizes optimality over feasibility.
FEASIBLE emphasizes feasibility over optimality.

By default, EMPHASIS=BALANCE.

FEASTOL=number
specifies the tolerance that the MILP solver uses to check the feasibility of a solution. This tolerance applies both to the maximum violation of bounds on variables and to the difference between the right-hand sides and left-hand sides of constraints. The value of number can be any value between 1E–4 and 1E–9, inclusive. However, the value of number cannot be larger than the integer feasibility tolerance. If the value of number is larger than the value of the INTTOL= option, then the solver sets FEASTOL= to the value of INTTOL=. The default value is 1E–6.

If the MILP solver fails to find a feasible solution within this tolerance but does find a solution that has some violation, then the solver stops with a solution status of OPTIMAL_COND (see the section “Macro Variable _OROPTMODEL_ ” on page 629).
INTTOL=number

specifies the amount by which an integer variable value can differ from an integer and still be considered integer feasible. The value of number can be any number between 1E–9 and 0.5, inclusive. The MILP solver attempts to find an optimal solution whose integer infeasibility is less than number. The default value is 1E–5.

If the best solution that the solver finds has an integer infeasibility larger than the value of number, then the solver stops with a solution status of OPTIMAL_COND (see the section “Macro Variable _OROPTMODEL_ ” on page 629).

LOGFREQ=k

prints information in the node log every k seconds, where k is any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If $k=0$, then the node log is disabled. If $k$ is positive, then the root node processing information is printed and, if possible, an entry is made every $k$ seconds. An entry is also made each time a better integer solution is found.

By default, LOGFREQ=5.

LOGLEVEL=NONE | BASIC | MODERATE | AGGRESSIVE

controls the amount of information displayed in the SAS log by the MILP solver. You can specify the following values:

NONE turns off all solver-related messages in the SAS log.

BASIC displays a solver summary after stopping.

MODERATE prints a solver summary and a node log by using the interval specified in the LOGFREQ= option.

AGGRESSIVE prints a detailed solver summary and a node log by using the interval specified in the LOGFREQ= option.

By default, LOGLEVEL=MODERATE.

MAXNODES=number

specifies the maximum number of branch-and-bound nodes to be processed. The value of number can be any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value of number is $2^{31} - 1$. If you run the MILP solver in concurrent mode, then the solver stops as soon as number is reached on any machine. If you run the MILP solver in distributed mode, then the solver periodically checks and stops as soon as the total number of nodes that are processed by all grid nodes exceeds number.

MAXPOOLSOLS=number

specifies the number of solutions to return from the MILP solution pool, where number can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The solutions are returned in order of objective value, with the best solution first. The default value of number is 1. The number of solutions that are found is reported in both the solution summary and the _OROPTMODEL_ macro variable.
**MAXSOLS=number**
specifies a stopping criterion, where *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If *number* of solutions have been found, then the solver stops. The default value of *number* is $2^{31} - 1$.

**MAXTIME=**
specifies an upper limit of *t* units of time for the optimization process, including problem generation time and solution time. The value of the TIMETYPE= option determines the type of units used. If you do not specify the MAXTIME= option, the solver does not stop based on the amount of time elapsed. If concurrent or distributed mode of the solver is enabled, then the solver stops as soon as *t* is reached on any machine. The value of *t* can be any positive number; the default value is the largest number that can be represented by a double.

**OPTTOL=number**
specifies the tolerance used to determine the optimality of nodes in the branch-and-bound tree. The value of *number* can be any value between (and including) 1E–4 and 1E–9. The default is 1E–6.

**PROBE=AUTOMATIC | NONE | MODERATE | AGGRESSIVE**
specifies a probing strategy. You can specify the following values:

- **AUTOMATIC** uses the probing strategy determined by the MILP solver.
- **NONE** disables probing.
- **MODERATE** uses the probing moderately.
- **AGGRESSIVE** uses probing aggressively.

By default, PROBE=AUTOMATIC.

**RELOBJGAP=number**
specifies a stopping criterion based on the best integer objective (BestInteger) and the best bound on the objective function value (BestBound). The relative objective gap is equal to

$$\frac{|\text{BestInteger} - \text{BestBound}|}{(1\times10^{-10} + |\text{BestBound}|)}$$

When this value becomes smaller than the specified gap size *number*, the MILP solver stops. The value of *number* can be any nonnegative number; the default value is 1E–4.

**SCALE=AUTOMATIC | NONE**
indicates whether to scale the problem matrix. You can specify the following values:

- **AUTOMATIC** scales the matrix as determined by the MILP solver.
- **NONE** disables scaling.

By default, SCALE=AUTOMATIC.

**SEED=number**
specifies the initial seed of the random number generator. This option affects the perturbation in the simplex solvers; thus it might result in a different optimal solution and a different solver path. This option usually has a significant, but unpredictable, effect on the solution time. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. By default, SEED=100.
TARGET=number
specifies a stopping criterion for a minimization or maximization problem. If the best integer objective is better than or equal to number, the solver stops. The value of number can be any number; the default value is the largest (in magnitude) negative number (for a minimization problem) or the largest (in magnitude) positive number (for a maximization problem) that can be represented by a double.

TIMETYPE=CPU | REAL
specifies the units of time used by the MAXTIME= option and reported by the PRESOLVE_TIME and SOLUTION_TIME terms in the _OROPTMODEL_ macro variable. You can specify the following values:

- CPU specifies that units are in CPU time.
- REAL specifies that units are in real time.

The “Optimization Statistics” table, an output of PROC OPTMODEL if you specify PRINTLEVEL=2 in the PROC OPTMODEL statement, also includes the same time units for Presolver Time and Solver Time. The other times (such as Problem Generation Time) in the “Optimization Statistics” table are also in the same units.

By default, TIMETYPE=REAL.

**Heuristics Option**

HEURISTICS=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE
controls the level of primal heuristics applied by the MILP solver. This level determines how frequently primal heuristics are applied during the branch-and-bound tree search. It also affects the maximum number of iterations allowed in iterative heuristics. Some computationally expensive heuristics might be disabled by the solver at less aggressive levels. You can specify the following values:

- AUTOMATIC applies the default level of heuristics, similar to MODERATE.
- NONE disables all primal heuristics. This value does not disable the heuristics that repair an infeasible input solution that is specified by using the PRIMALIN option.
- BASIC applies basic primal heuristics at low frequency.
- MODERATE applies most primal heuristics at moderate frequency.
- AGGRESSIVE applies all primal heuristics at high frequency.

By default, HEURISTICS=AUTOMATIC. For more information about primal heuristics, see the section “Primal Heuristics” on page 626.

**Search Options**

CONFLICTSEARCH=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the level of conflict search performed by the MILP solver. A conflict search finds clauses resulting from infeasible subproblems that arise in the search tree. You can specify the following values:
Chapter 14: The Mixed Integer Linear Programming Solver

AUTOMATIC performs conflict search based on a strategy determined by the MILP solver.
NONE disables conflict search.
MODERATE performs a moderate conflict search.
AGGRESSIVE performs an aggressive conflict search.

By default, CONFLICTSEARCH=AUTOMATIC.

NODESEL=AUTOMATIC | BESTBOUND | BESTESTIMATE | DEPTH
specifies the node selection strategy. You can specify the following values:

AUTOMATIC uses automatic node selection.
BESTBOUND chooses the node with the best relaxed objective (best-bound-first strategy).
BESTESTIMATE chooses the node with the best estimate of the integer objective value (best-estimate-first strategy).
DEPTH chooses the most recently created node (depth-first strategy).

By default, NODESEL=AUTOMATIC. For more information about node selection, see the section “Node Selection” on page 622.

PRIORITY=TRUE | FALSE
indicates whether to use specified branching priorities for integer variables. You can specify the following values:

TRUE uses priorities when they exist.
FALSE ignores variable priorities.

By default, PRIORITY=TRUE. For more information, see the section “Branching Priorities” on page 624.

RESTARTS=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE
specifies the strategy for restarting the processing of the root node. You can specify the following values:

AUTOMATIC uses a restarting strategy determined by the MILP solver.
NONE disables restarting.
BASIC uses a basic restarting strategy.
MODERATE uses a moderate restarting strategy.
AGGRESSIVE uses an aggressive restarting strategy.

By default, RESTARTS=AUTOMATIC.

STRONGITER=number | AUTOMATIC
specifies the number of simplex iterations performed for each variable in the candidate list when the strong branching variable selection strategy is used. The value of number can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC, the MILP solver uses the default value; this value is calculated automatically.
**STRONGLEN=**<em>number</em> | **AUTOMATIC**

specifies the number of candidates used when the strong branching variable selection strategy is performed. The value of <em>number</em> can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC, the MILP solver uses the default value; this value is calculated automatically.

**SYMMETRY=**<em>AUTOMATIC</em> | **NONE** | **BASIC** | **MEDIUM** | **AGGRESSIVE**

specifies the level of symmetry detection. Symmetry detection identifies groups of equivalent decision variables and uses this information to solve the problem more efficiently. You can specify the following values:

- **AUTOMATIC** performs symmetry detection based on a strategy that is determined by MILP solver.
- **NONE** disables symmetry detection.
- **BASIC** performs a basic symmetry detection.
- **MEDIUM** performs a moderate symmetry detection.
- **AGGRESSIVE** performs an aggressive symmetry detection.

By default, SYMMETRY=AUTOMATIC. For more information about symmetry detection, see (Ostrowski 2008).

**VARSEL=**<em>AUTOMATIC</em> | **MAXINEAS** | **MININEAS** | **PSEUDO** | **STRONG**

specifies the rule for selecting the branching variable. You can specify the following values:

- **AUTOMATIC** uses automatic branching variable selection.
- **MAXINEAS** chooses the variable with maximum infeasibility.
- **MININEAS** chooses the variable with minimum infeasibility.
- **PSEUDO** chooses a branching variable based on pseudocost.
- **STRONG** uses a strong branching variable selection strategy.

By default, VARSEL=AUTOMATIC. For more information about variable selection, see the section “Variable Selection” on page 623.

### Cut Options

Table 14.2 describes the <em>string</em> values for the cut options in the OPTMODEL procedure.

<table>
<thead>
<tr>
<th>&lt;em&gt;string&lt;/em&gt;</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOMATIC</td>
<td>Generates cutting planes based on a strategy determined by the MILP solver</td>
</tr>
<tr>
<td>NONE</td>
<td>Disables generation of cutting planes</td>
</tr>
<tr>
<td>MODERATE</td>
<td>Uses a moderate cut strategy</td>
</tr>
<tr>
<td>AGGRESSIVE</td>
<td>Uses an aggressive cut strategy</td>
</tr>
</tbody>
</table>
You can specify the **CUTSTRATEGY=** option to set the overall aggressiveness of the cut generation in the MILP solver. Alternatively, you can use the **ALLCUTS=** option to set all cut types to the same level. You can override the **ALLCUTS=** value by using the options that correspond to particular cut types. For example, if you want the MILP solver to generate only Gomory cuts, specify **ALLCUTS=NONE** and **CUTGOMORY=AUTOMATIC**. If you want to generate all cuts aggressively but generate no lift-and-project cuts, set **ALLCUTS=AGGRESSIVE** and **CUTLAP=NONE**.

**ALLCUTS=AUTOMATIC | NONE | MODERATE | AGGRESSIVE**

provides a shorthand way of setting all the cuts-related options in one setting. In other words, **ALLCUTS=** *string* is equivalent to setting each of the individual cuts parameters to the same value *string*. Thus, **ALLCUTS=** *string* has the effect of setting **CUTCLIQUE=AUTOMATIC**, **CUTFLOWCOVER=AUTOMATIC**, **CUTFLOWPATH=AUTOMATIC**, . . . , **CUTMULTICOMMODITY=AUTOMATIC**, and **CUTZEROHALF=AUTOMATIC**. Table 14.2 lists the values that can be assigned to *option*. In addition, you can override levels for individual cuts with the **CUTCLIQUE=**, **CUTFLOWCOVER=**, **CUTFLOWPATH=**, **CUTGOMORY=**, **CUTGUB=**, **CUTIMPLIED=**, **CUTKNAPSACK=**, **CUTLAP=**, **CUTMILIFTED=**, **CUTMIR=**, **CUTMULTICOMMODITY=**, and **CUTZEROHALF=** options. If the **ALLCUTS=** option is not specified, then all the cuts-related options are either at their individually specified values (if the corresponding option is specified) or at their default values (if that option is not specified).

**CUTCLIQUE=AUTOMATIC | NONE | MODERATE | AGGRESSIVE**

specifies the level of clique cuts that are generated by the MILP solver. Table 14.2 describes the possible values. This option overrides the **ALLCUTS=** option. By default, **CUTCLIQUE=AUTOMATIC**.

**CUTFLOWCOVER=AUTOMATIC | NONE | MODERATE | AGGRESSIVE**

specifies the level of flow cover cuts that are generated by the MILP solver. Table 14.2 describes the possible values. The option overrides the **ALLCUTS=** option. By default, **CUTFLOWCOVER=AUTOMATIC**.

**CUTFLOWPATH=AUTOMATIC | NONE | MODERATE | AGGRESSIVE**

specifies the level of flow path cuts that are generated by the MILP solver. Table 14.2 describes the possible values. This option overrides the **ALLCUTS=** option. By default, **CUTFLOWPATH=AUTOMATIC**.

**CUTGOMORY=AUTOMATIC | NONE | MODERATE | AGGRESSIVE**

specifies the level of Gomory cuts that are generated by the MILP solver. Table 14.2 describes the possible values. This option overrides the **ALLCUTS=** option. By default, **CUTGOMORY=AUTOMATIC**.

**CUTGUB=AUTOMATIC | NONE | MODERATE | AGGRESSIVE**

specifies the level of generalized upper bound (GUB) cover cuts that are generated by the MILP solver. Table 14.2 describes the possible values. This option overrides the **ALLCUTS=** option. By default, **CUTGUB=AUTOMATIC**.

**CUTIMPLIED=AUTOMATIC | NONE | MODERATE | AGGRESSIVE**

specifies the level of implied bound cuts that are generated by the MILP solver. Table 14.2 describes the possible values. This option overrides the **ALLCUTS=** option. By default, **CUTIMPLIED=AUTOMATIC**.
CUTKNAPSACK=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the level of knapsack cover cuts that are generated by the MILP solver. Table 14.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTKNAPSACK=AUTOMATIC.

CUTLAP=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the level of lift-and-project (LAP) cuts that are generated by the MILP solver. Table 14.2 describes the possible values that can be assigned to option. This option overrides the ALLCUTS= option. By default, CUTLAP=NONE.

CUTMILIFTED=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the level of mixed lifted 0-1 cuts that are generated by the MILP solver. Table 14.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTMILIFTED=AUTOMATIC.

CUTMIR=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the level of mixed integer rounding (MIR) cuts that are generated by the MILP solver. Table 14.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTMIR=AUTOMATIC.

CUTMULTICOMMODITY=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the level of multicommodity network flow cuts that are generated by the MILP solver. Table 14.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTMULTICOMMODITY=AUTOMATIC.

CUTSFACtor=number
specifies a row multiplier factor for cuts. The number of cuts that are added is limited to number times the original number of rows. The value of number can be any nonnegative number less than or equal to 100; the default value is automatically calculated by the MILP solver.

CUTSTRATEGY=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
CUTS=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the overall aggressiveness of the cut generation in the solver. By default, CUTSTRATEGY=AUTOMATIC. Setting a nondefault value adjusts a number of cut parameters such that the cut generation is none, moderate, or aggressive compared to the default value.

CUTZEROHALF=AUTOMATIC | NONE | MODERATE | AGGRESSIVE
specifies the level of zero-half cuts that are generated by the MILP solver. Table 14.2 describes the possible values. This option overrides the ALLCUTS= option. By default, CUTZEROHALF=AUTOMATIC.

Decomposition Algorithm Options

The following options are available for the decomposition algorithm in the MILP solver. For information about the decomposition algorithm, see Chapter 18, “The Decomposition Algorithm.”

DECOMP=(options)
enables the decomposition algorithm and specifies overall control options for the algorithm. For more information about this option, see Chapter 18, “The Decomposition Algorithm.”
DECOMPMASTER=(options)
specifies options for the master problem. For more information about this option, see Chapter 18, “The Decomposition Algorithm.”

DECOMPMASTERIP=(options)
specifies options for the (restricted) master problem solved as a MILP with the current set of columns in an effort to obtain an integer feasible solution. For more information about this option, see Chapter 18, “The Decomposition Algorithm.”

DECOMPSUBPROB=(options)
specifies option for the subproblem. For more information about this option, see Chapter 18, “The Decomposition Algorithm.”

Parallel Options

CONCURRENT=TRUE | FALSE
specifies whether to enable concurrent mode for the MILP solver. You can specify the following values:

TRUE runs the MILP solver in concurrent mode. The solver automatically runs the same problem on each grid node with different MILP solver options.

FALSE runs the default MILP solver.

By default, CONCURRENT=FALSE.

DETERMINISTIC=TRUE | FALSE
specifies whether to enable the MILP solver to run deterministically when distributed or concurrent mode is enabled. The DETERMINISTIC= option is valid only when the DISTRIBUTED= or CONCURRENT= option is specified. You can specify the following values:

TRUE runs the MILP solver in deterministic distributed or concurrent mode. The solver always returns the same solution for subsequent runs on the same grid.

FALSE runs the MILP solver in nondeterministic distributed or concurrent mode.

By default, DETERMINISTIC=FALSE when you specify the CONCURRENT= option, and DETERMINISTIC=TRUE when you specify the DISTRIBUTED= option.

DISTRIBUTED=TRUE | FALSE
specifies whether to run the MILP solver in distributed mode. For more information about this option as it relates to the standard MILP algorithm, see the section “Parallel Processing” on page 626. For more information about this option as it relates to the decomposition algorithm, see Chapter 18, “The Decomposition Algorithm.” You can specify the following values:

TRUE runs the MILP solver in distributed mode.

FALSE turns off distributed mode of the MILP solver.

By default, DISTRIBUTED=FALSE.
**NTHREADS=** *number*

specifies the maximum number of threads for the MILP solver to use for multithreaded processing, where *number* can be any integer between 1 and 256, inclusive. The branch-and-cut algorithm can take advantage of multicore machines and can potentially run faster when *number* is greater than 1. The default is the value of the `NTHREADS=` option in PROC OPTMODEL.

---

**Details: MILP Solver**

**Branch-and-Bound Algorithm**

The branch-and-bound algorithm, first proposed by Land and Doig (1960), is an effective approach to solving mixed integer linear programs. The following discussion outlines the approach and explains how to enhance its progress by using several advanced techniques.

The branch-and-bound algorithm solves a mixed integer linear program by dividing the search space and generating a sequence of subproblems. The search space of a mixed integer linear program can be represented by a tree. Each node in the tree is identified with a subproblem derived from previous subproblems on the path that leads to the root of the tree. The subproblem (MILP) associated with the root is identical to the original problem, which is called (MILP), given in the section “Overview: MILP Solver” on page 607.

The linear programming relaxation (LP) of (MILP) can be written as

\[
\begin{align*}
\min & \quad c^T x \\
\text{subject to} & \quad Ax \{\geq, =, \leq\} b \\
& \quad 1 \leq x \leq u
\end{align*}
\]

The branch-and-bound algorithm generates subproblems along the nodes of the tree by using the following scheme. Consider \( \bar{x}^0 \), the optimal solution to (LP), which is usually obtained by using the dual simplex algorithm. If \( \bar{x}^0_i \) is an integer for all \( i \in S \), then \( \bar{x}^0 \) is an optimal solution to (MILP). Suppose that for some \( i \in S \), \( \bar{x}^0_i \) is nonintegral. In that case the algorithm defines two new subproblems (MILP\(^1\)) and (MILP\(^2\)), descendants of the parent subproblem (MILP). The subproblem (MILP\(^1\)) is identical to (MILP) except for the additional constraint

\[ x_i \leq \lfloor \bar{x}^0_i \rfloor \]

and the subproblem (MILP\(^2\)) is identical to (MILP) except for the additional constraint

\[ x_i \geq \lceil \bar{x}^0_i \rceil \]

The notation \( \lfloor y \rfloor \) represents the largest integer that is less than or equal to \( y \), and the notation \( \lceil y \rceil \) represents the smallest integer that is greater than or equal to \( y \). The two preceding constraints can be handled by modifying the bounds of the variable \( x_i \) rather than by explicitly adding the constraints to the constraint matrix. The two new subproblems do not have \( \bar{x}^0 \) as a feasible solution, but the integer solution to (MILP) must satisfy one of the preceding constraints. The two subproblems thus defined are called *active nodes* in the branch-and-bound tree, and the variable \( x_i \) is called the *branching variable*. 
In the next step the branch-and-bound algorithm chooses one of the active nodes and attempts to solve the linear programming relaxation of that subproblem. The relaxation might be infeasible, in which case the subproblem is dropped (fathomed). If the subproblem can be solved and the solution is integer feasible (that is, \( x_i \) is an integer for all \( i \in S \)), then its objective value provides an upper bound for the objective value in the minimization problem (MILP); if the solution is not integer feasible, then it defines two new subproblems. Branching continues in this manner until there are no active nodes. At this point the best integer solution found is an optimal solution for (MILP). If no integer solution has been found, then (MILP) is integer infeasible. You can specify other criteria to stop the branch-and-bound algorithm before it processes all the active nodes; see the section “Controlling the Branch-and-Bound Algorithm” on page 622 for details.

Upper bounds from integer feasible solutions can be used to fathom or cut off active nodes. Since the objective value of an optimal solution cannot be greater than an upper bound, active nodes with lower bounds higher than an existing upper bound can be safely deleted. In particular, if \( z \) is the objective value of the current best integer solution, then any active subproblems whose relaxed objective value is greater than or equal to \( z \) can be discarded.

It is important to realize that mixed integer linear programs are nondeterministic polynomial-time hard (NP-hard). Roughly speaking, this means that the effort required to solve a mixed integer linear program grows exponentially with the size of the problem. For example, a problem with 10 binary variables can generate in the worst case \( 2^{10} = 1,024 \) nodes in the branch-and-bound tree. A problem with 20 binary variables can generate in the worst case \( 2^{20} = 1,048,576 \) nodes in the branch-and-bound tree. Although it is unlikely that the branch-and-bound algorithm has to generate every single possible node, the need to explore even a small fraction of the potential number of nodes for a large problem can be resource-intensive.

A number of techniques can speed up the search progress of the branch-and-bound algorithm. Heuristics are used to find feasible solutions, which can improve the upper bounds on solutions of mixed integer linear programs. Cutting planes can reduce the search space and thus improve the lower bounds on solutions of mixed integer linear programs. When using cutting planes, the branch-and-bound algorithm is also called the branch-and-cut algorithm. Preprocessing can reduce problem size and improve problem solvability. The MILP solver in PROC OPTMODEL employs various heuristics, cutting planes, preprocessing, and other techniques, which you can control through corresponding options.

### Controlling the Branch-and-Bound Algorithm

There are numerous strategies that can be used to control the branch-and-bound search (see Linderoth and Savelsbergh 1998, Achterberg, Koch, and Martin 2005). The MILP solver in PROC OPTMODEL implements the most widely used strategies and provides several options that enable you to direct the choice of the next active node and of the branching variable. In the discussion that follows, let \((LP^k)\) be the linear programming relaxation of subproblem \((MILP^k)\). Also, let

\[
f_i(k) = \hat{x}^k_i - \lfloor \hat{x}^k_i \rfloor
\]

where \(\hat{x}^k\) is the optimal solution to the relaxation problem \((LP^k)\) solved at node \(k\).

### Node Selection

The NODESEL= option specifies the strategy used to select the next active node. The valid keywords for this option are AUTOMATIC, BESTBOUND, BESTESTIMATE, and DEPTH. The following list describes the strategy associated with each keyword:
AUTOMATIC enables the MILP solver to choose the best node selection strategy based on problem characteristics and search progress. This is the default setting.

BESTBOUND chooses the node with the smallest (or largest, in the case of a maximization problem) relaxed objective value. The best-bound strategy tends to reduce the number of nodes to be processed and can improve lower bounds quickly. However, if there is no good upper bound, the number of active nodes can be large. This can result in the solver running out of memory.

BESTESTIMATE chooses the node with the smallest (or largest, in the case of a maximization problem) objective value of the estimated integer solution. Besides improving lower bounds, the best-estimate strategy also attempts to process nodes that can yield good feasible solutions.

DEPTH chooses the node that is deepest in the search tree. Depth-first search is effective in locating feasible solutions, since such solutions are usually deep in the search tree. Compared to the costs of the best-bound and best-estimate strategies, the cost of solving LP relaxations is less in the depth-first strategy. The number of active nodes is generally small, but it is possible that the depth-first search will remain in a portion of the search tree with no good integer solutions. This occurrence is computationally expensive.

Variable Selection

The VARSEL= option specifies the strategy used to select the next branching variable. The valid keywords for this option are AUTOMATIC, MAXINFEAS, MININFEAS, PSEUDO, and STRONG. The following list describes the action taken in each case when \( \bar{x}^k \), a relaxed optimal solution of (MILP\(^k\)), is used to define two active subproblems. In the following list, “INTTOL” refers to the value assigned using the INTTOL= option. For details about the INTTOL= option, see the section “Control Options” on page 612.

AUTOMATIC enables the MILP solver to choose the best variable selection strategy based on problem characteristics and search progress. This is the default setting.

MAXINFEAS chooses as the branching variable the variable \( x_i \) such that \( i \) maximizes

\[
\{ \min \{ f_i(k), 1 - f_i(k) \} \mid i \in S \text{ and } \text{INTTOL} \leq f_i(k) \leq 1 - \text{INTTOL} \}
\]

MININFEAS chooses as the branching variable the variable \( x_i \) such that \( i \) minimizes

\[
\{ \min \{ f_i(k), 1 - f_i(k) \} \mid i \in S \text{ and } \text{INTTOL} \leq f_i(k) \leq 1 - \text{INTTOL} \}
\]

PSEUDO chooses as the branching variable the variable \( x_i \) such that \( i \) maximizes the weighted up and down pseudocosts. Pseudocost branching attempts to branch on significant variables first, quickly improving lower bounds. Pseudocost branching estimates significance based on historical information; however, this approach might not be accurate for future search.
 Chapter 14: The Mixed Integer Linear Programming Solver

STRONG chooses as the branching variable the variable \( x_i \) such that \( i \) maximizes the estimated improvement in the objective value. Strong branching first generates a list of candidates, then branches on each candidate and records the improvement in the objective value. The candidate with the largest improvement is chosen as the branching variable. Strong branching can be effective for combinatorial problems, but it is usually computationally expensive.

Branching Priorities

In some cases, it is possible to speed up the branch-and-bound algorithm by branching on variables in a specific order. You can accomplish this in PROC OPTMODEL by attaching branching priorities to the integer variables in your model by using the .priority suffix. More information about this suffix is available in the section “Integer Variable Suffixes” on page 384 in Chapter 10. For an example in which branching priorities are used, see Example 14.3.

Presolve and Probing

The MILP solver in PROC OPTMODEL includes a variety of presolve techniques to reduce problem size, improve numerical stability, and detect infeasibility or unboundedness (Andersen and Andersen 1995; Gondzio 1997). During presolve, redundant constraints and variables are identified and removed. Presolve can further reduce the problem size by substituting variables. Variable substitution is a very effective technique, but it might occasionally increase the number of nonzero entries in the constraint matrix. Presolve might also modify the constraint coefficients to tighten the formulation of the problem.

In most cases, using presolve is very helpful in reducing solution times. You can enable presolve at different levels by specifying the PRESOLVER= option.

Probing is a technique that tentatively sets each binary variable to 0 or 1, then explores the logical consequences (Savelsbergh 1994). Probing can expedite the solution of a difficult problem by fixing variables and improving the model. However, probing is often computationally expensive and can significantly increase the solution time in some cases. You can enable probing at different levels by specifying the PROBE= option.

Cutting Planes

The feasible region of every linear program forms a polyhedron. Every polyhedron in \( n \)-space can be written as a finite number of half-spaces (equivalently, inequalities). In the notation used in this chapter, this polyhedron is defined by the set \( Q = \{ x \in \mathbb{R}^n \mid A x \leq b, l \leq x \leq u \} \). After you add the restriction that some variables must be integral, the set of feasible solutions, \( F = \{ x \in Q \mid x_i \in \mathbb{Z} \ \forall i \in S \} \), no longer forms a polyhedron.

The convex hull of a set \( X \) is the minimal convex set that contains \( X \). In solving a mixed integer linear program, in order to take advantage of LP-based algorithms you want to find the convex hull, \( \text{conv}(F) \), of \( F \). If you can find \( \text{conv}(F) \) and describe it compactly, then you can solve a mixed integer linear program with a linear programming solver. This is generally very difficult, so you must be satisfied with finding an approximation. Typically, the better the approximation, the more efficiently the LP-based branch-and-bound algorithm can perform.
As described in the section “Branch-and-Bound Algorithm” on page 621, the branch-and-bound algorithm begins by solving the linear programming relaxation over the polyhedron \( \mathcal{Q} \). Clearly, \( \mathcal{Q} \) contains the convex hull of the feasible region of the original integer program; that is, \( \text{conv}(\mathcal{F}) \subseteq \mathcal{Q} \).

Cutting plane techniques are used to tighten the linear relaxation to better approximate \( \text{conv}(\mathcal{F}) \). Assume you are given a solution \( \bar{x} \) to some intermediate linear relaxation during the branch-and-bound algorithm. A cut, or valid inequality \((\pi x \leq \pi^0, \forall x \in \mathcal{F})\), is some half-space with the following characteristics:

- The half-space contains \( \text{conv}(\mathcal{F}) \); that is, every integer feasible solution is feasible for the cut \((\pi x \leq \pi^0, \forall x \in \mathcal{F})\).
- The half-space does not contain the current solution \( \bar{x} \); that is, \( \bar{x} \) is not feasible for the cut \((\pi \bar{x} > \pi^0)\).

Cutting planes were first made popular by Dantzig, Fulkerson, and Johnson (1954) in their work on the traveling salesman problem. The two major classifications of cutting planes are generic cuts and structured cuts. Generic cuts are based solely on algebraic arguments and can be applied to any relaxation of any integer program. Structured cuts are specific to certain structures that can be found in some relaxations of the mixed integer linear program. These structures are automatically discovered during the cut initialization phase of the MILP solver. Table 14.3 lists the various types of cutting planes that are built into the MILP solver. Included in each type are algorithms for numerous variations based on different relaxations and lifting techniques. For a survey of cutting plane techniques for mixed integer programming, see Marchand et al. (1999). For a survey of lifting techniques, see Atamturk (2004).

### Table 14.3 Cutting Planes in the MILP Solver

<table>
<thead>
<tr>
<th>Generic Cutting Planes</th>
<th>Structured Cutting Planes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gomory mixed integer</td>
<td>Cliques</td>
</tr>
<tr>
<td>Lift-and-project</td>
<td>Flow cover</td>
</tr>
<tr>
<td>Mixed integer rounding</td>
<td>Flow path</td>
</tr>
<tr>
<td>Mixed lifted 0-1</td>
<td>Generalized upper bound cover</td>
</tr>
<tr>
<td>Zero-half</td>
<td>Implied bound</td>
</tr>
<tr>
<td></td>
<td>Knapsack cover</td>
</tr>
<tr>
<td></td>
<td>Multicommodity network flow</td>
</tr>
</tbody>
</table>

You can set levels for individual cuts by using the `CUTCLIQUE=`, `CUTFLOWCOVER=`, `CUTFLOWPATH=`, `CUTGOMORY=`, `CUTGUB=`, `CUTIMPLIED=`, `CUTKNAPSACK=`, `CUTLAP=`, `CUTMILIFTED=`, `CUTMIR=`, `CUTMULTICOMMODITY=`, and `CUTZEROHALF=` options. The valid levels for these options are listed in Table 14.2.

The cut level determines the internal strategy that is used by the MILP solver for generating the cutting planes. The strategy consists of several factors, including how frequently the cut search is called, the number of cuts allowed, and the aggressiveness of the search algorithms.

Sophisticated cutting planes, such as those included in the MILP solver, can take a great deal of CPU time. Usually, additional tightening of the relaxation helps speed up the overall process because it provides better bounds for the branch-and-bound tree and helps guide the LP solver toward integer solutions. In rare cases, shutting off cutting planes completely might lead to faster overall run times.

The default settings of the MILP solver have been tuned to work well for most instances. However, problem-specific expertise might suggest adjusting one or more of the strategies. These options give you that flexibility.
Primal Heuristics

Primal heuristics, an important component of the MILP solver in PROC OPTMODEL, are applied during the branch-and-bound algorithm. They are used to find integer feasible solutions early in the search tree, thereby improving the upper bound for a minimization problem. Primal heuristics play a role that is complementary to cutting planes in reducing the gap between the upper and lower bounds, thus reducing the size of the branch-and-bound tree.

Applying primal heuristics in the branch-and-bound algorithm assists in the following areas:

- finding a good upper bound early in the tree search (this can lead to earlier fathoming, resulting in fewer subproblems to be processed)
- locating a reasonably good feasible solution when that is sufficient (sometimes a reasonably good feasible solution is the best the solver can produce within certain time or resource limits)
- providing upper bounds for some bound-tightening techniques

The MILP solver implements several heuristic methodologies. Some algorithms, such as rounding and iterative rounding (diving) heuristics, attempt to construct an integer feasible solution by using fractional solutions to the continuous relaxation at each node of the branch-and-cut tree. Other algorithms start with an incumbent solution and attempt to find a better solution within a neighborhood of the current best solution.

The HEURISTICS= option enables you to control the level of primal heuristics that are applied by the MILP solver. This level determines how frequently primal heuristics are applied during the tree search. Some expensive heuristics might be disabled by the solver at less aggressive levels. Setting the HEURISTICS= option to a lower level also reduces the maximum number of iterations that are allowed in iterative heuristics.

Parallel Processing

You can run the branch-and-cut algorithm in single-machine mode, distributed mode, or concurrent mode. In single-machine mode, the computation is executed on a single computer by multiple threads. The number of threads is specified by the NTHREADS= option. In distributed mode, the branch-and-cut algorithm is executed on grid nodes. In concurrent mode, the computation is executed on grid nodes by solving the same problem across grid nodes under different MILP solver settings. If distributed mode is enabled, then by default the MILP solver is executed deterministically. You disable deterministic distributed mode by setting the DETERMINISTIC= option to FALSE. If concurrent mode is enabled, then by default the MILP solver is executed nondeterministically.

You can run the decomposition algorithm in either single-machine or distributed mode. In distributed mode, the computation is executed on multiple computing nodes in a distributed computing environment.
The following information about the status of the branch-and-bound algorithm is printed in the node log:

Node indicates the sequence number of the current node in the search tree.
Active indicates the current number of active nodes in the branch-and-bound tree.
Sols indicates the number of feasible solutions found so far. You can use the MAXPOOL-SOLS= option to obtain more than one solution. Note that the number printed in the node log can be different from the number of solutions found that is reported in the solution summary and in the _OROPTMODEL_ macro variable.

BestInteger indicates the best upper bound (assuming minimization) found so far.
BestBound indicates the best lower bound (assuming minimization) found so far.
Gap indicates the relative gap between BestInteger and BestBound, displayed as a percentage. If the relative gap is larger than 1,000, then the absolute gap is displayed. If no active nodes remain, the value of Gap is 0.
Time indicates the elapsed real or CPU time.

If the CONCURRENT= option is set to TRUE, then the following information about the status of the branch-and-bound algorithm is printed in the node log:

Node indicates the total number of nodes processed by all concurrent solves.
Active indicates the minimum number of active nodes among all concurrent solves.
Sols indicates the number of feasible solutions found by all concurrent solves.
BestInteger indicates the best upper bound (assuming minimization) found by all concurrent solves.
BestBound indicates the best lower bound (assuming minimization) found by all concurrent solves.
Gap indicates the relative gap between BestInteger and BestBound, displayed as a percentage. If the relative gap is larger than 1,000, then the absolute gap is displayed. If no active nodes remain, the value of Gap is 0.
Time indicates the elapsed real or CPU time.

If CONCURRENT=TRUE, the node log displays the progress of the overall grid. If DETERMINISTIC=TRUE, the solver tries to return a reproducible solution. Therefore, there might be discrepancies between the node log and the final solution that is reported by the deterministic concurrent MILP solver.

The LOGFREQ= option can be used to control the amount of information printed in the node log. By default, the root node processing information is printed and, if possible, an entry is made every five seconds. A new entry is also included each time a better integer solution is found. The LOGFREQ= option enables you to change the interval between entries in the node log. Figure 14.3 shows a sample node log.
Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of $10^9$) can cause difficulty when the remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTMODEL procedure causes the ODS table ProblemStatistics to be generated when the MILP solver is called. This table provides basic data magnitude information that enables you to improve the formulation of your models.
The example output in Figure 14.4 demonstrates the contents of the ODS table ProblemStatistics.

**Figure 14.4** ODS Table ProblemStatistics

<table>
<thead>
<tr>
<th>Obs</th>
<th>Label1</th>
<th>cValue1</th>
<th>nValue1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number of Constraint Matrix Nonzeros</td>
<td>8</td>
<td>8.000000</td>
</tr>
<tr>
<td>2</td>
<td>Maximum Constraint Matrix Coefficient</td>
<td>3</td>
<td>3.000000</td>
</tr>
<tr>
<td>3</td>
<td>Minimum Constraint Matrix Coefficient</td>
<td>1</td>
<td>1.000000</td>
</tr>
<tr>
<td>4</td>
<td>Average Constraint Matrix Coefficient</td>
<td>1.875</td>
<td>1.875000</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Number of Objective Nonzeros</td>
<td>3</td>
<td>3.000000</td>
</tr>
<tr>
<td>7</td>
<td>Maximum Objective Coefficient</td>
<td>4</td>
<td>4.000000</td>
</tr>
<tr>
<td>8</td>
<td>Minimum Objective Coefficient</td>
<td>2</td>
<td>2.000000</td>
</tr>
<tr>
<td>9</td>
<td>Average Objective Coefficient</td>
<td>3</td>
<td>3.000000</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Number of RHS Nonzeros</td>
<td>3</td>
<td>3.000000</td>
</tr>
<tr>
<td>12</td>
<td>Maximum RHS</td>
<td>7</td>
<td>7.000000</td>
</tr>
<tr>
<td>13</td>
<td>Minimum RHS</td>
<td>4</td>
<td>4.000000</td>
</tr>
<tr>
<td>14</td>
<td>Average RHS</td>
<td>5.3333333333</td>
<td>5.3333333333</td>
</tr>
<tr>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>Maximum Number of Nonzeros per Column</td>
<td>3</td>
<td>3.000000</td>
</tr>
<tr>
<td>17</td>
<td>Minimum Number of Nonzeros per Column</td>
<td>2</td>
<td>2.000000</td>
</tr>
<tr>
<td>18</td>
<td>Average Number of Nonzeros per Column</td>
<td>2.67</td>
<td>2.666667</td>
</tr>
<tr>
<td>19</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>Maximum Number of Nonzeros per Row</td>
<td>3</td>
<td>3.000000</td>
</tr>
<tr>
<td>21</td>
<td>Minimum Number of Nonzeros per Row</td>
<td>2</td>
<td>2.000000</td>
</tr>
<tr>
<td>22</td>
<td>Average Number of Nonzeros per Row</td>
<td>2.67</td>
<td>2.666667</td>
</tr>
</tbody>
</table>

The variable names in the ODS table ProblemStatistics are Label1, cValue1, and nValue1.

---

**Macro Variable _OROPTMODEL_**

The OPTMODEL procedure defines a macro variable named _OROPTMODEL_. This variable contains a character string that indicates the status of the solver upon termination. The contents of the macro variable depend on which solver was invoked. For the MILP solver, the various terms of _OROPTMODEL_ are interpreted as follows.

**STATUS**

indicates the solver status at termination. It can take one of the following values:

- **OK** The solver terminated normally.
- **SYNTAX_ERROR** Syntax was used incorrectly.
- **DATA_ERROR** The input data was inconsistent.
- **OUT_OF_MEMORY** Insufficient memory was allocated to the solver.
IO_ERROR       A problem occurred in reading or writing data.
SEMANTIC_ERROR An evaluation error, such as an invalid operand type, was found.
ERROR          The status cannot be classified into any of the preceding categories.

**ALGORITHM**
indicates the algorithm that produced the solution data in the macro variable. This term only appears when STATUS=OK. It can take one of the following values:

BAC             The branch-and-cut algorithm produced the solution data.
DECOMP          The decomposition algorithm produced the solution data.

**SOLUTION_STATUS**
indicates the solution status at termination. It can take one of the following values:

OPTIMAL         The solution is optimal.
OPTIMAL_AGAP    The solution is optimal within the absolute gap specified by the
                ABSOBJGAP= option.
OPTIMAL_RGAP    The solution is optimal within the relative gap specified by the
                RELOBJGAP= option.
OPTIMAL_COND    The solution is optimal, but some infeasibilities (primal, bound, or integer) exceed tolerances due to scaling or choice of small
                INTTOL= value.
TARGET          The solution is not worse than the target specified by the TARGET= option.
INFEASIBLE      The problem is infeasible.
UNBOUNDED       The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED The problem is infeasible or unbounded.
BAD_PROBLEM_TYPE The problem type is unsupported by solver.
SOLUTION_LIM    The solver reached the maximum number of solutions specified
                by the MAXSOLS= option.
NODE_LIM_SOL    The solver reached the maximum number of nodes specified by
                the MAXNODES= option and found a solution.
NODE_LIM_NOSOL  The solver reached the maximum number of nodes specified by
                the MAXNODES= option and did not find a solution.
TIME_LIM_SOL    The solver reached the execution time limit specified by the
                MAXTIME= option and found a solution.
TIME_LIM_NOSOL  The solver reached the execution time limit specified by the
                MAXTIME= option and did not find a solution.
ABORT_SOL       The solver was stopped by user but still found a solution.
ABORT_NOSOL     The solver was stopped by user and did not find a solution.
OUTMEM_SOL      The solver ran out of memory but still found a solution.
OUTMEM_NOSOL
The solver ran out of memory and either did not find a solution or failed to output the solution due to insufficient memory.

FAIL_SOL
The solver stopped due to errors but still found a solution.

FAIL_NOSOL
The solver stopped due to errors and did not find a solution.

OBJECTIVE
indicates the objective value obtained by the solver at termination.

RELATIVE_GAP
indicates the relative gap between the best integer objective (BestInteger) and the best bound on the objective function value (BestBound) upon termination of the MILP solver. The relative gap is equal to

$$\frac{|\text{BestInteger} - \text{BestBound}|}{10 + |\text{BestBound}|}$$

ABSOLUTE_GAP
indicates the absolute gap between the best integer objective (BestInteger) and the best bound on the objective function value (BestBound) upon termination of the MILP solver. The absolute gap is equal to $|\text{BestInteger} - \text{BestBound}|$.

PRIMAL_INFEASIBILITY
indicates the maximum (absolute) violation of the primal constraints by the solution.

BOUND_INFEASIBILITY
indicates the maximum (absolute) violation by the solution of the lower or upper bounds (or both).

INTEGER_INFEASIBILITY
indicates the maximum (absolute) violation of the integrality of integer variables returned by the MILP solver.

BEST_BOUND
indicates the best bound on the objective function value at termination. A missing value indicates that the MILP solver was not able to obtain such a bound.

NODES
indicates the number of nodes enumerated by the MILP solver by using the branch-and-bound algorithm.

SOLUTIONS_FOUND
indicates the number of solutions that are found by the MILP solver.

ITERATIONS
indicates the number of simplex iterations taken to solve the problem.

PRESOLVE_TIME
indicates the time (in seconds) used in preprocessing.

SOLUTION_TIME
indicates the time (in seconds) taken to solve the problem, including preprocessing time.
**Examples: MILP Solver**

This section contains examples that illustrate the options and syntax of the MILP solver in PROC OPTMODEL. Example 14.1 illustrates the use of PROC OPTMODEL to solve an employee scheduling problem. Example 14.2 discusses a multicommodity transshipment problem with fixed charges. Example 14.3 demonstrates how to warm start the MILP solver. Example 14.4 shows the solution of an instance of the traveling salesman problem in PROC OPTMODEL. Other examples of mixed integer linear programs, along with example SAS code, are given in Chapter 7.

**Example 14.1: Scheduling**

The following example has been adapted from the example “A Scheduling Problem” in Chapter 4, “The LP Procedure” (SAS Optimization: Mathematical Optimization Procedures).

Scheduling is a common application area in which mixed integer linear programming techniques are used. In this example, you have eight one-hour time slots in each of five days. You have to assign four employees to these time slots so that each slot is covered every day. You allow the employees to specify preference data for each slot on each day. In addition, the following constraints must be satisfied:

- Each employee has some time slots for which he or she is unavailable (OneEmpPerSlot).
- Each employee must have either time slot 4 or time slot 5 off for lunch (EmpMustHaveLunch).
- Each employee can work at most two time slots in a row (AtMost2ConSlots).
- Each employee can work only a specified number of hours in the week (WeeklyHoursLimit).

To formulate this problem, let $i$ denote a person, $j$ denote a time slot, and $k$ denote a day. Then, let $x_{ijk} = 1$ if person $i$ is assigned to time slot $j$ on day $k$, and 0 otherwise. Let $p_{ijk}$ denote the preference of person $i$ for slot $j$ on day $k$. Let $h_i$ denote the number of hours in a week that person $i$ will work. The formulation of
Example 14.1: Scheduling

This problem follows:

$$\begin{align*}
\text{max} & \quad \sum_{ijk} p_{ijk} x_{ijk} \\
\text{s.t.} & \quad \sum_{i} x_{ijk} = 1 \quad \forall j, k \quad \text{(OneEmpPerSlot)} \\
& \quad x_{i4k} + x_{i5k} \leq 1 \quad \forall i, k \quad \text{(EmpMustHaveLunch)} \\
& \quad x_{i,l,k} + x_{i,l+1,k} + x_{i,l+2,k} \leq 2 \quad \forall i, k, \text{ and } l \leq 6 \quad \text{(AtMost2ConSlots)} \\
& \quad \sum_{jk} x_{ijk} \leq h_i \quad \forall i \quad \text{(WeeklyHoursLimit)} \\
& \quad x_{ijk} = 0 \quad \forall i, j, k \text{ s.t. } p_{ijk} > 0 \\
& \quad x_{ijk} \in \{0, 1\}
\end{align*}$$

The following data set preferences gives the preferences for each individual, time slot, and day. A 10 represents the most desirable time slot, and a 1 represents the least desirable time slot. In addition, a 0 indicates that the time slot is not available. The data set maxhours gives the maximum number of hours each employee can work per week.

data preferences;
  input name $ slot mon tue wed thu fri;
data lines;
  marc 1 10 10 10 10 10
  marc 2 9 9 9 9 9
  marc 3 8 8 8 8 8
  marc 4 1 1 1 1 1
  marc 5 1 1 1 1 1
  marc 6 1 1 1 1 1
  marc 7 1 1 1 1 1
  marc 8 1 1 1 1 1
  mike 1 10 9 8 7 6
  mike 2 10 9 8 7 6
  mike 3 10 9 8 7 6
  mike 4 10 3 3 3 3
  mike 5 1 1 1 1 1
  mike 6 1 2 3 4 5
  mike 7 1 2 3 4 5
  mike 8 1 2 3 4 5
  bill 1 10 10 10 10 10
  bill 2 9 9 9 9 9
  bill 3 8 8 8 8 8
  bill 4 0 0 0 0 0
  bill 5 1 1 1 1 1
  bill 6 1 1 1 1 1
  bill 7 1 1 1 1 1
  bill 8 1 1 1 1 1
  bob 1 10 9 8 7 6
  bob 2 10 9 8 7 6
  bob 3 10 9 8 7 6
  bob 4 10 3 3 3 3
  bob 5 1 1 1 1 1
  bob 6 1 2 3 4 5
data maxhours;
  input name $ hour;
datalines;
marc 20
mike 20
bill 20
bob 20
;
Using PROC OPTMODEL, you can model and solve the scheduling problem as follows:

proc optmodel;
  /* read in the preferences and max hours from the data sets */
  set <string,num> DailyEmployeeSlots;
  set <string> Employees;
  set <num> TimeSlots = (setof {<name,slot> in DailyEmployeeSlots} slot);
  set <string> WeekDays = {"mon","tue","wed","thu","fri"};
  num WeeklyMaxHours{Employees};
  num PreferenceWeights{DailyEmployeeSlots, Weekdays};
  num NSlots = card(TimeSlots);
  read data preferences into DailyEmployeeSlots=[name slot]
  {day in WeekDays} <PreferenceWeights[name,slot,day] = col(day)>;
  read data maxhours into Employees=[name] WeeklyMaxHours=hour;
  /* declare the binary assignment variable x[i,j,k] */
  var Assign{<name,slot> in DailyEmployeeSlots, day in Weekdays} binary;
  /* for each p[i,j,k] = 0, fix x[i,j,k] = 0 */
  for {<name,slot> in DailyEmployeeSlots, day in Weekdays:
    PreferenceWeights[name,slot,day] = 0}
    fix Assign[name,slot,day] = 0;
  /* declare the objective function */
  max TotalPreferenceWeight =
    sum{<name,slot> in DailyEmployeeSlots, day in Weekdays}
    PreferenceWeights[name,slot,day] * Assign[name,slot,day];
  /* declare the constraints */
  con OneEmpPerSlot{slot in TimeSlots, day in Weekdays}:
    sum{name in Employees} Assign[name,slot,day] = 1;
  con EmpMustHaveLunch{name in Employees, day in Weekdays}:
    Assign[name,4,day] + Assign[name,5,day] <= 1;
  con AtMost2ConsSlots{name in Employees, start in 1..NSlots-2, day in Weekdays}:
Example 14.1: Scheduling

Assign[\text{name, start, day}] + Assign[\text{name, start+1, day}] + Assign[\text{name, start+2, day}] \leq 2 ;

\text{con WeeklyHoursLimit(name in Employees)}:
\sum(\text{slot in TimeSlots, day in Weekdays}) Assign[\text{name, slot, day}] \leq \text{WeeklyMaxHours}[\text{name}];

\text{/* solve the model */}
solve with milp;

\text{/* clean up the solution */}
for \{(\text{name, slot}) in \text{DailyEmployeeSlots, day in Weekdays}\}
Assign[\text{name, slot, day}] = \text{round}(Assign[\text{name, slot, day}], 1e-6);

\text{str assigned_employee \{\text{TimeSlots, Weekdays} \} init '';}
\text{for \{\text{slot in TimeSlots, day in Weekdays\} do;}
\text{\text{for \{\text{name in Employees: Assign[\text{name, slot, day}] > 0\} do;}
\text{\text{assigned_employee[\text{slot, day}] = name;}}
\text{\text{leave;}}
\text{\text{end;}}
\text{\text{end;}}
\text{\text{end;}}

\text{create data report from \{\text{slot}=\text{TimeSlots}}
\text{\{day in Weekdays\} <\text{col(day)}=\text{assigned_employee[\text{slot, day}]}>;}
\text{quit;}

The following statements demonstrate how to use the PRINT procedure to display a schedule that shows how the eight time slots are covered for the week:

\text{/* report the solution */}
\text{title 'Reported Solution';}
\text{proc print data=report;}
\text{id slot;}
\text{run;}

The output from the preceding code is displayed in Output 14.1.1.

\textbf{Output 14.1.1} Scheduling Reported Solution

\begin{center}
\begin{tabular}{c c c c c c}
\hline
\text{slot} & \text{mon} & \text{tue} & \text{wed} & \text{thu} & \text{fri} \\
\hline
1 & marc & marc & marc & marc & marc \\
2 & mike & marc & marc & marc & marc \\
3 & mike & mike & bill & bill & \\
4 & bob & mike & mike & mike & mike \\
5 & marc & marc & marc & marc & marc \\
6 & marc & mike & mike & mike & mike \\
7 & mike & mike & mike & mike & \\
8 & marc & bob & bob & bob & bob \\
\hline
\end{tabular}
\end{center}
Example 14.2: Multicommodity Transshipment Problem with Fixed Charges

The following example has been adapted from the example “A Multicommodity Transshipment Problem with Fixed Charges” in Chapter 4, “The LP Procedure” (SAS/OR User’s Guide: Mathematical Programming Legacy Procedures).

This example illustrates the use of PROC OPTMODEL to generate a mixed integer linear program to solve a multicommodity network flow model with fixed charges. Consider a network with nodes $N$, arcs $A$, and a set $C$ of commodities to be shipped between the nodes. The commodities are defined in the data set COMMODITY_DATA, as follows:

```r
title 'Multicommodity Transshipment Problem with Fixed Charges';

data commodity_data;
  do c = 1 to 4;
    output;
  end;
run;
```

Shipping cost $s_{ijc}$ is for each of the four commodities $c$ across each of the arcs $(i, j)$. In addition, there is a fixed charge $f_{ij}$ for the use of each arc $(i, j)$. The shipping costs and fixed charges are defined in the data set ARC_DATA, as follows:

```r
data arc_data;
  input from $ to $ c1 c2 c3 c4 fx;
  datalines;
  farm-a Chicago 20 15 17 22 100
  farm-b Chicago 15 15 15 30 75
  farm-c Chicago 30 30 10 10 100
  farm-a StLouis 30 25 27 22 150
  farm-c StLouis 10 9 11 10 75
  Chicago NY 75 75 75 75 200
  StLouis NY 80 80 80 80 200
;
```

The supply (positive numbers) or demand (negative numbers) $d_{ic}$ at each of the nodes for each commodity $c$ is shown in the data set SUPPLY_DATA, as follows:

```r
data supply_data;
  input node $ sd1 sd2 sd3 sd4;
  datalines;
  farm-a 100 100 40 .
  farm-b 100 200 50 50
  farm-c 40 100 75 100
  NY -150 -200 -50 -75
;
```
Let $x_{ijc}$ define the flow of commodity $c$ across arc $(i, j)$. Let $y_{ij} = 1$ if arc $(i, j)$ is used, and 0 otherwise. Since the total flow on an arc $(i, j)$ must be at most the total demand across all nodes $k \in N$, you can define the trivial upper bound $u_{ijc}$ as

$$x_{ijc} \leq u_{ijc} = \sum_{k \in N \mid d_{kc} < 0} (-d_{kc})$$

This model can be represented using the following mixed integer linear program:

$$\begin{align*}
\text{min} & \quad \sum_{(i,j) \in A} \sum_{c \in C} s_{ijc} x_{ijc} + \sum_{(i,j) \in A} f_{ij} y_{ij} \\
\text{s.t.} & \quad \sum_{j \in N \mid (j,i) \in A} x_{ijc} - \sum_{j \in N \mid (i,j) \in A} x_{jic} \leq d_{ic} \quad \forall i \in N, c \in C \quad \text{(balance_con)} \\
& \quad x_{ijc} \leq u_{ijc} y_{ij} \quad \forall (i, j) \in A, c \in C \quad \text{(fixed_charge_con)} \\
& \quad x_{ijc} \geq 0 \quad \forall (i, j) \in A, c \in C \\
& \quad y_{ij} \in \{0, 1\} \quad \forall (i, j) \in A
\end{align*}$$

Constraint (balance_con) ensures conservation of flow for both supply and demand. Constraint (fixed_charge_con) models the fixed charge cost by forcing $y_{ij} = 1$ if $x_{ijc} > 0$ for some commodity $c \in C$.

The PROC OPTMODEL statements follow:

```plaintext
proc optmodel;
   set COMMODITIES;
      read data commodity_data into COMMODITIES=[c];

   set <str,str> ARCS;
   num unit_cost {ARCS, COMMODITIES};
   num fixed_charge {ARCS};
      read data arc_data into ARCS=[from to] {c in COMMODITIES}
         <unit_cost[from,to,c]=col('c'||c)> fixed_charge=fx;
      print unit_cost fixed_charge;

   set <str> NODES = union {<i,j> in ARCS} {i,j};
   num supply {NODES, COMMODITIES} init 0;
      read data supply_data nomiss into [node] {c in COMMODITIES}
         <supply[node,c]=col('sd'||c)>;
      print supply;

   var AmountShipped {ARCS, c in COMMODITIES} >= 0 <= sum {i in NODES}
      max(supply[i,c],0);

   /* UseArc[i,j] = 1 if arc (i,j) is used, 0 otherwise */
   var UseArc {ARCS} binary;

   /* TotalCost = variable costs + fixed charges */
   min TotalCost = sum {<i,j> in ARCS, c in COMMODITIES}
      unit_cost[i,j,c] * AmountShipped[i,j,c]
```

+ sum {<i,j> in ARCS} fixed_charge[i,j] * UseArc[i,j];

con flow_balance {i in NODES, c in COMMODITIES}:
    sum {<(i),j> in ARCS} AmountShipped[i,j,c] -
    sum {<j,(i)> in ARCS} AmountShipped[j,i,c] <= supply[i,c]

/* if AmountShipped[i,j,c] > 0 then UseArc[i,j] = 1 */
con fixed_charge_def {<i,j> in ARCS, c in COMMODITIES}:
    AmountShipped[i,j,c] <= AmountShipped[i,j,c].ub * UseArc[i,j];

solve;

print AmountShipped;

create data solution from [from to commodity]={<i,j> in ARCS,
    c in COMMODITIES: AmountShipped[i,j,c].sol ne 0} amount=AmountShipped;
quit;

Although the PROC LP example used \( M = 1.0e6 \) in the FIXED_CHARGE_DEF constraint that links the continuous variable to the binary variable, it is numerically preferable to use a smaller, data-dependent value. Here, the upper bound on \( \text{AmountShipped}[i,j,c] \) is used instead. This upper bound is calculated in the first VAR statement as the sum of all positive supplies for commodity \( c \). The logical condition \( \text{AmountShipped}[i,j,k].sol \neq 0 \) in the CREATE DATA statement ensures that only the nonzero parts of the solution appear in the SOLUTION data set.

The problem summary, solution summary, and the output from the three PRINT statements are shown in Output 14.2.1.
Output 14.2.1  Multicommodity Transshipment Problem with Fixed Charges Solution Summary

Multicommodity Transshipment Problem with Fixed Charges

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th></th>
<th>[2]</th>
<th>[3] unit_cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago</td>
<td>NY</td>
<td>1 75</td>
</tr>
<tr>
<td>Chicago</td>
<td>NY</td>
<td>2 75</td>
</tr>
<tr>
<td>Chicago</td>
<td>NY</td>
<td>3 75</td>
</tr>
<tr>
<td>Chicago</td>
<td>NY</td>
<td>4 75</td>
</tr>
<tr>
<td>StLouis</td>
<td>NY</td>
<td>1 80</td>
</tr>
<tr>
<td>StLouis</td>
<td>NY</td>
<td>2 80</td>
</tr>
<tr>
<td>StLouis</td>
<td>NY</td>
<td>3 80</td>
</tr>
<tr>
<td>StLouis</td>
<td>NY</td>
<td>4 80</td>
</tr>
<tr>
<td>farm-a</td>
<td>Chicago</td>
<td>1 20</td>
</tr>
<tr>
<td>farm-a</td>
<td>Chicago</td>
<td>2 15</td>
</tr>
<tr>
<td>farm-a</td>
<td>Chicago</td>
<td>3 17</td>
</tr>
<tr>
<td>farm-a</td>
<td>Chicago</td>
<td>4 22</td>
</tr>
<tr>
<td>farm-a</td>
<td>StLouis</td>
<td>1 30</td>
</tr>
<tr>
<td>farm-a</td>
<td>StLouis</td>
<td>2 25</td>
</tr>
<tr>
<td>farm-a</td>
<td>StLouis</td>
<td>3 27</td>
</tr>
<tr>
<td>farm-a</td>
<td>StLouis</td>
<td>4 22</td>
</tr>
<tr>
<td>farm-b</td>
<td>Chicago</td>
<td>1 15</td>
</tr>
<tr>
<td>farm-b</td>
<td>Chicago</td>
<td>2 15</td>
</tr>
<tr>
<td>farm-b</td>
<td>Chicago</td>
<td>3 15</td>
</tr>
<tr>
<td>farm-b</td>
<td>Chicago</td>
<td>4 30</td>
</tr>
<tr>
<td>farm-c</td>
<td>Chicago</td>
<td>1 30</td>
</tr>
<tr>
<td>farm-c</td>
<td>Chicago</td>
<td>2 30</td>
</tr>
<tr>
<td>farm-c</td>
<td>Chicago</td>
<td>3 10</td>
</tr>
<tr>
<td>farm-c</td>
<td>Chicago</td>
<td>4 10</td>
</tr>
<tr>
<td>farm-c</td>
<td>StLouis</td>
<td>1 10</td>
</tr>
<tr>
<td>farm-c</td>
<td>StLouis</td>
<td>2 9</td>
</tr>
<tr>
<td>farm-c</td>
<td>StLouis</td>
<td>3 11</td>
</tr>
<tr>
<td>farm-c</td>
<td>StLouis</td>
<td>4 10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>[2]</th>
<th>fixed_charge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago</td>
<td>NY</td>
<td>200</td>
</tr>
<tr>
<td>StLouis</td>
<td>NY</td>
<td>200</td>
</tr>
<tr>
<td>farm-a</td>
<td>Chicago</td>
<td>100</td>
</tr>
<tr>
<td>farm-a</td>
<td>StLouis</td>
<td>150</td>
</tr>
<tr>
<td>farm-b</td>
<td>Chicago</td>
<td>75</td>
</tr>
<tr>
<td>farm-c</td>
<td>Chicago</td>
<td>100</td>
</tr>
<tr>
<td>farm-c</td>
<td>StLouis</td>
<td>75</td>
</tr>
</tbody>
</table>
### Output 14.2.1 continued

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>NY</td>
<td>-150</td>
<td>-200</td>
<td>-50</td>
<td>-75</td>
</tr>
<tr>
<td>StLouis</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>farm-a</td>
<td>100</td>
<td>100</td>
<td>40</td>
<td>0</td>
</tr>
<tr>
<td>farm-b</td>
<td>100</td>
<td>200</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>farm-c</td>
<td>40</td>
<td>100</td>
<td>75</td>
<td>100</td>
</tr>
</tbody>
</table>

#### Problem Summary

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
<td>Minimization</td>
</tr>
<tr>
<td>Objective Function</td>
<td>TotalCost</td>
</tr>
<tr>
<td>Objective Type</td>
<td>Linear</td>
</tr>
</tbody>
</table>

- Number of Variables: 35
- Bounded Above: 0
- Bounded Below: 0
- Bounded Below and Above: 35
- Free: 0
- Fixed: 0
- Binary: 7
- Integer: 0

- Number of Constraints: 52
- Linear LE (<=): 52
- Linear EQ (=): 0
- Linear GE (>=): 0
- Linear Range: 0

- Constraint Coefficients: 112
### Output 14.2.1 continued

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>MILP</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Branch and Cut</td>
</tr>
<tr>
<td>Objective Function</td>
<td>TotalCost</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Optimal</td>
</tr>
<tr>
<td>Objective Value</td>
<td>42825</td>
</tr>
</tbody>
</table>

Relative Gap 0
Absolute Gap 0
Primal Infeasibility 0
Bound Infeasibility 0
Integer Infeasibility 0

Best Bound 42825
Nodes 1
Solutions Found 5
Iterations 41
Presolve Time 0.02
Solution Time 0.26
Example 14.3: Facility Location

Consider the classic facility location problem. Given a set $L$ of customer locations and a set $F$ of candidate facility sites, you must decide on which sites to build facilities and assign coverage of customer demand to these sites so as to minimize cost. All customer demand $d_i$ must be satisfied, and each facility has a demand capacity limit $C$. The total cost is the sum of the distances $c_{ij}$ between facility $j$ and its assigned customer $i$, plus a fixed charge $f_j$ for building a facility at site $j$. Let $y_j = 1$ represent choosing site $j$ to build a facility, and 0 otherwise. Also, let $x_{ij} = 1$ represent the assignment of customer $i$ to facility $j$, and 0 otherwise. This model can be formulated as the following integer linear program:
Example 14.3: Facility Location

\[
\begin{align*}
\min & \quad \sum_{i \in L} \sum_{j \in F} c_{ij} x_{ij} + \sum_{j \in F} f_j y_j \\
\text{s.t.} & \quad \sum_{j \in F} x_{ij} = 1 \quad \forall i \in L \quad \text{(assign_def)} \\
& \quad x_{ij} \leq y_j \quad \forall i \in L, j \in F \quad \text{(link)} \\
& \quad \sum_{i \in L} d_i x_{ij} \leq C y_j \quad \forall j \in F \quad \text{(capacity)} \\
& \quad x_{ij} \in \{0, 1\} \quad \forall i \in L, j \in F \\
& \quad y_j \in \{0, 1\} \quad \forall j \in F
\end{align*}
\]

Constraint (assign_def) ensures that each customer is assigned to exactly one site. Constraint (link) forces a facility to be built if any customer has been assigned to that facility. Finally, constraint (capacity) enforces the capacity limit at each site.

Consider also a variation of this same problem where there is no cost for building a facility. This problem is typically easier to solve than the original problem. For this variant, let the objective be

\[
\min \quad \sum_{i \in L} \sum_{j \in F} c_{ij} x_{ij}
\]

First, construct a random instance of this problem by using the following DATA steps:

```
title 'Facility Location Problem';

%let NumCustomers = 50;
%let NumSites = 10;
%let SiteCapacity = 35;
%let MaxDemand = 10;
%let xmax = 200;
%let ymax = 100;
%let seed = 423;

/* generate random customer locations */
data cdata(drop=i);
    call streaminit(&seed);
    length name $8;
    do i = 1 to &NumCustomers;
        name = compress('C'||put(i,best.));
        x = rand('UNIFORM') * &xmax;
        y = rand('UNIFORM') * &ymax;
        demand = rand('UNIFORM') * &MaxDemand;
        output;
    end;
run;

/* generate random site locations and fixed charge */
data sdata(drop=i);
    call streaminit(&seed);
    length name $8;
    do i = 1 to &NumSites;
```
The following PROC OPTMODEL statements first generate and solve the model that contains the no-fixed-charge variant of the cost function. Next, they solve the fixed-charge model and demonstrate the usage of the MAXPOOLSOLS= option for outputting multiple solutions. Note that the solution to the model that has no fixed charge is feasible for the fixed-charge model and should provide a good starting point for the MILP solver. Use the PRIMALIN option to provide an incumbent solution (warm start).
Example 14.3: Facility Location

function='line' drawspace='datavalue' linethickness=1 linecolor='black';
submit;
data csdata;
  set cdata(rename=(y=cy)) sdata(rename=(y=sy));
run;
title1 "Facility Location Problem";
title2 "TotalCost = &varcostNo (Variable = &varcostNo, Fixed = 0)"
proc sgplot data=csdata sganno=CostNoFixedCharge_Data noautolegend;
  scatter x=x y=cy / datalabel=name datalabelattrs=(size=6pt)
    markerattrs=(symbol=circlefilled color=black size=6pt);
  scatter x=x y=sy / datalabel=name datalabelattrs=(size=6pt)
    markerattrs=(symbol=diamond color=blue size=6pt);
  xaxis display=(nolabel);
  yaxis display=(nolabel);
run;
endsubmit;

/* solve the MILP with fixed charges with warm start */
solve obj CostFixedCharge with milp / primalin maxpoolsols=3;
num varcost = sum {i in CUSTOMERS, j in SITES} dist[i,j] * Assign[i,j].sol;
num fixcost = sum {j in SITES} fixed_charge[j] * Build[j].sol;
for {s in 1.._NSOL_} do;
  /* clean up the solution */
  for {i in CUSTOMERS, j in SITES} Assign[i,j] = round(Assign[i,j].sol[s]);
  for {j in SITES} Build[j] = round(Build[j].sol[s]);
  call symput('varcost', put(varcost,6.1));
  call symput('fixcost', put(fixcost,5.1));
  call symput('totalcost', put(CostFixedCharge,6.1));
  /* create a data set for use by PROC SGPLOT */
  create data CostFixedCharge_Data from
    [customer site]{i in CUSTOMERS, j in SITES: Assign[i,j] = 1}
    x1=x[i] y1=y[i] x2=x[j] y2=y[j]
    function='line' drawspace='datavalue' linethickness=1 linecolor='black';
submit s;
  title1 "Facility Location Problem: Solution &s";
  title2 "TotalCost = &totalcost (Variable = &varcost, Fixed = &fixcost)"
  proc sgplot data=csdata sganno=CostFixedCharge_Data noautolegend;
    scatter x=x y=cy / datalabel=name datalabelattrs=(size=6pt)
      markerattrs=(symbol=circlefilled color=black size=6pt);
    scatter x=x y=sy / datalabel=name datalabelattrs=(size=6pt)
      markerattrs=(symbol=diamond color=blue size=6pt);
    xaxis display=(nolabel);
    yaxis display=(nolabel);
run;
endsubmit;
end;
quit;

Output 14.3.1 displays the information that is printed in the log for the facility location problem.
Output 14.3.1  OPTMODEL Log for Facility Location

NOTE: Problem generation will use 4 threads.
NOTE: The problem has 510 variables (0 free, 0 fixed).
NOTE: The problem has 510 binary and 0 integer variables.
NOTE: The problem has 560 linear constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem has 2010 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 10 variables and 500 constraints.
NOTE: The MILP presolver removed 1010 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 500 variables, 60 constraints, and 1000 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1249.9063818</td>
<td>1177.1539196</td>
<td>6.18%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The MILP presolver is applied again.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>1249.9063818</td>
<td>1177.1539196</td>
<td>6.18%</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>1192.6273240</td>
<td>1192.6260561</td>
<td>0.00%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The MILP solver added 12 cuts with 273 cut coefficients at the root.
NOTE: Optimal within relative gap.
NOTE: Objective = 1192.627324.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 510 variables (0 free, 0 fixed).
NOTE: The problem uses 1 implicit variables.
NOTE: The problem has 510 binary and 0 integer variables.
NOTE: The problem has 560 linear constraints (510 LE, 50 EQ, 0 GE, 0 range).
NOTE: The problem has 2010 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 510 variables, 560 constraints, and 2010 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>24086.8916716</td>
<td>0</td>
<td>24087</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>24086.8916716</td>
<td>19197.7909681</td>
<td>25.47%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>24086.8916716</td>
<td>19206.4233772</td>
<td>25.41%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>24086.8916716</td>
<td>19210.6103687</td>
<td>25.38%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>24086.8916716</td>
<td>19217.4194820</td>
<td>25.34%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>7</td>
<td>21664.5736647</td>
<td>19220.9530793</td>
<td>12.71%</td>
<td>0</td>
</tr>
</tbody>
</table>
The output of the program is shown in Output 14.3.2.

Output 14.3.2  Solution Plots for Facility Location

Facility Location Problem
TotalCost = 1192.6 (Variable = 1192.6, Fixed = 0)
Facility Location Problem: Solution 1
TotalCost = 21548 (Variable = 1456.0, Fixed = 20092)
Facility Location Problem: Solution 2
TotalCost = 21550 (Variable = 1457.4, Fixed = 20092)
The economic trade-off for the fixed-charge model forces you to build fewer sites and push more demand to each site.
Example 14.4: Traveling Salesman Problem

The traveling salesman problem (TSP) is that of finding a minimum cost tour in an undirected graph $G$ with vertex set $V = \{1, \ldots, |V|\}$ and edge set $E$. A tour is a connected subgraph for which each vertex has degree two. The goal is then to find a tour of minimum total cost, where the total cost is the sum of the costs of the edges in the tour. Associated with each edge $e \in E$ are a binary variable $x_e$, which indicates whether edge $e$ is part of the tour, and a cost $c_e \in \mathbb{R}$. Let $\delta(S) = \{(i, j) \in E \mid i \in S, j \notin S\}$. Then an integer linear programming (ILP) formulation of the TSP is as follows:

$$\begin{align*}
\min & \quad \sum_{e \in E} c_e x_e \\
\text{s.t.} & \quad \sum_{e \in \delta(i)} x_e = 2 \quad \forall i \in V \quad \text{(two_match)} \\
& \quad \sum_{e \in \delta(S)} x_e \geq 2 \quad \forall S \subset V, 2 \leq |S| \leq |V| - 1 \quad \text{(subtour_elim)} \\
& \quad x_e \in \{0, 1\} \quad \forall e \in E
\end{align*}$$

The equations (two_match) are the matching constraints, which ensure that each vertex has degree two in the subgraph, while the inequalities (subtour_elim) are known as the subtour elimination constraints (SECs) and enforce connectivity.

Since there is an exponential number $O(2^{|V|})$ of SECs, it is impossible to explicitly construct the full TSP formulation for large graphs. An alternative formulation of polynomial size was introduced by Miller, Tucker, and Zemlin (1960) (MTZ):

$$\begin{align*}
\min & \quad \sum_{(i, j) \in E} c_{ij} x_{ij} \\
\text{s.t.} & \quad \sum_{j \in V} x_{ij} = 1 \quad \forall i \in V \quad \text{(assign_i)} \\
& \quad \sum_{i \in V} x_{ij} = 1 \quad \forall j \in V \quad \text{(assign_j)} \\
& \quad u_i - u_j + 1 \leq (|V| - 1)(1 - x_{ij}) \quad \forall (i, j) \in V, i \neq 1, j \neq 1 \quad \text{(mtz)} \\
& \quad 2 \leq u_i \leq |V| \quad \forall i \in \{2, \ldots, |V|\}, \\
& \quad x_{ij} \in \{0, 1\} \quad \forall (i, j) \in E
\end{align*}$$

This formulation uses a directed graph. Constraints (assign_i) and (assign_j) now enforce that each vertex has degree two (one edge in, one edge out). The MTZ constraints (mtz) enforce that no subtours exist.

TSPLIB is a set of benchmark instances for the TSP. For more information about TSPLIB, see https://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/. The following DATA step converts a TSPLIB instance of type EUC_2D into a SAS data set that contains the coordinates of the vertices:

```sas
/* convert the TSPLIB instance into a data set */
data tspData(drop=H);
  infile "st70.tsp";
  input H $1. @;
  if H not in ('N', 'T', 'C', 'D', 'E');
  input @1 var1-var3;
```

The following PROC OPTMODEL statements attempt to solve the TSPLIB instance st70.tsp by using the MTZ formulation:

```plaintext
/* direct solution using the MTZ formulation */
proc optmodel;
set VERTICES;
set EDGES = {i in VERTICES, j in VERTICES: i ne j};
num xc {VERTICES};
num yc {VERTICES};
/* read in the instance and customer coordinates (xc, yc) */
read data tspData into VERTICES=[_n_] xc=var2 yc=var3;
/* the cost is the euclidean distance rounded to the nearest integer */
num c {<i,j> in EDGES}
  init floor( sqrt( ((xc[i]-xc[j])**2 + (yc[i]-yc[j])**2)) + 0.5);
var x {EDGES} binary;
var u {i in 2..card(VERTICES)} >= 2 <= card(VERTICES);
/* each vertex has exactly one in-edge and one out-edge */
con assign_i {i in VERTICES}:
  sum {j in VERTICES: i ne j} x[i,j] = 1;
con assign_j {j in VERTICES}:
  sum {i in VERTICES: i ne j} x[i,j] = 1;
/* minimize the total cost */
min obj
  = sum {<i,j> in EDGES} (if i > j then c[i,j] else c[j,i]) * x[i,j];
/* no subtours */
con mtz {<i,j> in EDGES : (i ne 1) and (j ne 1)}:
  u[i] - u[j] + 1 <= (card(VERTICES) - 1) * (1 - x[i,j]);
solve with milp / maxtime = 600;
quit;
```

It is well known that the MTZ formulation is much weaker than the subtour formulation. The exponential number of SECs makes it impossible, at least in large instances, to use a direct call to the MILP solver with the subtour formulation. For this reason, if you want to solve the TSP with one SOLVE statement, you must use the MTZ formulation and rely strictly on generic cuts and heuristics. Except for very small instances, this is unlikely to be a good approach.

A much more efficient way to tackle the TSP is to dynamically generate the subtour inequalities as cuts. Typically this is done by solving the LP relaxation of the two-matching problem, finding violated subtour cuts, and adding them iteratively. The problem of finding violated cuts is known as the separation problem. In this case, the separation problem takes the form of a minimum cut problem, which is nontrivial to implement efficiently. Therefore, for the sake of illustration, an integer program is solved at each step of the process.

The initial formulation of the TSP is the integral two-matching problem. You solve this by using PROC OPTMODEL to obtain an integral matching, which is not necessarily a tour. In this case, the separation problem is trivial. If the solution is a connected graph, then it is a tour, so the problem is solved. If the solution is a disconnected graph, then each component forms a violated subtour constraint. These constraints are added to the formulation, and the integer program is solved again. This process is repeated until the solution defines a tour.

The following PROC OPTMODEL statements solve the TSP by using the subtour formulation and iteratively adding subtour constraints:
Example 14.4: Traveling Salesman Problem

/* iterative solution using the subtour formulation */
proc optmodel;
  set VERTICES;
  set EDGES = {i in VERTICES, j in VERTICES: i > j};
  num xc {VERTICES};
  num yc {VERTICES};

  num numsubtour init 0;
  set SUBTOUR {1..numsubtour};

  /* read in the instance and customer coordinates (xc, yc) */
  read data tspData into VERTICES=[var1] xc=var2 yc=var3;

  /* the cost is the euclidean distance rounded to the nearest integer */
  num c {<i,j> in EDGES}
      init floor( sqrt( ((xc[i]-xc[j])**2 + (yc[i]-yc[j])**2)) + 0.5);

  var x {EDGES} binary;

  /* minimize the total cost */
  min obj =
    sum {<i,j> in EDGES} c[i,j] * x[i,j];

  /* each vertex has exactly one in-edge and one out-edge */
  con two_match {i in VERTICES}:
    sum {j in VERTICES: i > j} x[i,j]
    + sum {j in VERTICES: i < j} x[j,i] = 2;

  /* no subtours (these constraints are generated dynamically) */
  con subtour_elim {s in 1..numsubtour}:
    sum {<i,j> in EDGES: (i in SUBTOUR[s] and j not in SUBTOUR[s])
      or (i not in SUBTOUR[s] and j in SUBTOUR[s])} x[i,j] >= 2;

  /* this starts the algorithm to find violated subtours */
  set <num,num> EDGES1;
  set VERTICES1 = union{<i,j> in EDGES1} {i, j};
  num component {VERTICES1};
  num numcomp init 2;
  num iter init 1;
  num numiters init 1;
  set ITERS = 1..numiters;
  num sol {ITERS, EDGES};

  /* initial solve with just matching constraints */
  solve;
  call symput(compress('obj'||put(iter,best.)),
    trim(left(put(round(obj),best.))));
  for {<i,j> in EDGES} sol[iter,i,j] = round(x[i,j]);

  /* while the solution is disconnected, continue */
  do while (numcomp > 1);
    iter = iter + 1;
/* find connected components of support graph */
EDGES1 = \{<i,j> in EDGES: \text{round}(x[i,j].sol) = 1\};
solve with network /
   links = (include=EDGES1)
   nodes = (include=VERTICES1)
   concomp
   out = (concomp=component);

numcomp = _oroptmodel_num_\["NUM_COMPONENTS"\];
if numcomp = 1 then leave;
numiters = iter;
umbsubtour = numbsubtour + numcomp;
for \{comp in 1..numcomp\} do;
   SUBTOUR[numbsubtour-numcomp+comp]
   = \{i in VERTICES: component[i] = comp\};
end;
solve;
call symput(compress('obj'||put(iter,best.)),
   trim(left(put(round(obj),best.))));
for \{<i,j> in EDGES\} sol[iter,i,j] = \text{round}(x[i,j]);
end;

/* create a data set for use by PROC SGPLOT */
create data solData from
   \{iter in ITERS, <i,j> in EDGES: sol[iter,i,j] = 1\}
x1=xc[i] y1=yc[i] x2=xc[j] y2=yc[j];
call symput('numiters',put(numiters,best.));
quit;

You can generate plots of the solution and objective value at each stage by using the following statements:

%macro plotTSP;
%do i = 1 %to &numiters;
   /* create annotate data set to draw subtours */
data anno(drop=iter);
   retain drawspace 'datavalue' linethickness 1 function 'line';
   set solData;
   where iter = &i;
run;

title1 h=2 "TSP: Iter = &i, Objective = &&obj&i";
title2;
proc sgplot data=tspData sganno=anno;
   scatter x=var2 y=var3 / datalabel=var1;
   xaxis display=none;
   yaxis display=none;
run;
%end;
%mend plotTSP;

%plotTSP;
The plot in Output 14.4.1 shows the solution and objective value at each stage. Notice that each stage restricts some subset of subtours. When you reach the final stage, you have a valid tour.

**NOTE:** See the “Examples” section in Chapter 15, “The Network Solver,” for an example of how to use a specialized algorithm to solve the TSP.

**Output 14.4.1** Traveling Salesman Problem Iterative Solution

**TSP: Iter = 1, Objective = 625**

**TSP: Iter = 2, Objective = 652**
Output 14.4.1 continued

TSP: Iter = 3, Objective = 673

TSP: Iter = 4, Objective = 674

TSP: Iter = 5, Objective = 675
References


Chapter 15
The Network Solver

Contents

Overview: Network Solver ................................................................. 660
Getting Started: Network Solver ....................................................... 660
Syntax: Network Solver ................................................................. 665
  Functional Summary ................................................................. 665
  SOLVE WITH NETWORK Statement ............................................. 669
    General Options ................................................................. 670
    Input and Output Options ..................................................... 671
    Algorithm Options ............................................................. 675
Details: Network Solver ................................................................. 680
  Input Data for the Network Solver ............................................. 680
  Solving over Subsets of Nodes and Links (Filters) ......................... 683
  Numeric Limitations ................................................................ 688
  Biconnected Components and Articulation Points ......................... 689
  Clique ....................................................................................... 692
  Connected Components ............................................................ 695
  Cycle ......................................................................................... 699
  Linear Assignment (Matching) .................................................... 705
  Minimum-Cost Network Flow ..................................................... 706
  Minimum Cut ............................................................................. 712
  Minimum Spanning Tree ........................................................... 716
  Shortest Path ............................................................................ 718
  Transitive Closure .................................................................... 731
  Traveling Salesman Problem ...................................................... 734
  Macro Variable _OROPTMODEL_ ................................................. 740
Examples: Network Solver ............................................................... 744
  Example 15.1: Articulation Points in a Terrorist Network .............. 744
  Example 15.2: Cycle Detection for Kidney Donor Exchange .......... 746
  Example 15.3: Linear Assignment Problem for Minimizing Relay Times 751
  Example 15.4: Linear Assignment Problem, Sparse Format versus Dense Format 753
  Example 15.5: Minimum Spanning Tree for Computer Network Topology 757
  Example 15.6: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System 759
  Example 15.7: Traveling Salesman Tour of US Capital Cities .......... 762
References ...................................................................................... 767
Overview: Network Solver

The network solver includes a number of graph theory, combinatorial optimization, and network analysis algorithms. The algorithm classes are listed in Table 15.1.

### Table 15.1  Algorithm Classes in the Network solver

<table>
<thead>
<tr>
<th>Algorithm Class</th>
<th>SOLVE WITH NETWORK Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biconnected components</td>
<td>BICONNECTEDCOMPONENTS</td>
</tr>
<tr>
<td>Maximal cliques</td>
<td>CLIQUE=</td>
</tr>
<tr>
<td>Connected components</td>
<td>CONNECTEDCOMPONENTS=</td>
</tr>
<tr>
<td>Cycle detection</td>
<td>CYCLE=</td>
</tr>
<tr>
<td>Linear assignment (matching)</td>
<td>LINEARASSIGNMENT</td>
</tr>
<tr>
<td>Minimum-cost network flow</td>
<td>MINCOSTFLOW</td>
</tr>
<tr>
<td>Minimum cut</td>
<td>MINCUT=</td>
</tr>
<tr>
<td>Minimum spanning tree</td>
<td>MINSPEANTRIE</td>
</tr>
<tr>
<td>Shortest path</td>
<td>SHORTESTPATH=</td>
</tr>
<tr>
<td>Transitive closure</td>
<td>TRANSITIVECLOSURE</td>
</tr>
<tr>
<td>Traveling salesman</td>
<td>TSP=</td>
</tr>
</tbody>
</table>

You can use the network solver to analyze relationships between entities. These relationships are typically defined by using a graph. A graph, $G = (N, A)$, is defined over a set $N$ of nodes, and a set $A$ of links. A node is an abstract representation of some entity (or object), and an arc defines some relationship (or connection) between two nodes. The terms node and vertex are often interchanged in describing an entity. The term arc is often interchanged with the term edge or link in describing a relationship.

Unlike other solvers that PROC OPTMODEL uses, the network solver operates directly on arrays and sets. You do not need to explicitly define variables, constraints, and objectives to use the network solver. PROC OPTMODEL declares the appropriate objects internally as needed. You specify the names of arrays and sets that define your inputs and outputs as options in the SOLVE WITH NETWORK statement.

Getting Started: Network Solver

This section shows an introductory example for getting started with the network solver. For more information about the expected input formats and the various algorithms available, see the sections “Details: Network Solver” on page 680 and “Examples: Network Solver” on page 744.

Consider the following road network between a SAS employee’s home in Raleigh, NC, and the SAS headquarters in Cary, NC.

In this road network (graph), the links are the roads and the nodes are intersections between roads. For each road, you assign a link attribute in the parameter time_to_travel to describe the number of minutes that it takes to drive from one node to another. The following data were collected using Google Maps (Google 2011):
Using the network solver, you want to find the route that yields the shortest path between home (614 Capital Blvd) and the SAS headquarters (SAS Campus Drive). This can be done by using the SHORTESTPATH= option as follows:

```plaintext
data LinkSetInRoadNC10am;
    input start_inter $1-20 end_inter $21-40 miles miles_per_hour;
datalines;
614CapitalBlvd Capital/WadeAve 0.6 25
614CapitalBlvd Capital/US70W 0.6 25
614CapitalBlvd Capital/US440W 3.0 45
Capital/WadeAve WadeAve/RaleighExpy 3.0 40
Capital/US70W US70W/US440W 3.2 60
US70W/US440W US440W/RaleighExpy 2.7 60
Capital/US440W US440W/RaleighExpy 6.7 60
US440W/RaleighExpy RaleighExpy/US40W 3.0 60
WadeAve/RaleighExpy RaleighExpy/US40W 3.0 60
RaleighExpy/US40W US40W/HarrisonAve 1.3 55
US40W/HarrisonAve SASCampusDrive 0.5 25;
```

For more information about shortest path algorithms in the network solver, see the section “Shortest Path” on page 718. Figure 15.1 displays the output data set ShortPath, which shows the best route to take to minimize travel time at 10:00 a.m. This route is also shown in Google Maps in Figure 15.2.
Now suppose that it is rush hour (5:00 p.m.) and the time to traverse the roads has changed because of traffic patterns. You want to find the route that is the shortest path for going home from SAS headquarters under different speed assumptions due to traffic.

The following statements are similar to the first network solver run, except that one `miles_per_hour` value is modified and the `SOURCE=` and `SINK=` option values are reversed:
proc optmodel;
   set<str,str> LINKS;
   num miles(LINKS);
   num miles_per_hour(LINKS);
   num time_to_travel{(i,j) in LINKS} = miles[i,j] / miles_per_hour[i,j] * 60;
   read data LinkSetInRoadNC10am into
      LINKS=[start_inter end_inter]
      miles miles_per_hour
   ;
   /* high traffic */
   miles_per_hour['Capital/WadeAve','WadeAve/RaleighExpy'] = 25;
   /* You can compute paths between many pairs of source and destination, so these parameters are declared as sets */
   set HOME = /"614CapitalBlvd"/;
   set WORK = /"SASCampusDrive"/;
   /* The path is stored as a set of: Start, End, Sequence, Tail, Head */
   set<str,str,num,str,str> PATH;

   solve with network /
      links = ( weight = time_to_travel )
      shortpath = ( source = WORK
                  sink = HOME )
      out = ( sppaths = PATH )
   ;
   create data ShortPath from [s t order start_inter end_inter]=PATH
      time_to_travel[start_inter,end_inter];
quit;

Now, the output data set ShortPath, shown in Figure 15.3, shows the best route for going home at 5:00 p.m. Because the traffic on Wade Avenue is usually heavy at this time of day, the route home is different from the route to work.

**Figure 15.3** Shortest Path for Road Network at 5:00 P.M.

<table>
<thead>
<tr>
<th>order</th>
<th>start_inter</th>
<th>end_inter</th>
<th>time_to_travel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>US40W/HarrisonAve</td>
<td>SASCampusDrive</td>
<td>1.2000</td>
</tr>
<tr>
<td>2</td>
<td>RaleighExpy/US40W</td>
<td>US40W/HarrisonAve</td>
<td>1.4182</td>
</tr>
<tr>
<td>3</td>
<td>US440W/RaleighExpy</td>
<td>RaleighExpy/US40W</td>
<td>3.0000</td>
</tr>
<tr>
<td>4</td>
<td>US70W/US440W</td>
<td>US440W/RaleighExpy</td>
<td>2.7000</td>
</tr>
<tr>
<td>6</td>
<td>614CapitalBlvd</td>
<td>Capital/US70W</td>
<td>1.4400</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>12.9582</strong></td>
</tr>
</tbody>
</table>
This new route is shown in Google Maps in Figure 15.4.

**Figure 15.4** Shortest Path for Road Network at 5:00 P.M. in Google Maps
Syntax: Network Solver

SOLVE WITH NETWORK /

General and Diagnostic Options:
<GRAPH_DIRECTION=DIRECTED | UNDIRECTED> 
<INCLUDE_SELFLINK>
<LOGFREQ=number>
<LOGLEVEL=number | string>
<MAXTIME=number>
<NTHREADS=number>
<TIMETYPE=number | string>

Data Input and Output Options:
<LINKS=( suboptions )>
<NODES=( suboptions )>
<OUT=( suboptions )>
<SUBGRAPH=( suboptions )>

Algorithm Options:
<BICONNECTEDCOMPONENTS=( suboptions )> 
<CLIQUE=( suboption )>
<CONNECTEDCOMPONENTS=( suboption )>
<CYCLE=( suboptions )>
<LINEARASSIGNMENT=( suboptions )>
<MINCOSTFLOW=( suboptions )>
<MINCUT=( suboptions )>
<MINSPANTREE=( suboptions )>
<SHORTESTPATH=( suboptions )>
<TRANSITIVECLOSURE=( suboptions )>
<TSP=( suboptions )> ;

There are three types of SOLVE WITH NETWORK statement options:

- General and diagnostic options have the same meaning for multiple algorithms.
- Data input and output options, such as the LINKS=, NODES=, and OUT= options, control the names of the sets and variables that the network solver uses to build the graph and that the algorithms use for output.
- Algorithm options select an algorithm to run, and where available, provide further algorithm-specific configuration directives.

The section “Functional Summary” on page 665 provides a quick reference for each of the suboptions for each option. Each option is then described in more detail in its own section, in alphabetical order.

Functional Summary

Table 15.2 summarizes the options and suboptions available in the SOLVE WITH NETWORK statement.
### Table 15.2 Functional Summary of SOLVE WITH NETWORK Options

<table>
<thead>
<tr>
<th>Description</th>
<th>Option Suboption</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies directed or undirected graphs</td>
<td>GRAPH_DIRECTION=</td>
</tr>
<tr>
<td>Includes self links in the graph definition</td>
<td>INCLUDE_SELFLINK=</td>
</tr>
<tr>
<td>Specifies the iteration log frequency</td>
<td>LOGFREQ=</td>
</tr>
<tr>
<td>Controls the amount of information that is displayed in the SAS log</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the maximum time spent calculating results</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Specifies the number for threads to use for threaded processing</td>
<td>NTHREADS=</td>
</tr>
<tr>
<td>Specifies whether time units are in CPU time or real time</td>
<td>TIMETYPE=</td>
</tr>
<tr>
<td><strong>Input and Output Options</strong></td>
<td></td>
</tr>
<tr>
<td>Groups link-indexed data</td>
<td></td>
</tr>
<tr>
<td>Names a set of links to include in the graph definition even if no weights or bounds are available for them</td>
<td>INCLUDE=</td>
</tr>
<tr>
<td>Specifies the flow lower bound for each link</td>
<td>LOWER=</td>
</tr>
<tr>
<td>Specifies the flow upper bound for each link</td>
<td>UPPER=</td>
</tr>
<tr>
<td>Specifies link weights</td>
<td>WEIGHT=</td>
</tr>
<tr>
<td>Groups node-indexed data</td>
<td></td>
</tr>
<tr>
<td>Names a set of nodes to include in the graph definition even if no weights are available for them</td>
<td>INCLUDE=</td>
</tr>
<tr>
<td>Specifies node supply lower bounds in the minimum-cost network flow problem</td>
<td>LOWER=</td>
</tr>
<tr>
<td>Specifies node supply upper bounds in the minimum-cost network flow problem</td>
<td>UPPER=</td>
</tr>
<tr>
<td>Specifies node weights</td>
<td>WEIGHT=</td>
</tr>
<tr>
<td>Specifies the input sets that enable you to solve a problem over a subgraph</td>
<td>SUBGRAPH=()</td>
</tr>
<tr>
<td>Specifies the subset of links to use</td>
<td>LINKS=</td>
</tr>
<tr>
<td>Specifies the subset of nodes to use</td>
<td>NODES=</td>
</tr>
<tr>
<td>Specifies the output sets or arrays for each algorithm (see Table 15.4 for which OUT= suboptions you can specify for each algorithm)</td>
<td>OUT=()</td>
</tr>
<tr>
<td>Specifies the output set for articulation points</td>
<td>ARTPPOINTS=</td>
</tr>
<tr>
<td>Specifies the output set for linear assignment</td>
<td>ASSIGNMENTS=</td>
</tr>
<tr>
<td>Specifies the array to contain the biconnected component of each link</td>
<td>BICONCOMP=</td>
</tr>
<tr>
<td>Specifies the output set for cliques</td>
<td>CLIQUES=</td>
</tr>
<tr>
<td>Specifies the set to contain the pairs ${u, v}$ of nodes where $v$ is reachable from $u$</td>
<td>CLOSURE=</td>
</tr>
<tr>
<td>Specifies the output array for connected components</td>
<td>CONCOMP=</td>
</tr>
<tr>
<td>Specifies the output set for the cut-sets for minimum cuts</td>
<td>CUTSETS=</td>
</tr>
<tr>
<td>Specifies the output set for cycles</td>
<td>CYCLES=</td>
</tr>
<tr>
<td>Specifies the output array for the flow on each link</td>
<td>FLOW=</td>
</tr>
<tr>
<td>Specifies the output set for the minimum spanning tree (forest)</td>
<td>FOREST=</td>
</tr>
<tr>
<td>Description</td>
<td>Option Suboption</td>
</tr>
<tr>
<td>-------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Specifies the output set for the links that remain after the SUBGRAPH= option is applied</td>
<td>LINKS=</td>
</tr>
<tr>
<td>Specifies the output set for the nodes that remain after the SUBGRAPH= option is applied</td>
<td>NODES=</td>
</tr>
<tr>
<td>Specifies the output array for the node order in the traveling salesman problem</td>
<td>ORDER=</td>
</tr>
<tr>
<td>Specifies the output set for the partitions for minimum cuts</td>
<td>PARTITIONS=</td>
</tr>
<tr>
<td>Specifies the set to contain the link sequence for each path</td>
<td>SPPATHS=</td>
</tr>
<tr>
<td>Specifies the numeric array to contain the path weight for each source and sink node pair</td>
<td>SPWEIGHTS=</td>
</tr>
<tr>
<td>Specifies the output set for the tour in the traveling salesman problem</td>
<td>TOUR=</td>
</tr>
</tbody>
</table>

### Algorithm Options and Suboptions

- **BICONNECTEDCOMPONENTS**
  - Finds biconnected components and articulation points of an undirected input graph
  - Finds maximal cliques in the input graph
    - Specifies the maximum number of cliques to return | MAXCLIQUES= |
  - Finds the connected components of the input graph
    - Specifies the algorithm to use for calculating connected components | ALGORITHM= |
  - Finds the cycles (or the existence of a cycle) in the input graph
    - Specifies the algorithm to use in enumerating cycles | ALGORITHM= |
    - Specifies the maximum number of cycles to return | MAXCYCLES= |
    - Specifies the maximum link count for the cycles to return | MAXLENGTH= |
    - Specifies the maximum link weight for the cycles to return | MAXLINKWEIGHT= |
    - Specifies the maximum sum of node weights to allow in a cycle | MAXNODEWEIGHT= |
    - Specifies the minimum link count for the cycles to return | MINLENGTH= |
    - Specifies the minimum link weight for the cycles to return | MINLINKWEIGHT= |
    - Specifies the minimum node weight for the cycles to return | MINNODEWEIGHT= |
  - Solves the minimal-cost linear assignment problem | LINEARASSIGNMENT |
  - Solves the minimum-cost network flow problem | MINCOSTFLOW |
  - Finds the minimum link-weighted cut of an input graph
    - Specifies the maximum number of cuts to return from the algorithm | MAXCUTS= |
    - Specifies the maximum weight of each cut to return from the algorithm | MAXWEIGHT= |
  - Solves the minimum link-weighted spanning tree problem on an input graph | MINSPANTREE |
  - Calculates shortest paths between sets of nodes on the input graph
    - Specifies the maximum path weight | MAXSPATHWEIGHT= |
    - Specifies the set of sink nodes | SINK= |
    - Specifies the set of source nodes | SOURCE= |
  - Calculates the transitive closure of an input graph | TRANSITIVECLOSURE |
  - Solves the traveling salesman problem | TSP= |
Table 15.2  (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Option Suboption</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requests that the stopping criterion be based on the absolute objective gap</td>
<td>ABSOBJGAP=</td>
</tr>
<tr>
<td>Specifies the cutoff value for branch-and-bound node removal</td>
<td>CUTOFF=</td>
</tr>
<tr>
<td>Specifies the level of cutting planes to be generated by the network solver</td>
<td>CUTSTRATEGY=</td>
</tr>
<tr>
<td>Specifies the initial and primal heuristics level</td>
<td>HEURISTICS=</td>
</tr>
<tr>
<td>Specifies the maximum number of branch-and-bound nodes to be processed</td>
<td>MAXNODES=</td>
</tr>
<tr>
<td>Specifies the maximum number of feasible tours to be identified</td>
<td>MAXSOLS=</td>
</tr>
<tr>
<td>Specifies whether to use a mixed integer linear programming solver</td>
<td>MILP=</td>
</tr>
<tr>
<td>Requests that the stopping criterion be based on relative objective gap</td>
<td>RELOBJGAP=</td>
</tr>
<tr>
<td>Requests that the stopping criterion be based on the target objective value</td>
<td>TARGET=</td>
</tr>
</tbody>
</table>

Table 15.3 lists the valid GRAPH_DIRECTION= values for each algorithm option in the SOLVE WITH NETWORK statement.

Table 15.3  Supported Graph Directions by Algorithm

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Undirected</th>
<th>Directed</th>
</tr>
</thead>
<tbody>
<tr>
<td>BICONNECTEDCOMPONENTS</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>CLIQUE</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>CONNECTEDCOMPONENTS</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>CYCLE</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>LINEARASSIGNMENT</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>MINCOSTFLOW</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>MINCUT</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>MINSPANTREE</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>SHORTESTPATH</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>TRANSITIVECLOSURE</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>TSP</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

Table 15.4 indicates, for each algorithm option in the SOLVE WITH NETWORK statement, which output options you can specify, and what their types can be. The types vary depending on whether nodes are of type STRING or NUMBER.
The SOLVE WITH NETWORK statement invokes the network solver. You can specify the following `options` to define various processing and diagnostic controls, the graph input and output, and the algorithm to run:
General Options

You can specify the following general options, which have the same meaning for multiple algorithms.

**GRAPH_DIRECTIONS=DIRECTED | UNDIRECTED**

**DIRECTION=DIRECTED | UNDIRECTED**

specifies directed or undirected graphs.

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIRECTED</td>
<td>Requests a directed graph. In a directed graph, each link ((i, j)) has a direction that defines how something (for example, information) might flow over that link. In link ((i, j)), information flows from node (i) to node (j) ((i \rightarrow j)). The node (i) is called the source (tail) node, and node (j) is called the sink (head) node.</td>
</tr>
<tr>
<td>UNDIRECTED</td>
<td>Requests an undirected graph. In an undirected graph, each link ({i, j}) has no direction and information can flow in either direction. That is, ({i, j} = {j, i}).</td>
</tr>
</tbody>
</table>

By default, **GRAPH_DIRECTIONS=UNDIRECTED**.

**INCLUDE_SELFLINK**

includes self links in the graph definition—for example, \((i, i)\)—when an input graph is read. By default, when the network solver reads the **LINKS=** data, it removes all self links.

**LOGFREQ=**number

controls the frequency with which an algorithm reports progress from its underlying solver. This setting is recognized by the traveling salesman problem and minimum-cost flow algorithms. You can set number to 0 to turn off log updates from underlying algorithms.

**LOGLEVEL=**number | string

controls the amount of information that is displayed in the SAS log. This setting sets the log level for all algorithms. Table 15.6 describes the valid values for this option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all procedure-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the input, output, and algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the input, output, and algorithmic processing</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the input, output, and algorithmic processing</td>
</tr>
</tbody>
</table>
By default, LOGLEVEL=BASIC.

**MAXTIME=number**

specifies the maximum time spent calculating results. The type of time (either CPU time or real time) is determined by the value of the TIMETYPE= option. The value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment. The clique, cycle, minimum-cost network flow, and traveling salesman problem algorithms recognize the MAXTIME= option.

**NTHREADS=number**

specifies the number for threads to use for threaded processing. The SHORTESTPATH and CYCLE ALGORITHM=BUILD algorithms can use threaded processing. The value of *number* must be an integer between 1 and 256, inclusive. The default is the value of the OPTMODEL NTHREADS= option.

**TIMETYPE=number | string**

specifies whether CPU time or real time is used for measuring solution times. This affects the MAXTIME= option for each applicable algorithm. Table 15.7 describes the valid values of the TIMETYPE= option.

<table>
<thead>
<tr>
<th><em>number</em></th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CPU</td>
<td>Specifies units of CPU time</td>
</tr>
<tr>
<td>1</td>
<td>REAL</td>
<td>Specifies units of real time</td>
</tr>
</tbody>
</table>

By default, TIMETYPE=REAL.

**Input and Output Options**

The following options enable you to specify the graph to run algorithms on. These options take array and set names. They are known as *identifier expressions* in Chapter 10, “The OPTMODEL Procedure.” Also see Table 15.4 for semantic requirements and the section “Input Data for the Network Solver” on page 680 for use cases.

**LINKS=( suboptions )**

groups link-indexed data. For more information, see the section “Input Data for the Network Solver” on page 680.

You can specify the following *suboptions*:

**INCLUDE=set-name**

names a set of links to include in the graph definition even if no weights or bounds are available for them. For more information, see “Example 15.1: Articulation Points in a Terrorist Network” on page 744. The set must be numeric, and it must be indexed over a subset of the links of the graph.
LOWER=\textit{array-name}  
specifies the flow lower bound for each link. The array must be numeric, and it must be indexed over a subset of the inks of the graph.

UPPER=\textit{array-name}  
specifies the flow upper bound for each link. The array must be numeric, and it must be indexed over a subset of the links of the graph.

WEIGHT=\textit{array-name}  
specifies link weights. The array must be numeric, and it must be indexed over a subset of the links of the graph. If you specify this suboption, then any link that does not appear in the index set of the WEIGHT= array has weight 0. If you do not specify this suboption, then every link has weight 1.

\textbf{NODES=} (\textit{suboptions})  
groups node-indexed data. For more information, see the section “Input Data for the Network Solver” on page 680.

You can specify the following \textit{suboptions}:

\textbf{INCLUDE=} \textit{set-name}  
names a set of nodes to include in the graph definition even if no weights are available for them. For more information, see the section “Connected Components” on page 695.

\textbf{LOWER=} \textit{array-name}  
specifies node supply lower bounds in the minimum-cost network flow problem. The array must be numeric, and it must be indexed over a subset of the nodes of the graph. For more information, see the section “Minimum-Cost Network Flow” on page 706.

\textbf{UPPER=} \textit{array-name}  
specifies node supply upper bounds in the minimum-cost network flow problem. The array must be numeric, and it must be indexed over a subset of the nodes of the graph. For more information, see the section “Minimum-Cost Network Flow” on page 706.

\textbf{WEIGHT=} \textit{array-name}  
specifies node weights. The array must be numeric, and it must be indexed over a subset of the nodes of the graph. For more information, see the section “Cycle” on page 699.

\textbf{OUT=} (\textit{suboptions})  
specifies the output sets or arrays for each algorithm (see Table 15.4 for which OUT= suboptions you can specify for each algorithm). You can use some of these options (even if you do not invoke any algorithm) to see the filtering outcome that is produced by the SUBGRAPH= option.

If you do not specify a \textit{suboption} that matches the \textit{algorithm option} in the statement, the algorithm runs and only updates the objective.

If you specify a suboption that does not match the \textit{algorithm option} in the statement, OPTMODEL issues a warning.

When you declare arrays that are indexed over nodes, over links, or over sets of nodes or links, you must use the same type you used in your node definition.

See the various algorithm sections for examples of the use of these OUT= \textit{suboptions}. 


**ARTPONTS**=`set-name`

specifies the output set for articulation points. Each element of the set represents a node ID. This suboption is used with the BICONNECTEDCOMPONENTS algorithm option.

**ASSIGNMENTS**=`set-name`

specifies the output set for linear assignment. This suboption is used with the LINEARASSIGNMENT algorithm option.

**BICONCOMP**=`array-name`

specifies the array to contain the biconnected component of each link. This suboption is used with the BICONNECTEDCOMPONENTS algorithm option.

**CLIQUEs**=`set-name`

specifies the output set for cliques. Each tuple of the set represents clique ID and node ID. This suboption is used with the CLIQUE= algorithm option.

**CLOSURE**=`set-name`

specifies the set to contain the pairs \((u, v)\) of nodes where \(v\) is reachable from \(u\). This suboption is used with the TRANSITIVECLOSURE algorithm option.

**CONCOMP**=`array-name`

specifies the output array for connected components. This suboption is used with the CONNECTEDCOMPONENTS= algorithm option.

**CUTSETS**=`set-name`

specifies the output set for the cut-sets for minimum cuts. Each tuple of the set represents the cut ID, the tail node ID, and the head node ID. This suboption is used with the MINCUT= algorithm option.

**CYCLES**=`set-name`

specifies the output set for cycles. Each tuple of the set represents a cycle ID, the order within that cycle, and the node ID. This suboption is used with the CYCLE= algorithm option.

**FLOW**=`array-name`

specifies the output array for the flow on each link. This suboption is used with the MINCOSTFLOW algorithm option.

**FOREST**=`set-name`

specifies the output set for the minimum spanning tree (forest). This suboption is used with the MINSPANTREE algorithm option.

**LINKS**=`set-name`

specifies the output set for the links that remain after the SUBGRAPH= option is applied. Each tuple of the set represents tail and head nodes, followed by a sequence of numbers that correspond to the attributes that you provide in the LINKS= suboptions. The length of the tuples must be the number of attributes that you specify plus 2 (for the tail and head node information). The options that you specify in the LINKS= option will appear in the output in the following order: WEIGHT=, LOWER=, and UPPER=. For an example, see Figure 15.12 in “Solving over Subsets of Nodes and Links (Filters)” on page 683.
Chapter 15: The Network Solver

**NODES=** *set-name*

specifies the output set for the nodes that remain after the **SUBGRAPH=** option is applied. Each tuple of the set represents a node, followed by a sequence of numbers that correspond to the attributes that you provide in the **NODES=** suboptions. The length of the tuples must be the number of attributes that you specify plus 1 (for node information). The options that you specify in the **NODES=** option will appear in the output in the following order: **WEIGHT=**, **LOWER=**, and **UPPER=**. For an example, see the section “Minimum-Cost Network Flow with Flexible Supply and Demand” on page 710.

**ORDER=** *array-name*

specifies the numeric array to contain the position of each node within the optimal tour. This suboption is used with the TSP= algorithm option.

**PARTITIONS=** *set-name*

specifies the output set for the partitions for minimum cuts. The set contains, for each partition, the node IDs in the smaller of the two subsets. Each tuple of the set represents a cut ID and a node ID. This suboption is used with the MINCUT= algorithm option.

**SPPATHS=** *set-name*

specifies the set to contain the link sequence for each path. Each tuple of the set represents a source node ID, a sink node ID, a sequence number, a tail node ID, and a head node ID. This suboption is used with the SHORTESTPATH= algorithm option.

**SPWEIGHTS=** *array-name*

specifies the numeric array to contain the path weight for each source and sink node pair. This suboption is used with the SHORTESTPATH= algorithm option.

**TOUR=** *set-name*

specifies the output set for the tour in the traveling salesman problem. This suboption is used with the TSP= algorithm option.

**SUBGRAPH=**( *suboptions*)

specifies the input sets that enable you to solve a problem over a subgraph. For more information, see the section “Input Data for the Network Solver” on page 680.

You can specify the following **suboptions**:

**LINKS=** *set-name*

specifies the subset of links to use. If you specify a node pair that is not referenced in any of the suboptions of the **LINKS=** option, then the network solver returns an error.

**NODES=** *set-name*

specifies the subset of nodes to use. If you specify a node that is not referenced in any of the suboptions of the **LINKS=** option or the **NODES=** option, then the network solver returns an error.
Algorithm Options

**BICONNECTEDCOMPONENTS**

finds biconnected components and articulation points of an undirected input graph. For more information, see the section “Biconnected Components and Articulation Points” on page 689.

**CLIQUE**

finds maximal cliques in the input graph. For more information, see the section “Clique” on page 692.

You can specify the following *suboption*:

**MAXCLIQUE**=`number` | `ALL`

specifies the maximum number of cliques to return. You can specify `number` as an integer in the range 1 to $2^{31} - 1$, inclusive. `ALL` represents the maximum value, $2^{31} - 1$. The default is 1.

**CONNECTEDCOMPONENTS**

finds the connected components of the input graph. For more information, see the section “Connected Components” on page 695.

You can specify the following *suboption*:

**ALGORITHM**=`AUTOMATIC` | `DFS` | `UNIONFIND`

specifies the algorithm to use for calculating connected components. Table 15.8 describes the valid values for this option.

### Table 15.8 Values for the ALGORITHM= Option

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOMATIC</td>
<td>Automatically determines the algorithm for connected components</td>
</tr>
<tr>
<td>DFS</td>
<td>Uses the depth-first search algorithm for connected components</td>
</tr>
<tr>
<td>UNIONFIND</td>
<td>Uses the union-find algorithm for connected components. You can use ALGORITHM=UNIONFIND only with undirected graphs.</td>
</tr>
</tbody>
</table>

By default, ALGORITHM=UNIONFIND for undirected graphs, and ALGORITHM=DFS for directed graphs.

**CYCLE**

finds the cycles (or the existence of a cycle) in the input graph. For more information, see the section “Cycle” on page 699.

You can specify the following *suboptions* in the CYCLE= option:
**ALGORITHM=BACKTRACK | BUILD**

specifies the algorithm to use in enumerating cycles. Table 15.9 lists the valid values for this option.

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BACKTRACK</td>
<td>Uses a backtracking algorithm based on Johnson (1975)</td>
</tr>
</tbody>
</table>

By default, ALGORITHM=BACKTRACK for MAXLENGTH= values greater than 20; otherwise, ALGORITHM=BUILD.

**MAXCYCLES=number | ALL**

specifies the maximum number of cycles to return. You can specify *number* as an integer in the range 1 to $2^{31} - 1$, inclusive. ALL represents the maximum value, $2^{31} - 1$. The default is 1.

**MAXLENGTH=number**

specifies the maximum number of links to allow in a cycle. Any cycle whose length is greater than *number* is removed from the results. You can specify *number* as an integer in the range 1 to $2^{31} - 1$, inclusive. The default is the maximum value, $2^{31} - 1$. By default, nothing is removed from the results.

**MAXLINKWEIGHT=number**

specifies the maximum sum of link weights to allow in a cycle. Any cycle whose sum of link weights is greater than *number* is removed from the results. The default is the positive number that has the largest absolute value that can be represented in your operating environment. By default, nothing is filtered.

**MAXNODEWEIGHT=number**

specifies the maximum sum of node weights to allow in a cycle. Any cycle whose sum of node weights is greater than *number* is removed from the results. The default is the positive number that has the largest absolute value that can be represented in your operating environment. By default, nothing is filtered.

**MINLENGTH=number**

specifies the minimum number of links to allow in a cycle. Any cycle that has fewer links than *number* is removed from the results. You can specify *number* as an integer in the range 1 to $2^{31} - 1$, inclusive. The default is 1. By default, only self-loops are filtered.

**MINLINKWEIGHT=number**

specifies the minimum sum of link weights to allow in a cycle. Any cycle whose sum of link weights is less than *number* is removed from the results. The default the negative number that has the largest absolute value that can be represented in your operating environment. By default, nothing is filtered.

**MINNODEWEIGHT=number**

specifies the minimum sum of node weights to allow in a cycle. Any cycle whose sum of node weights is less than *number* is removed from the results. The default is the negative number that has the largest absolute value that can be represented in your operating environment. By default, nothing is filtered.
LINEARASSIGNMENT< () >

LAP< () >
solves the minimal-cost linear assignment problem. In graph terms, this problem is also known as the minimum link-weighted matching problem on a bipartite directed graph. The input data (the cost matrix) are defined as a directed graph by specifying the LINKS= option in the SOLVE WITH NETWORK statement, where the costs are defined as link weights. Internally, the graph is treated as a bipartite directed graph.

For more information, see the section “Linear Assignment (Matching)” on page 705.

MINCOSTFLOW< () >

MCF< () >
solves the minimum-cost network flow problem.

For more information, see the section “Minimum-Cost Network Flow” on page 706.

MINCUT< ( suboptions ) >

finds the minimum link-weighted cut of an input graph. For more information, see the section “Minimum Cut” on page 712. You can specify the following suboptions in the MINCUT= option:

MAXCUTS=number
specifies the maximum number of cuts to return from the algorithm. The minimal cut and any others found during the search, up to number, are returned. By default, MAXCUTS=1.

MAXWEIGHT=number
specifies the maximum weight of the cuts to return from the algorithm. Only cuts that have weight less than or equal to number are returned. The default is the positive number that has the largest absolute value that can be represented in your operating environment.

MINSPANTREE< () >

MST< () >
solves the minimum link-weighted spanning tree problem on an input graph. For more information, see the section “Minimum Spanning Tree” on page 716.

SHORTESTPATH< ( suboptions ) >
SHORTPATH< ( suboptions ) >
calculates shortest paths between sets of nodes on the input graph. For more information, see the section “Shortest Path” on page 718.

You can specify the following suboptions:

MAXPATHWEIGHT=number
specifies the maximum path weight.

Any shortest path whose sum of link weights is greater than number is removed from the results. The default is the positive number that has the largest absolute value that can be represented in your operating environment, which causes no paths to be removed from the results.

SINK=set-name
specifies the set of sink nodes.
**SOURCE**=*set-name*

specifies the set of source nodes.

**TRANSITIVECLOSURE**<=()>

**TRANSCL**<=()>

calculates the transitive closure of an input graph. For more information, see the section “Transitive Closure” on page 731.

**TSP**<=( *suboptions* )>

solves the traveling salesman problem. For more information, see the section “Traveling Salesman Problem” on page 734.

The algorithm that is used to solve this problem is built around the same method that is used in PROC OPTMILP: a branch-and-cut algorithm. Many of the following *suboptions* are the same as those described for the OPTMILP procedure in Chapter 7, “The OPTMILP Procedure.”

You can specify the following *suboptions*:

**ABSOBJGAP**=*number*

specifies a stopping criterion. When the absolute difference between the best integer objective and the objective of the best remaining branch-and-bound node becomes less than the value of *number*, the solver stops. The value of *number* can be any nonnegative number. By default, ABSOBJGAP=1E–6.

**CUTOFF**=*number*

cuts off any branch-and-bound nodes in a minimization problem that has an objective value that is greater than *number*. The value of *number* can be any number.

The default value is the positive number that has the largest absolute value that can be represented in your operating environment.

**CUTSTRATEGY**=*number* | *string*

specifies the level of cutting planes to be generated by the network solver. TSP-specific cutting planes are always generated. Table 15.10 describes the valid values for this option.

<table>
<thead>
<tr>
<th><em>number</em></th>
<th><em>string</em></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>–1</td>
<td>AUTOMATIC</td>
<td>Generates cutting planes based on a strategy determined by the mixed integer linear programming solver</td>
</tr>
<tr>
<td>0</td>
<td>NONE</td>
<td>Disables generation of mixed integer programming cutting planes (some TSP-specific cutting planes are still active for validity)</td>
</tr>
<tr>
<td>1</td>
<td>MODERATE</td>
<td>Uses a moderate cut strategy</td>
</tr>
<tr>
<td>2</td>
<td>AGGRESSIVE</td>
<td>Uses an aggressive cut strategy</td>
</tr>
</tbody>
</table>

By default, CUTSTRATEGY=NONE.
HEURISTICS=number | string
controls the level of initial and primal heuristics that the network solver applies. This level
determines how frequently the network solver applies primal heuristics during the branch-and-bound
tree search. It also affects the maximum number of iterations that are allowed in iterative
heuristics. Some computationally expensive heuristics might be disabled by the solver at less
aggressive levels. Table 15.11 lists the valid values for this option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>–1</td>
<td>AUTOMATIC</td>
<td>Applies the default level of heuristics</td>
</tr>
<tr>
<td>0</td>
<td>NONE</td>
<td>Disables all initial and primal heuristics</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Applies basic initial and primal heuristics at low frequency</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Applies most initial and primal heuristics at moderate frequency</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Applies all initial primal heuristics at high frequency</td>
</tr>
</tbody>
</table>

By default, HEURISTICS=AUTOMATIC.

MAXNODES=number
specifies the maximum number of branch-and-bound nodes to be processed. The value of number
can be any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$.

By default, MAXNODES=$2^{31} - 1$.

MAXSOLS=number
specifies the maximum number of feasible tours to be identified. If number solutions have been
found, then the solver stops. The value of number can be any positive integer up to the largest
four-byte signed integer, which is $2^{31} - 1$.

By default, MAXSOLS=$2^{31} - 1$.

MILP=number | string
specifies whether to use a mixed integer linear programming (MILP) solver for solving the
traveling salesman problem. The MILP solver attempts to find the overall best TSP tour by using
a branch-and-bound based algorithm. This algorithm can be expensive for large-scale problems.
If MILP=FALSE, then the network solver uses its initial heuristics to find a feasible, but not
necessarily optimal, tour as quickly as possible. Table 15.12 describes the valid values for this
option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TRUE</td>
<td>Uses a mixed integer linear programming solver</td>
</tr>
<tr>
<td>0</td>
<td>FALSE</td>
<td>Does not use a mixed integer linear programming solver</td>
</tr>
</tbody>
</table>

By default, MILP=TRUE.
**RELOBJGAP=number**

specifies a stopping criterion that is based on the best integer objective (BestInteger) and the objective of the best remaining node (BestBound). The relative objective gap is equal to

\[
\frac{|\text{BestInteger} - \text{BestBound}|}{(1E-10 + |\text{BestBound}|)}
\]

When this value becomes less than the specified gap size `number`, the solver stops. The value of `number` can be any nonnegative number.

By default, RELOBJGAP=1E-4.

**TARGET=number**

specifies a stopping criterion for minimization problems. If the best integer objective is better than or equal to `number`, the solver stops. The value of `number` can be any number.

By default, TARGET is the negative number that has the largest absolute value that can be represented in your operating environment.

---

**Details: Network Solver**

The network solver uses a collection of specialized algorithms that optimize specific types of common problems. When you use the network solver, you specify variable arrays, numeric arrays, and sets, both to define an instance and to get solutions, without explicitly formulating objectives and constraints.

---

**Input Data for the Network Solver**

This section describes how you can import and export node, link, and problem data from and to SAS data sets and how you can solve problems over a subgraph without changing your original sets. The section “Graph Input Data” on page 680 describes how to load node and link data in some common formats. The section “Solving over Subsets of Nodes and Links (Filters)” on page 683 describes subgraphs and how to access the objective value of a network problem.

---

**Graph Input Data**

This section describes how to input a graph for analysis by the network solver. Because PROC OPTMODEL uses node and link attributes that are indexed over the sets of nodes and links, you need to provide only node and link attributes. PROC OPTMODEL infers the graph from the attributes you provide. When a documented default value exists for the attribute of a link or a node, you need to provide only the values that differ from the default. For example, the section “Minimum-Cost Network Flow” on page 706 assumes that the link flow upper bound is \(\infty\). You need to specify only the finite upper bounds.

Consider the directed graph shown in Figure 15.5.
Each node and link has associated attributes: a node label and a link weight.

None of the algorithms in PROC OPTMODEL support Null Graphs, i.e., graphs with 0 nodes. PROC OPTMODEL will usually raise a semantic error and stop processing any remaining statements in a block if after processing its inputs it determines that the graph is null. If the graph definition itself is not null, but the graph to be passed to the solver after applying the SUBGRAPH= option is null, then the predeclared parameter _SOLUTION_STATUS_ will be set to NULL_GRAPH. For more information, see “Solving over Subsets of Nodes and Links (Filters)” on page 683.

Data Indexed by Nodes or Links

Nodes often represent entities, and links represent relationships between these entities. Therefore, it is common to store a graph as a link-indexed table. When nodes have attributes beyond their name (label), these attributes are stored in a node-indexed table. This section covers the more complex link-indexed case. The node-indexed case is essentially identical to this one, except that the PROC OPTMODEL set has tuple length of one when node-indexed data are read, whereas the PROC OPTMODEL set has tuple length two when link-indexed data are read.

Let \( G = (N, A) \) define a graph with a set \( N \) of nodes and a set \( A \) of links. A link is an ordered pair of nodes. Each node is defined by using either numeric or string labels.

The directed graph \( G \) shown in Figure 15.5 can be represented by the following links data set LinkSetIn:

```plaintext
data LinkSetIn;
  input from $ to $ weight @@;
datalines;
 A B 1 A C 2 A D 4 B C 1 B E 2
 B F 5 C E 1 D E 1 E D 1 E F 2
 F G 6 G H 1 G I 1 H G 2 H I 3
;
```

The following statements read in this graph and print the resulting links and nodes sets. These statements do not run any algorithms, so the resulting output contains only the input graph.
proc optmodel;
    set<str,str> LINKS;
    set NODES = union{<ni,nj> in LINKS} {ni,nj};
    num weight{LINKS};

    read data LinkSetIn into LINKS=[from to] weight;

    print weight;
    put NODES=; /* computed automatically by OPTMODEL */
    quit;

The network solver preserves the node order of each link that you provide, even in cases where the link
is traversed in the opposite order, such as in paths or tours. For an example, see the tour \(1, 4, 2, 3, 5\) in
Figure 15.8.

The log output in Figure 15.6 shows the nodes that are read from the input link data set. In this example
PROC OPTMODEL computed the node set \(N\) (NODES) from its definition when it was needed. The ODS
output in Figure 15.7 shows the weights that are read from the input link data set, which is indexed by link.
PUT is used for NODES because PROC OPTMODEL sets are basic types such as number and string. Thus,
you use PUT to quickly inspect a set value. In contrast, you use PRINT to inspect an array, such as weight.

**Figure 15.6** Node Set Printout of a Simple Directed Graph

| NODES=\{’A’,’B’,’C’,’D’,’E’,’F’,’G’,’H’,’I’\} |

As described in the GRAPH_DIRECTION= option, if the graph is undirected, the from and to labels are
interchangeable. If you define this graph as undirected, then reciprocal links (for example, \(D \rightarrow E\) and
\(E \rightarrow D\)) are treated as the same link, and duplicates are removed. The network solver takes the first
occurrence of the link and ignores the others. By default, GRAPH\_DIRECTION=UNDIRECTED, so to declare the graph as undirected you can just omit this option.

After you read the data into PROC OPTMODEL sets, you pass link information to the solver by using the LINKS= option. Node input is analogous to link input. You pass node information to the solver by using the NODES= option.

The INCLUDE= suboption is especially useful for algorithms that depend only on the graph topology, (such as the connected components algorithm). If an algorithm requires a node or link property and that property is not defined for a node or link that is added by the INCLUDE= suboption, the algorithm will not run.

**Matrix Input Data**

The contents of a table can be represented as a graph. The relationships between two sets of nodes, \( N_1 \) and \( N_2 \), can be represented by a \(|N_1| \times |N_2|\) incidence matrix \( A \), in which \( N_1 \) is the set of rows and \( N_2 \) is the set of columns.

To read a matrix that is stored in a data set into PROC OPTMODEL, you need to take two extra steps:

1. Determine the name of each numeric variable that you want to use. PROC CONTENTS can be useful for this task.
2. Use an iterated READ DATA statement.

For more information, see “Example 15.3: Linear Assignment Problem for Minimizing Relay Times” on page 751.

---

**Solving over Subsets of Nodes and Links (Filters)**

You can solve a problem over a subgraph without declaring new link and node sets. You can specify the LINKS= and NODES= suboptions of the SUBGRAPH= option to filter nodes and links before PROC OPTMODEL builds and solves the instance. If you want to see the resulting subgraph, you can specify the LINKS= and NODES= suboptions of the OUT= option. If you just want to produce a subgraph, you do not need to invoke an algorithm.

You can keep all the input and output arrays defined over the original graph and define a subgraph by providing any combination of the LINKS= and NODES= suboptions of the SUBGRAPH= option. If you specify either of the suboptions of the SUBGRAPH= option, then union semantics apply. PROC OPTMODEL uses the following rules:

- Only the links that are included in the set named in the LINKS= option are used to create the instance.
- Only the nodes that appear either in the NODES= suboption of the SUBGRAPH= option or that appear as the head or tail of a link in the LINKS= suboption are used to create the instance.
- A node or a link that appears only in the SUBGRAPH= option, but not in the original graph, is discarded. To add nodes or links that do not have attributes, see the INCLUDE= suboption of the LINKS= and NODE= options.
If the value of the LOGLEVEL= suboption is equal to or greater than 3, PROC OPTMODEL issues a message for each of the nodes and links that it discards until the number of messages issued during problem generation reaches the value of the MSGLIMIT= option in the PROC OPTMODEL statement. If the value of the LOGLEVEL= suboption is greater than 0, PROC OPTMODEL also issues a summary that shows the total count of discarded nodes and links from each input array or set.

The following statements call PROC OPTMODEL and declare a five-node complete undirected graph; a subset of links that contains all links between nodes 1, 2, 3, and 4; and a subset of nodes that contains nodes 3, 4, and 5:

```plaintext
proc optmodel;
  set NODES = 1..5;
  set LINKS = {vi in NODES, vj in NODES: vi < vj};
  num distance {<vi,vj> in LINKS} = 10*vi + vj;

  set <num,num> TOUR;

  /* Build a link set using only nodes 1..4 nodes */
  set <num,num> LINKS_BETWEEN_1234 = {vi in 1..3, vj in (vi+1)..4};

  /* Build a node subset consisting of nodes 3..5 */
  set NODES_345 = 3..5;

After the sets are declared, the statements in the following steps solve several traveling salesman problems (TSPs) on subgraphs. For more information about TSPs, see the section “Traveling Salesman Problem” on page 734.

1. The first SOLVE statement solves a TSP on the original graph. Note that the links in the tour (see Figure 15.8) are returned with the same orientation that you provide in the input. For example, the second step on the tour goes from node 4 to node 2 using link (2, 4). This guarantees that you do not need to do extra processing of output to check for link orientation. You can just use the output directly.

```plaintext
/* Implicit network 1: solve over nodes 1..5 -- The original network*/
solve with NETWORK /
      links=( weight=distance )
      out=( tour=TOUR )
      tsp
    ;
put TOUR=;
```

As shown in Figure 15.8, all links implied by the WEIGHT= suboption of the LINKS= option become part of the graph.
To access the objective value of a network problem, use the _OROPTMODEL_NUM_ predefined array. The network solver ignores the _OBJ_ predefined symbol, which is part of the current named problem. The current named problem is independent of the network solver, because the network solver uses sets and numeric arrays for input and output. For more information, see the sections “Multiple Subproblems” on page 397 and “Solver Status Parameters” on page 408 in Chapter 10, “The OPTMODEL Procedure.”

```plaintext
put _OROPTMODEL_NUM_['OBJECTIVE'];
```

2. The next SOLVE statement solves a TSP on the subgraph that is defined by the link set LINKS_BETWEEN_1234.

```plaintext
/* Filter on LINKS: solve over nodes 1..4 */
solve with NETWORK /
   links=( weight=distance )
   subgraph=( links=LINKS_BETWEEN_1234 )
   out=( tour=TOUR )
   tsp
;
put TOUR=;
```

As shown in Figure 15.9, the network solver now ignores node 5.
Figure 15.9  SOLVE WITH NETWORK Log: Traveling Salesman Tour over Nodes $N = \{1, 2, 3, 4\}$

111
NOTE: The SUBGRAPH= option filtered 4 elements from 'distance.'
NOTE: The number of nodes in the input graph is 4.
NOTE: The number of links in the input graph is 6.
NOTE: The network solver is called.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 74 using 0.00 (cpu: 0.00) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Branch and Cut algorithm is used.
NOTE: Optimal.
NOTE: Objective = 74.
NOTE: Processing the traveling salesman problem used 0.00 (cpu: 0.00) seconds.
TOUR=\{<1,3>,<2,3>,<2,4>,<1,4>\}

3. The next SOLVE statement solves a TSP on the subgraph that is defined by the node set NODES_345.

        /* Filter on NODES: solve over nodes 3..5 */
        solve with NETWORK /
        links=( weight=distance )
        subgraph=( nodes=NODES_345 )
        out=( tour=TOUR )
        tsp
        ;
        put TOUR=;

As shown in Figure 15.10, the network solver now ignores nodes 1 and 2, along with any links incident to them.

Figure 15.10  SOLVE WITH NETWORK Log: Traveling Salesman Tour over Nodes $N = \{3, 4, 5\}$

NOTE: The SUBGRAPH= option filtered 7 elements from 'distance.'
NOTE: The number of nodes in the input graph is 3.
NOTE: The number of links in the input graph is 3.
NOTE: The network solver is called.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 114 using 0.00 (cpu: 0.00) seconds.
NOTE: Optimal.
NOTE: Objective = 114.
NOTE: Processing the traveling salesman problem used 0.00 (cpu: 0.00) seconds.
TOUR=\{<3,4>,<4,5>,<3,5>\}

4. The next SOLVE statement attempts to solve a TSP on the subgraph that is defined by the node set NODES_345 and the link set that is defined by the links on the nodes \{1, 2, 3, 4\}. This subgraph creates an infeasible instance because the links \{(1, 5), (2, 5), (3, 5), (4, 5)\} that were defined in the original graph have been filtered out. Thus, node 5 is disconnected and no tour can exist.
/* Explicit nodes and links: semantic error over nodes 1..5
   * Links <u,5> are undefined and no documented default exists. */
solve with NETWORK /
   links=( weight=distance )
   subgraph=( nodes=NODES_345 links=LINKS_BETWEEN_1234 )
   out=( tour=TOUR )
   tsp
;

As shown in Figure 15.11, the network solver identifies that no tour exists over the surviving nodes and links.

Figure 15.11 SOLVE WITH NETWORK Log: Infeasible Traveling Salesman Problem after Filtering

NOTE: The SUBGRAPH= option filtered 4 elements from 'distance.'
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 6.
NOTE: The number of singleton nodes in the input graph is 1.
NOTE: The network solver is called.
NOTE: Processing the traveling salesman problem.
NOTE: Infeasible.
NOTE: Processing the traveling salesman problem used 0.00 (cpu: 0.00) seconds.

5. The last SOLVE statement uses the LINKS= suboption of the OUT= option to capture exactly which nodes and links were generated and with which attributes. In this case, because the only attribute defined is link weight, the set LINKS_OUT has tuples of length three.

   /* make room for tail, head, and weight */
   set<num,num,num> LINKS_OUT;
   solve with NETWORK /
      links=( weight=distance )
      subgraph=( nodes=NODES_345 links=LINKS_BETWEEN_1234 )
      out=( tour=TOUR links=LINKS_OUT )
      tsp
   ;
   put LINKS_OUT=;
   quit;

As shown in Figure 15.12, the network solver can return the graph after filtering. This feature can sometimes help you identify why you might get counterintuitive results.

Figure 15.12 SOLVE WITH NETWORK Log: Remaining Links after Filtering

NOTE: The SUBGRAPH= option filtered 4 elements from 'distance.'
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 6.
NOTE: The number of singleton nodes in the input graph is 1.
NOTE: The network solver is called.
NOTE: Processing the traveling salesman problem.
NOTE: Infeasible.
NOTE: Processing the traveling salesman problem used 0.00 (cpu: 0.00) seconds.
LINKS_OUT={<1,2,12>,<1,3,13>,<1,4,14>,<2,3,23>,<2,4,24>,<3,4,34>}
Numeric Limitations

Extremely large or extremely small numerical values might cause computational difficulties for some of the algorithms in the network solver. For this reason, each algorithm restricts the magnitude of the data values to a particular threshold number. If the user data values exceed this threshold, the network solver issues an error message. The value of the threshold limit is different for each algorithm and depends on the operating environment. The threshold limits are listed in Table 15.13, where $M$ is defined as the largest absolute value representable in your operating environment.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Graph Links</th>
<th>Graph Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>weight</td>
<td>lower</td>
</tr>
<tr>
<td>CYCLE</td>
<td>$\sqrt{M}$</td>
<td>$\sqrt{M}$</td>
</tr>
<tr>
<td>LINEARASSIGNMENT</td>
<td>$\sqrt{M}$</td>
<td></td>
</tr>
<tr>
<td>MINCOSTFLOW</td>
<td>1e15</td>
<td>1e15</td>
</tr>
<tr>
<td>MINCUT</td>
<td>$\sqrt{M}$</td>
<td></td>
</tr>
<tr>
<td>MINSPANTREE</td>
<td>$\sqrt{M}$</td>
<td></td>
</tr>
<tr>
<td>SHORTESTPATH</td>
<td>$\sqrt{M}$</td>
<td></td>
</tr>
<tr>
<td>TSP</td>
<td>1e20</td>
<td></td>
</tr>
</tbody>
</table>

To obtain these limits, use the SAS CONSTANT function. For example, the following PROC OPTMODEL code assigns $\sqrt{M}$ to a variable $x$ and prints that value to the log:

```sas
proc optmodel;
  num c = constant('SQRTBIG');
  put c=;
quit;
```

Missing Values

A missing value has no valid interpretation for most of the algorithms in the network solver. If the user data contain a missing value, the network solver issues an error message. There is only one exception: the minimum-cost network flow algorithm interprets a missing value in the lower or upper bound option as the default bound value. For more information about this algorithm, see the section “Minimum-Cost Network Flow” on page 706.

Negative Link Weights

For certain algorithms in the network solver, a negative link weight is not allowed. The following algorithms issue an error message if a negative link weight is provided:

- MINCUT
Biconnected Components and Articulation Points

A **biconnected component** of a graph \( G = (N, A) \) is a connected subgraph that cannot be broken into disconnected pieces by deleting any single node (and its incident links). An **articulation point** is a node of a graph whose removal would cause an increase in the number of connected components. Articulation points can be important when you analyze any graph that represents a communications network. Consider an articulation point \( i \in N \) which, if removed, disconnects the graph into two components \( C^1 \) and \( C^2 \). All paths in \( G \) between some nodes in \( C^1 \) and some nodes in \( C^2 \) must pass through node \( i \). In this sense, articulation points are critical to communication. Examples of where articulation points are important are airline hubs, electric circuits, network wires, protein bonds, traffic routers, and numerous other industrial applications.

In the network solver, you can find biconnected components and articulation points of an input graph by invoking the BICONNECTEDCOMPONENTS option. This algorithm works only with undirected graphs.

The results for the biconnected components algorithm are written to the link-indexed numeric array that is specified in the BICONCOMP= suboption of the OUT= option. For each link in the links array, the value in this array identifies its component. The component identifiers are numbered sequentially starting from 1. The articulation points are written to the set that is specified in the ARTPOINTS= suboption of the OUT= option.

The algorithm that the network solver uses to compute biconnected components is a variant of depth-first search (Tarjan 1972). This algorithm runs in time \( O(|N| + |A|) \) and therefore should scale to very large graphs.

**Biconnected Components of a Simple Undirected Graph**

This section illustrates the use of the biconnected components algorithm on the simple undirected graph \( G \) that is shown in Figure 15.13.

![Figure 15.13 A Simple Undirected Graph G](image)
Chapter 15: The Network Solver

The undirected graph $G$ can be represented by the links data set LinkSetInBiCC as follows:

```plaintext
data LinkSetInBiCC;
   input from $ to $ @@;
datalines;
  A B A F A G B C B D
  B E C D E F G I G H
  H I
;```

The following statements calculate the biconnected components and articulation points and output the results in the data sets LinkSetOut and NodeSetOut:

```plaintext
data LinkSetInBiCC;
   input from $ to $ @@;
datalines;
  A B A F A G B C B D
  B E C D E F G I G H
  H I
;```

The following statements calculate the biconnected components and articulation points and output the results in the data sets LinkSetOut and NodeSetOut:

```plaintext
proc optmodel;
   set<str,str> LINKS;
   read data LinkSetInBiCC into LINKS=[from to];
   set NODES = union{<i,j> in LINKS} {i,j};
   num bicomponent{LINKS};
   set<str> ARTPOINTS;
   solve with NETWORK / loglevel = moderate
       links = (include=LINKS)
       biconcomp
       out = (biconcomp=bicomponent artpoints=ARTPOINTS)
   ;
   print bicomponent;
   put ARTPOINTS;
   create data LinkSetOut from [from to] biconcomp=bicomponent;
   create data NodeSetOut from [node]=ARTPOINTS artpoint=1;
quit;
```

The data set LinkSetOut now contains the biconnected components of the input graph, as shown in Figure 15.14.

**Figure 15.14** Biconnected Components of a Simple Undirected Graph

<table>
<thead>
<tr>
<th>from to</th>
<th>biconcomp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A B</td>
<td>2</td>
</tr>
<tr>
<td>A F</td>
<td>2</td>
</tr>
<tr>
<td>A G</td>
<td>4</td>
</tr>
<tr>
<td>B C</td>
<td>1</td>
</tr>
<tr>
<td>B D</td>
<td>1</td>
</tr>
<tr>
<td>B E</td>
<td>2</td>
</tr>
<tr>
<td>C D</td>
<td>1</td>
</tr>
<tr>
<td>E F</td>
<td>2</td>
</tr>
<tr>
<td>G I</td>
<td>3</td>
</tr>
<tr>
<td>G H</td>
<td>3</td>
</tr>
<tr>
<td>H I</td>
<td>3</td>
</tr>
</tbody>
</table>

In addition, the data set NodeSetOut contains the articulation points of the input graph, as shown in Figure 15.15.
The biconnected components are shown graphically in Figure 15.16 and Figure 15.17.

**Figure 15.16**  Biconnected Components $C^1$ and $C^2$

$C^1 = \{B, C, D\}$ \hspace{1cm} $C^2 = \{A, B, E, F\}$

**Figure 15.17**  Biconnected Components $C^3$ and $C^4$

$C^3 = \{G, H, I\}$ \hspace{1cm} $C^4 = \{A, G\}$
Clique

A clique of a graph $G = (N, A)$ is an induced subgraph that is a complete graph. Every node in a clique is connected to every other node in that clique. A maximal clique is a clique that is not a subset of the nodes of any larger clique. That is, it is a set $C$ of nodes such that every pair of nodes in $C$ is connected by a link and every node not in $C$ is missing a link to at least one node in $C$. The number of maximal cliques in a particular graph can be very large and can grow exponentially with every node added. Finding cliques in graphs has applications in numerous industries including bioinformatics, social networks, electrical engineering, and chemistry.

You can find the maximal cliques of an input graph by invoking the CLIQUE= option. The clique algorithm works only with undirected graphs.

The results for the clique algorithm are written to the set that is specified in the CLIQUES= suboption of the OUT= option. Each node of each clique is listed in the set along with a clique ID (the first argument of the tuple) to identify the clique to which it belongs. A node can appear multiple times in this set if it belongs to multiple cliques.

The algorithm that the network solver uses to compute maximal cliques is a variant of the Bron-Kerbosch algorithm (Bron and Kerbosch 1973; Harley 2003). Enumerating all maximal cliques is NP-hard, so this algorithm typically does not scale to very large graphs.

Maximal Cliques of a Simple Undirected Graph

This section illustrates the use of the clique algorithm on the simple undirected graph $G$ that is shown in Figure 15.18.

**Figure 15.18**  A Simple Undirected Graph $G
The undirected graph $G$ can be represented by the following links data set `LinkSetIn`:

```plaintext
data LinkSetIn;
  input from to @@;
datalines;
  0 1 0 2 0 3 0 4 0 5
  0 6 1 2 1 3 1 4 2 3
  2 4 2 5 2 6 2 7 2 8
  3 4 5 6 7 8 8 9
;
```

The following statements calculate the maximal cliques, output the results in the data set `Cliques`, and use the `CARD` function and `SLICE` operator as a convenient way to compute the clique sizes, which are output to a data set called `CliqueSizes`:

```plaintext
proc optmodel;
  set<num,num> LINKS;
  read data LinkSetIn into LINKS=[from to];
  set<num,num> CLIQUES;

  solve with NETWORK /
    links = (include=LINKS)
    clique = (maxcliques=all)
    out = (cliques=CLIQUES);

  put CLIQUES;
  create data Cliques from [clique node]=CLIQUES;
  num num_cliques = card(setof {<cid,node> in CLIQUES} cid);
  set CLIQUE_IDS = 1..num_cliques;
  num size {cid in CLIQUE_IDS} = card(slice(<cid,*>, CLIQUES));
  create data CliqueSizes from [clique] size;
quit;
```

The data set `Cliques` now contains the maximal cliques of the input graph; it is shown in Figure 15.19.

**Figure 15.19** Maximal Cliques of a Simple Undirected Graph

<table>
<thead>
<tr>
<th>clique node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>8</td>
</tr>
</tbody>
</table>


In addition, the data set CliqueSizes contains the number of nodes in each clique; it is shown in Figure 15.20.

**Figure 15.20** Sizes of Maximal Cliques of a Simple Undirected Graph

<table>
<thead>
<tr>
<th>clique size</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The maximal cliques are shown graphically in Figure 15.21 and Figure 15.22.

**Figure 15.21** Maximal Cliques $C^1$ and $C^2$

$C^1 = \{0, 1, 2, 3, 4\}$  \hspace{1cm} $C^2 = \{0, 2, 5, 6\}$

**Figure 15.22** Maximal Cliques $C^3$ and $C^4$

$C^3 = \{2, 7, 8\}$  \hspace{1cm} $C^4 = \{8, 9\}$
Connected Components

A connected component of a graph is a set of nodes that are all reachable from each other. That is, if two nodes are in the same component, then there exists a path between them. For a directed graph, there are two types of components: a strongly connected component has a directed path between any two nodes, and a weakly connected component ignores direction and requires only that a path exist between any two nodes.

In the network solver, you can invoke connected components by using the CONNECTEDCOMPONENTS= option.

There are two main algorithms for finding connected components in an undirected graph: a depth-first search algorithm (ALGORITHM=DFS) and a union-find algorithm (ALGORITHM=UNIONFIND). For a graph $G = (N, A)$, both algorithms run in time $O(|N| + |A|)$ and can usually scale to very large graphs. The default is the union-find algorithm. For directed graphs, only the depth-first search algorithm is available.

The results of the connected components algorithm are written to the node-indexed numeric array that you specify in the CONCOMP= suboption of the OUT= option. For each node in the set, the value of this array identifies its component. The component identifiers are numbered sequentially starting from 1.

Connected Components of a Simple Undirected Graph

This section illustrates the use of the connected components algorithm on the simple undirected graph $G$ that is shown in Figure 15.23.

Figure 15.23  A Simple Undirected Graph $G$

The undirected graph $G$ can be represented by the following links data set, LinkSetIn:

```sql
data LinkSetIn;
    input from $ to $ @@;
datalines;
    A B A C B C C H D E D F D G F E G I K L
;
```

The following statements calculate the connected components and output the results in the data set Node-SetOut:
Chapter 15: The Network Solver

```
proc optmodel;
  set<str,str> LINKS;
  read data LinkSetIn into LINKS=[from to];
  set NODES = union {<i,j> in LINKS} {i,j};
  num component{NODES};

  solve with NETWORK / 
      links = (include=LINKS)
      concomp
          out = (concomp=component)
  ;

  print component;
  create data NodeSetOut from [node] concomp=component;
quit;
```

The data set NodeSetOut contains the connected components of the input graph and is shown in Figure 15.24.

**Figure 15.24** Connected Components of a Simple Undirected Graph

<table>
<thead>
<tr>
<th>node</th>
<th>concomp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
</tr>
<tr>
<td>K</td>
<td>3</td>
</tr>
<tr>
<td>L</td>
<td>3</td>
</tr>
</tbody>
</table>

Notice that the graph is defined by using only the links array. As seen in Figure 15.23, this graph also contains a singleton node labeled J, which has no associated links. By definition, this node defines its own component. But because the input graph was defined by using only the links array, node J did not show up in the results data set. To define a graph by using nodes that have no associated links, you should also define the input nodes set. In this case, you can define a nodes data set NodeSetIn as follows:

```
data NodeSetIn;
  input node $ @@;
datalines;
  A B C D E F G H I J K L
QUIT;
```

You could also have defined the set directly in PROC OPTMODEL, but in this case, a separate data set nicely preserves the independence between the model and the data.

Now, when you calculate the connected components, you define the input graph by using both the nodes input data set and the links input data set:
proc optmodel;
set<str,str> LINKS;
read data LinkSetIn into LINKS=[from to];
set<str> NODES;
read data NodeSetIn into NODES=[node];
num component{NODES};
solve with NETWORK /
   links   = (include=LINKS)
   nodes   = (include=NODES)
   concomp
   out     = (concomp=component)
;
print component;
create data NodeSetOut from [node] concomp=component;
quit;

The resulting data set, NodeSetOut, includes the singleton node J as its own component, as shown in Figure 15.25.

**Figure 15.25** Connected Components of a Simple Undirected Graph

<table>
<thead>
<tr>
<th>node</th>
<th>concomp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
</tr>
<tr>
<td>J</td>
<td>3</td>
</tr>
<tr>
<td>K</td>
<td>4</td>
</tr>
<tr>
<td>L</td>
<td>4</td>
</tr>
</tbody>
</table>

**Connected Components of a Simple Directed Graph**

This section illustrates the use of the connected components algorithm on the simple directed graph $G$ that is shown in Figure 15.26.
The directed graph $G$ can be represented by the following links data set, LinkSetIn:

```plaintext
data LinkSetIn;
    input from $ to $ @@;
datalines;
A B B C B E B F C G
C D D C D H E A E F
F G G F H G H D
;
```

The following statements calculate the connected components and output the results in the data set NodeSetOut:

```plaintext
proc optmodel;
    set<str,str> LINKS;
    read data LinkSetIn into LINKS=[from to];
    set NODES = union {<i,j> in LINKS} {i,j};
    num component{NODES};

    solve with NETWORK /
        graph_direction = directed
        links = (include=LINKS)
        concomp
        out = (concomp=component)
    ;

    print component;
    create data NodeSetOut from [node] concomp=component;
quit;
```
The data set NodeSetOut, shown in Figure 15.27, now contains the connected components of the input graph.

**Figure 15.27** Connected Components of a Simple Directed Graph

<table>
<thead>
<tr>
<th>node</th>
<th>concomp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>H</td>
<td>2</td>
</tr>
</tbody>
</table>

The connected components are represented graphically in Figure 15.28.

**Figure 15.28** Strongly Connected Components of Graph G

---

**Cycle**

A *path* in a graph is a sequence of nodes, each of which has a link to the next node in the sequence. An *elementary cycle* is a path in which the start node and end node are the same and otherwise no node appears more than once in the sequence.

In the network solver, you can find (or just count) the elementary cycles of an input graph by invoking the CYCLE= algorithm option. To find the cycles and report them in a set, use the CYCLES= suboption in the OUT= option. You do not need to use the CYCLES= suboption to simply count the cycles.
For undirected graphs, each link represents two directed links. For this reason, the following cycles are filtered out: trivial cycles \((A \rightarrow B \rightarrow A)\) and duplicate cycles that are found by traversing a cycle in both directions \((A \rightarrow B \rightarrow C \rightarrow A \text{ and } A \rightarrow C \rightarrow B \rightarrow A)\).

The results of the cycle detection algorithm are written to the set that you specify in the CYCLES= suboption in the OUT= option. Each node of each cycle is listed in the CYCLES= set along with a cycle ID (the first argument of the tuple) to identify the cycle to which it belongs. The second argument of the tuple defines the order (sequence) of the node in the cycle.

The algorithm that the network solver uses to compute all cycles when the value of the MAXLENGTH= option is greater than 20 is a variant of the algorithm in Johnson (1975) (ALGORITHM=BACKTRACK). This algorithm runs in time \(O\left(|V| + |A|\left(c + 1\right)\right)\), where \(c\) is the number of elementary cycles in the graph. So the algorithm should scale to large graphs that contain few cycles. However, some graphs can have a very large number of cycles, so the algorithm might not scale. The default when the value of the MAXLENGTH= option is less than or equal to 20 is described in Liu and Wang (2006) (ALGORITHM=BUILD). This algorithm is typically much faster than the backtracking algorithm when the length of the cycles is sufficiently restricted.

If MAXCYCLES=ALL and there are many cycles, the CYCLES= set can become very large. It might be beneficial to check the number of cycles before you try to create the CYCLES= set. By default, MAXCYCLES=1, so the algorithm returns the first cycle that it finds and stops processing. This should run relatively quickly. For large-scale graphs, the MINLINKWEIGHT= and MAXLINKWEIGHT= suboptions might increase the computation time.

**Cycle Detection of a Simple Directed Graph**

This section provides a simple example of using the cycle detection algorithm on the simple directed graph \(G\) that is shown in Figure 15.29. Two other examples are “Example 15.2: Cycle Detection for Kidney Donor Exchange” on page 746, which shows the use of cycle detection for optimizing a kidney donor exchange, and “Example 15.6: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System” on page 759, which shows an application of cycle detection to dependencies between bug reports.

**Figure 15.29** A Simple Directed Graph \(G\)
The directed graph $G$ can be represented by the following links data set, LinkSetIn:

```plaintext
data LinkSetIn;
    input from $ to $ @@;
datalines;
A B A E B C C A C D
D E D F E B E C F E
;
```

The following statements check whether the graph has a cycle:

```plaintext
proc optmodel;
    set<str,str> LINKS;
    read data LinkSetIn into LINKS=[from to];
    set<num,num,str> CYCLES;

    solve with NETWORK /
        graph_direction = directed
        links = (include=LINKS)
        cycle = (maxcycles=1)
    ;
quit;
```

The result is written to the log of the procedure, as shown in Figure 15.30.

**Figure 15.30** Network Solver Log: Check the Existence of a Cycle in a Simple Directed Graph

NOTE: There were 10 observations read from the data set WORK.LINKSETIN.
NOTE: The number of nodes in the input graph is 6.
NOTE: The number of links in the input graph is 10.
NOTE: The network solver is called.
NOTE: Processing cycle detection.
NOTE: Required number of cycles found. You can increase this value using the MAXCYCLES= option.
NOTE: The algorithm found 1 cycles.
NOTE: Processing cycle detection used 0.00 (cpu: 0.00) seconds.

The following statements count the number of cycles in the graph:

```plaintext
proc optmodel;
    set<str,str> LINKS;
    read data LinkSetIn into LINKS=[from to];
    set<num,num,str> CYCLES;

    solve with NETWORK /
        graph_direction = directed
        links = (include=LINKS)
        cycle = (maxcycles=all)
    ;
quit;
```

The result is written to the log of the procedure, as shown in Figure 15.31.
The following statements return the first cycle found in the graph:

```plaintext
proc optmodel;
    set<str,str> LINKS;
    read data LinkSetIn into LINKS=[from to];
    set<num,num,str> CYCLES;
    solve with NETWORK /
        graph_direction = directed
        links = (include=LINKS)
        cycle = (maxcycles=1)
        out = (cycles=CYCLES)
    ;
    put CYCLES;
    create data Cycles from [cycle order node]=CYCLES;
quit;
```

The data set Cycles now contains the first cycle found in the input graph; it is shown in Figure 15.32.

**Figure 15.32** First Cycle Found in a Simple Directed Graph

<table>
<thead>
<tr>
<th>cycle</th>
<th>order</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>A</td>
</tr>
</tbody>
</table>

The first cycle that is found in the input graph is shown graphically in Figure 15.33.
The following statements return all the cycles in the graph:

```plaintext
proc optmodel;
    set<str,str> LINKS;
    read data LinkSetIn into LINKS=[from to];
    set<num,num,str> CYCLES;

    solve with NETWORK /
        graph_direction = directed
        links = (include=LINKS)
        cycle = (maxcycles=all)
        out = (cycles=CYCLES)
    ;

    put CYCLES;
    create data Cycles from [cycle order node]=CYCLES;
quit;
```

The data set `Cycles` now contains all the cycles in the input graph; it is shown in Figure 15.34.

**Figure 15.34** All Cycles in a Simple Directed Graph

<table>
<thead>
<tr>
<th>cycle</th>
<th>order</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>E</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>E</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>E</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>E</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>E</td>
</tr>
</tbody>
</table>
The six additional cycles are shown graphically in Figure 15.35 through Figure 15.37.

**Figure 15.35** Cycles

\[ A \rightarrow E \rightarrow B \rightarrow C \rightarrow A \]

\[ A \rightarrow E \rightarrow C \rightarrow A \]

**Figure 15.36** Cycles

\[ B \rightarrow C \rightarrow D \rightarrow E \rightarrow B \]

\[ B \rightarrow C \rightarrow D \rightarrow F \rightarrow E \rightarrow B \]
Linear Assignment (Matching)

The linear assignment problem (LAP) is a fundamental problem in combinatorial optimization that involves assigning workers to tasks at minimal costs. In graph theoretic terms, the LAP is equivalent to finding a minimum-weight matching in a weighted bipartite directed graph. In a bipartite graph, the nodes can be divided into two disjoint sets $S$ (workers) and $T$ (tasks) such that every link connects a node in $S$ to a node in $T$. That is, the node sets $S$ and $T$ are independent. The concept of assigning workers to tasks can be generalized to the assignment of any abstract object from one group to some abstract object from a second group.

The linear assignment problem can be formulated as an integer programming optimization problem. The form of the problem depends on the sizes of the two input sets, $S$ and $T$. Let $A$ represent the set of possible assignments between sets $S$ and $T$. In the bipartite graph, these assignments are the links. If $|S| \geq |T|$, then the following optimization problem is solved:

\[
\begin{align*}
\text{minimize} & \quad \sum_{(i,j) \in A} c_{ij} x_{ij} \\
\text{subject to} & \quad \sum_{(i,j) \in A} x_{ij} \leq 1 \quad i \in S \\
& \quad \sum_{(i,j) \in A} x_{ij} = 1 \quad j \in T \\
& \quad x_{ij} \in \{0, 1\} \quad (i, j) \in A
\end{align*}
\]

This model allows for some elements of set $S$ (workers) to go unassigned (if $|S| > |T|$).
If \(|S| < |T|\), then the following optimization problem is solved:

\[
\begin{align*}
\text{minimize} & \quad \sum_{(i,j) \in A} c_{ij} x_{ij} \\
\text{subject to} & \quad \sum_{(i,j) \in A} x_{ij} = 1 \quad i \in S \\
& \quad \sum_{(i,j) \in A} x_{ij} \leq 1 \quad j \in T \\
& \quad x_{ij} \in \{0, 1\} \quad (i, j) \in A
\end{align*}
\]

This model allows for some elements of set \(T\) (tasks) to go unassigned.

In the network solver, you can invoke the linear assignment problem solver by using the LINEAR_ASSIGNMENT option. The algorithm that the network solver uses for solving a LAP is based on augmentation of shortest paths (Jonker and Volgenant 1987). This algorithm can be applied as long as the graph is bipartite.

The resulting assignment (or matching) is contained in the set that is specified in the ASSIGNMENTS= suboption of the OUT= option.

For a detailed example, see “Example 15.3: Linear Assignment Problem for Minimizing Relay Times” on page 751.

---

**Minimum-Cost Network Flow**

The *minimum-cost network flow problem* (MCF) is a fundamental problem in network analysis that involves sending flow over a network at minimal cost. Let \(G = (N, A)\) be a directed graph. For each link \((i, j) \in A\), associate a cost per unit of flow, designated by \(c_{ij}\). The demand (or supply) at each node \(i \in N\) is designated as \(b_i\), where \(b_i \geq 0\) denotes a supply node and \(b_i < 0\) denotes a demand node. These values must be within \([b^l_i, b^u_i]\). Define decision variables \(x_{ij}\) that denote the amount of flow sent from node \(i\) to node \(j\). The amount of flow that can be sent across each link is bounded to be within \([l_{ij}, u_{ij}]\). The problem can be modeled as a linear programming problem as follows:

\[
\begin{align*}
\text{minimize} & \quad \sum_{(i,j) \in A} c_{ij} x_{ij} \\
\text{subject to} & \quad b^l_i \leq \sum_{(i,j) \in A} x_{ij} - \sum_{(j,i) \in A} x_{ji} \leq b^u_i \quad i \in N \\
& \quad l_{ij} \leq x_{ij} \leq u_{ij} \quad (i, j) \in A
\end{align*}
\]

When \(b_i = b^l_i = b^u_i\) for all nodes \(i \in N\), the problem is called a *standard network flow problem*. For these problems, the sum of the supplies and demands must be equal to 0 to ensure that a feasible solution exists.

In the network solver, you can invoke the minimum-cost network flow solver by using the MINCOSTFLOW option.

The algorithm that the network solver uses for solving MCF is a variant of the primal network simplex algorithm (Ahuja, Magnanti, and Orlin 1993). Sometimes the directed graph \(G\) is disconnected. In this case, the problem is first decomposed into its weakly connected components, and then each minimum-cost flow problem is solved separately.
The input for the network is the standard graph input, which is described in the section “Input Data for the Network Solver” on page 680. The MCF option uses the following suboptions of the LINKS= input option that specify link-indexed numeric arrays:

- The WEIGHT= suboption defines the link cost $c_{ij}$ per unit of flow. (The default is 0, but if the WEIGHT= suboption is not specified, then the default is 1.)
- The LOWER= suboption defines the link flow lower bound $l_{ij}$. (The default is 0.)
- The UPPER= suboption defines the link flow upper bound $u_{ij}$. (The default is $\infty$.)

The MCF option uses the following suboptions of the NODES= option. The parameter of each suboption is a numeric array that is positive for supply nodes and negative for demand nodes. To specify fixed node supplies and demands, use only the LOWER= suboption.

- The LOWER= suboption defines the node supply lower bounds. (The default is 0.)
- The UPPER= suboption defines the node supply upper bounds. (The default is $\infty$, but if the UPPER= option is not specified, then the upper bounds are set equal to the lower bounds).

The resulting optimal flow through the network is written to the link-indexed numeric array that is specified in the FLOW= suboption of the OUT= option in the SOLVE WITH NETWORK statement.

**Minimum Cost Network Flow for a Simple Directed Graph**

The following example demonstrates how to use the network simplex algorithm to find a minimum-cost flow in a directed graph. Consider the directed graph in Figure 15.38, which appears in Ahuja, Magnanti, and Orlin (1993).

**Figure 15.38** Minimum-Cost Network Flow Problem: Data

The directed graph $G$ can be represented by the following links data set LinkSetIn and nodes data set NodeSetIn:
data LinkSetIn;
   input from to weight upper;

datalines;
1 4 2 15
2 1 1 10
2 3 0 10
2 6 6 10
3 4 1 5
3 5 4 10
4 7 5 10
5 6 2 20
5 7 7 15
6 8 8 10
7 8 9 15
;

data NodeSetIn;
   input node supply;

datalines;
1 10
2 20
4 -5
7 -15
8 -10
;

You can use the following call to the network solver to find a minimum-cost flow:

proc optmodel;
   set <num,num> LINKS;
   num cost(LINKS);
   num upper(LINKS);
   read data LinkSetIn into LINKS={[from to] cost=weight upper;}
   set NODES = union {<i,j> in LINKS} {i,j};
   num supply(NODES) init 0;
   read data NodeSetIn into [node] supply;
   num flow(LINKS);

   solve with network /
      loglevel = moderate
      logfreq = 1
      graph_direction = directed
      links = (upper=upper weight=cost)
      nodes = (lower=supply)
      mcf
      out = (flow=flow)
   ;

   print flow;
   create data LinkSetOut from [from to] upper cost flow;
quit;
The progress of the procedure is shown in **Figure 15.39**.

**Figure 15.39**  Network Solver Log for Minimum-Cost Network Flow

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Primal Objective</th>
<th>Primal Infeasibility</th>
<th>Dual Infeasibility</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.00000000E+00</td>
<td>2.00000000E+01</td>
<td>8.90000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>0.00000000E+00</td>
<td>2.00000000E+01</td>
<td>8.90000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td>5.00000000E+00</td>
<td>1.50000000E+01</td>
<td>8.40000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>5.00000000E+00</td>
<td>1.50000000E+01</td>
<td>8.30000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>7.50000000E+01</td>
<td>1.50000000E+01</td>
<td>8.30000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>7.50000000E+01</td>
<td>1.50000000E+01</td>
<td>7.90000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>1.30000000E+02</td>
<td>1.00000000E+01</td>
<td>7.60000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>2.70000000E+02</td>
<td>0.00000000E+00</td>
<td>0.00000000E+00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: Processing the minimum-cost network flow problem used 0.00 (cpu: 0.00) seconds.
NOTE: The data set WORK.LINKSETOUT has 11 observations and 5 variables.

The optimal solution is displayed in **Figure 15.40**.

**Figure 15.40**  Minimum-Cost Network Flow Problem: Optimal Solution

<table>
<thead>
<tr>
<th>Obs</th>
<th>from</th>
<th>to</th>
<th>upper</th>
<th>cost</th>
<th>flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>15</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>5</td>
<td>10</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>6</td>
<td>20</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>7</td>
<td>15</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>8</td>
<td>15</td>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>
The optimal solution is represented graphically in Figure 15.41.

![Figure 15.41 Minimum-Cost Network Flow Problem: Optimal Solution](image)

**Minimum-Cost Network Flow with Flexible Supply and Demand**

Using the same directed graph shown in Figure 15.38, this example demonstrates a network that has a flexible supply and demand. Consider the following adjustments to the node bounds:

- Node 1 has an infinite supply, but it still requires at least 10 units to be sent.
- Node 4 is a throughput node that can now handle an infinite amount of demand.
- Node 8 has a flexible demand. It requires between 6 and 10 units.

You use the special missing values (.I) to represent infinity and (.M) to represent minus infinity. The adjusted node bounds can be represented by the following nodes data set:

```
data NodeSetIn;
  input node lower upper;
datalines;
  1 10 .I
  2 20 20
  4 .M -5
  7 -15 -15
  8 -10 -6
;
```

You can use the following call to PROC OPTMODEL to find a minimum-cost flow:

```
proc optmodel;
  set <num,num> LINKS;
  num cost(LINKS);
  num upper(LINKS);
  read data LinkSetIn into LINKS=[from to] cost=weight upper;
  set <num> NODES;
  num supply(NODES);
  num supplyUB(NODES); /* also demand lower bound, if negative */
  read data NodeSetIn into NODES=[node] supply=lower supplyUB=upper;
  num flow(LINKS);
```
solve with NETWORK /
  direction = directed
  links       = ( upper = upper weight = cost )
  nodes       = ( lower = supply upper = supplyUB )
  mcf
  out        = ( flow = flow )
;
print flow;
create data LinkSetOut from [from to] upper cost flow;
quit;

The progress of the procedure is shown in Figure 15.42.

**Figure 15.42** PROC OPTMODEL Log for Minimum-Cost Network Flow

NOTE: There were 11 observations read from the data set WORK.LINKSETIN.
NOTE: There were 5 observations read from the data set WORK.NODESETIN.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 11.
NOTE: The network solver is called.
NOTE: Processing the minimum-cost network flow problem.
NOTE: Objective = 226.
NOTE: Processing the minimum-cost network flow problem used 0.00 (cpu: 0.00) seconds.
NOTE: The data set WORK.LINKSETOUT has 11 observations and 5 variables.

The optimal solution is displayed in Figure 15.43.

**Figure 15.43** Minimum-Cost Network Flow Problem: Optimal Solution

<table>
<thead>
<tr>
<th>Obs</th>
<th>from</th>
<th>to</th>
<th>upper</th>
<th>cost</th>
<th>flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>15</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>5</td>
<td>10</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>6</td>
<td>20</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>7</td>
<td>15</td>
<td>7</td>
<td>5</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>8</td>
<td>15</td>
<td>9</td>
<td>0</td>
</tr>
</tbody>
</table>

The optimal solution is represented graphically in Figure 15.44.
Minimum Cut

A cut is a partition of the nodes of a graph into two disjoint subsets. The cut-set is the set of links whose from and to nodes are in different subsets of the partition. A minimum cut of an undirected graph is a cut whose cut-set has the smallest link metric, which is measured as follows: For an unweighted graph, the link metric is the number of links in the cut-set. For a weighted graph, the link metric is the sum of the link weights in the cut-set.

In the network solver, you can invoke the minimum cut algorithm by using the MINCUT= option. This algorithm can be used only on undirected graphs.

If the value of the MAXCUTS= suboption is greater than 1, then the algorithm can return more than one set of cuts. The resulting cuts can be described in terms of partitions of the nodes of the graph or the links in the cut-sets. The example in the next section illustrates several ways to manipulate the output from the minimum cut algorithm. The node partition is specified in the PARTITIONS= suboption of the OUT= option in the SOLVE WITH NETWORK statement. Each tuple in this set has a cut ID and a node. PROC OPTMODEL provides only the smaller of the two subsets that form each partition. You can use the DIFF set operator to get the complement of each partition. The cut-set is specified in the CUTSETS= suboption of the OUT= option. This set contains the cut ID and the corresponding list of links.

The network solver uses the Stoer-Wagner algorithm (Stoer and Wagner 1997) to compute the minimum cuts. This algorithm runs in time \( O(|N||A| + |N|^2 \log |N|) \).

Minimum Cut for a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 15.45.
The links data set can be represented as follows:

```sas
data LinkSetIn;
    input from to weight @@;
    datalines;
    1 2 2 1 5 3 2 3 3 2 5 2 2 6 2
    3 4 4 3 7 2 4 7 2 4 8 2 5 6 3
    6 7 1 7 8 3
;
```

*Figure 15.45* A Simple Undirected Graph
The following statements calculate minimum cuts in the graph and output the results in the data set `MinCut`:

```plaintext
proc optmodel;
    set<num,num> LINKS;
    num weight{LINKS};
    read data LinkSetIn into LINKS=[from to] weight;
    set<num> NODES = union {<i,j> in LINKS} {i,j};
    set<num,num> PARTITIONS;
    set<num,num,num> CUTSETS;

    solve with NETWORK / 
        loglevel = moderate 
        links = (weight=weight) 
        mincut = (maxcuts=3) 
        out = (partitions=PARTITIONS cutsets=CUTSETS)
    ;
    set CUTS = setof {<cut,i,j> in CUTSETS} cut;
    num minCutWeight {cut in CUTS} = sum {<(cut),i,j> in CUTSETS} weight[i,j];
    print minCutWeight;
    for { cut in CUTS }
        put "Cut ID: " cut 
            "Partition: " ( slice( <cut,*,> , PARTITIONS ) ) 
            "and " ( NODES diff slice( <cut,*,> , PARTITIONS ) ) 
            "Cut: " ( slice( <cut,*,*> , CUTSETS ) ) 
            "Weight: " minCutWeight[cut];

    create data MinCut from [mincut from to]=CUTSETS weight[from,to];
    num partition {cut in CUTS, node in NODES} =
        if <cut,node> in PARTITIONS then 0 else 1;
    create data NodeSetOut from [cut node]={CUTS, NODES} partition;
quit;
```

The progress of the procedure is shown in Figure 15.46.

**Figure 15.46** Network Solver Log for Minimum Cut

---

**NOTE:** There were 12 observations read from the data set WORK.LINKSETIN.

**NOTE:** The number of nodes in the input graph is 8.

**NOTE:** The number of links in the input graph is 12.

**NOTE:** The network solver is called.

**NOTE:** Processing the minimum-cut problem.

**NOTE:** The minimum-cut algorithm found 3 cuts.

**NOTE:** Objective = 4.

Cut ID: 1 Partition: (3,4,7,8) and (1,2,5,6) 
Cut: (<2,3>,<6,7>) Weight: 4

Cut ID: 2 Partition: (8) and (1,2,5,3,6,4,7) 
Cut: (<4,8>,<7,8>) Weight: 5

Cut ID: 3 Partition: (1) and (2,5,3,6,4,7,8) 
Cut: (<1,2>,<1,5>) Weight: 5

**NOTE:** The data set WORK.MINCUT has 6 observations and 4 variables.

**NOTE:** The data set WORK.NODESETOUT has 24 observations and 3 variables.
The data set NodeSetOut now contains the partition of the nodes for each cut, shown in Figure 15.47.

**Figure 15.47** Minimum Cut Node Partition

<table>
<thead>
<tr>
<th>cut</th>
<th>node</th>
<th>partition</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>1</td>
</tr>
</tbody>
</table>

The data set MinCut contains the links in the cut-sets for each cut. This data set is shown in Figure 15.48, which also shows each cut separately.

**Figure 15.48** Minimum Cut-sets

<table>
<thead>
<tr>
<th>mincut</th>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mincut=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>from</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

**mincut** 4
Minimum Spanning Tree

A *spanning tree* of a connected undirected graph is a subgraph that is a tree that connects all the nodes together. When weights have been assigned to the links, a *minimum spanning tree* (MST) is a spanning tree whose sum of link weights is less than or equal to the sum of link weights of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a *minimum spanning forest*, which is a union of minimum spanning trees of its connected components.

In the network solver, you can invoke the minimum spanning tree algorithm by using the MINSPANTR option. This algorithm can be used only on undirected graphs.

The resulting minimum spanning tree is contained in the set that is specified in the FOREST= suboption of the OUT= option in the SOLVE WITH NETWORK statement.

The network solver uses Kruskal’s algorithm (Kruskal 1956) to compute the minimum spanning tree. This algorithm runs in time $O(|A| \log |N|)$ and therefore should scale to very large graphs.

Minimum Spanning Tree for a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 15.49.
The links data set can be represented as follows:

```plaintext
data LinkSetIn;
  input from $ to $ weight @@;
datalines;
A B  7  A D  5  B C  8  B D  9  B E  7  
C E  5  D E 15  D F  6  E F  8  E G  9  
F G 11  H I  1  I J  3  H J  2  
;
```

The following statements calculate a minimum spanning forest and output the results in the data set MinSpanForest:

```plaintext
proc optmodel;
  set<str,str> LINKS;
  num weight{LINKS};
  read data LinkSetIn into LINKS=[from to] weight;
  set<str,str> FOREST;

  solve with NETWORK /
    links = (weight=weight)
    minspantree
    out = (forest=FOREST)
  ;

  put FOREST;
  create data MinSpanForest from [from to]=FOREST weight;
quit;
```

The data set MinSpanForest now contains the links that belong to a minimum spanning forest, which is shown in Figure 15.50.
The minimal cost links are shown in green in Figure 15.51.

For a more detailed example, see “Example 15.5: Minimum Spanning Tree for Computer Network Topology” on page 757.

**Shortest Path**

A *shortest path* between two nodes $u$ and $v$ in a graph is a path that starts at $u$ and ends at $v$ and has the lowest total link weight. The starting node is called the *source node*, and the ending node is called the *sink node*.

In the network solver, you can calculate shortest paths by using the `SHORTESTPATH=` option.

By default, the network solver finds shortest paths for all pairs. That is, it finds a shortest path for each possible combination of source and sink nodes. Alternatively, you can use the `SOURCE=` suboption to fix a particular source node and find shortest paths from the fixed source node to all possible sink nodes. Conversely, by using the `SINK=` suboption, you can fix a sink node and find shortest paths from all possible source nodes to the fixed sink node. By using both suboptions together, you can request one particular
shortest path for a specific source-sink pair. In addition, you can use the SOURCE= and SINK= suboptions to define a list of source-sink pairs to process. The following sections show examples of these suboptions.

Which algorithm the network solver uses to find shortest paths depends on the data. The algorithm and run-time complexity for each graph type is shown in Table 15.14.

<table>
<thead>
<tr>
<th>Graph Type</th>
<th>Algorithm</th>
<th>Complexity (per Source Node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unweighted</td>
<td>Breadth-first search</td>
<td>$O(</td>
</tr>
<tr>
<td>Weighted (nonnegative)</td>
<td>Dijkstra’s algorithm</td>
<td>$O(</td>
</tr>
<tr>
<td>Weighted (positive and negative allowed)</td>
<td>Bellman-Ford algorithm</td>
<td>$O(</td>
</tr>
</tbody>
</table>

Details for each algorithm can be found in Ahuja, Magnanti, and Orlin (1993).

For weighted graphs, the algorithm uses the parameter that is specified in the WEIGHT= suboption of the LINKS= option to evaluate a path’s total weight (cost).

## Outputs

The shortest path algorithm produces up to two outputs. The output set that you specify in the SPPATHS= suboption of the OUT= option contains the links of a shortest path for each source-sink pair combination. The output parameter that you specify in the SPWEIGHTS= suboption of the OUT= option contains the total weight for the shortest path for each source-sink pair combination.

### SPPATHS= Set

The SPPATHS= set contains the links present in the shortest path for each source-sink pair.

The individual links in this set always appear in the order that you provide. If you provide link $(u, v)$ and solve a shortest path problem on an undirected graph, and that path visits node $v$ before node $u$, then this set will contain link $(u, v)$, not link $(v, u)$, which is not part of the network. This approach simplifies indexing throughout your model.

For large graphs and a large requested number of source-sink pairs, this set can be extremely large. For extremely large graphs, generating the output can sometimes take longer than computing the shortest paths. For example, using the US road network data for the state of New York, the data contain a directed graph that has 264,346 nodes. Finding the shortest path for all pairs from only one source node results in 140,969,120 observations, which is a set of size 11 GB. Finding shortest paths for all pairs from all nodes would produce an enormous set.

The SPPATHS= set contains the following tuple members:

1. the source node of this shortest path
2. the sink node of this shortest path
3. for this source-sink pair, the order of this link in a shortest path
4. the tail node of this link in a shortest path
5. the head node of this link in a shortest path
**SPWEIGHTS= Array**
This array contains the total weight for the shortest path for each of the source-sink pairs.

**Shortest Paths for All Pairs**
This example illustrates the use of the shortest path algorithm for all source-sink pairs on the simple undirected graph $G$ that is shown in Figure 15.52.

**Figure 15.52** A Simple Undirected Graph $G$
The undirected graph $G$ can be represented by the following links data set, LinkSetIn:

```plaintext
data LinkSetIn;
    input from $ to $ weight @@;
datalines;
A B 3   A C 2   A D 6   A E 4   B D 5
B F 5   C E 1   D E 2   D F 1   E F 4
;
```

The following statements calculate shortest paths for all source-sink pairs:

```plaintext
proc optmodel;
    set <str,str> LINKS;
    num weight{LINKS};
    read data LinkSetIn into LINKS=[from to] weight;
    set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
    set NODES = union{<i,j> in LINKS} {i,j};
    num path_length{NODES, NODES};

    solve with NETWORK /
        links = (weight=weight)
        shortpath
        out = (sppaths=PATHS spweights=path_length)
    ;

    put PATHS;
    print path_length;
    create data ShortPathP from [source sink order from to]=PATHS
        weight[from,to];
    create data ShortPathW from [source sink]
        path_weight=path_length;
quit;
```

The data set ShortPathP contains the shortest paths and is shown in Figure 15.53.
The data set `ShortPathW` contains the path weight for the shortest paths of each source-sink pair and is shown in Figure 15.54.

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>order</th>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>1</td>
<td>A</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>2</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>3</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
<td>2</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>2</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>3</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>4</td>
<td>D</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>1</td>
<td>A</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>1</td>
<td>A</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>2</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>1</td>
<td>B</td>
<td>D</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>1</td>
<td>A</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>2</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>3</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>F</td>
<td>1</td>
<td>B</td>
<td>F</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>2</td>
<td>A</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>1</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>2</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
<td>1</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>1</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>2</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>3</td>
<td>D</td>
<td>F</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>order</th>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>A</td>
<td>1</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>2</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>3</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
<td>1</td>
<td>B</td>
<td>D</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
<td>1</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
<td>2</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
<td>1</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td>1</td>
<td>D</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>1</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>1</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
</tbody>
</table>
Shortest Paths for a Subset of Source-Sink Pairs

This section illustrates the use of the SOURCE= and SINK= suboptions and the shortest path algorithm to calculate shortest paths for a subset of source-sink pairs. If \( S \) denotes the nodes in the SOURCE= set and \( T \) denotes the nodes in the SINK= set, the network solver calculates all the source-sink pairs in the crossproduct of these two sets.

For example, the following statements calculate a shortest path for the four combinations of source-sink pairs in \( S \times T = \{A, C\} \times \{B, F\} \):

```plaintext
proc optmodel;
    set <str,str> LINKS;
    num weight{LINKS};
    read data LinkSetIn into LINKS=[from to] weight;
    set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
    set SOURCES = / A C /;
    set SINKS = / B F /;

    solve with NETWORK /
        links = (weight=weight)
        shortpath = (source=SOURCES sink=SINKS)
        out = (sppaths=PATHS)
    ;

    put PATHS;
    create data ShortPath from [source sink order from to]=PATHS weight[from,to];
quit;
```
Chapter 15: The Network Solver

The data set ShortPath contains the shortest paths and is shown in Figure 15.55.

![Figure 15.55](image)

**Shortest Paths for a Subset of Source-Sink Pairs**

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>order</th>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>1</td>
<td>A</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>2</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>3</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>4</td>
<td>D</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>2</td>
<td>A</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>1</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>2</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>3</td>
<td>D</td>
<td>F</td>
<td>1</td>
</tr>
</tbody>
</table>

**Shortest Paths for a Subset of Source or Sink Pairs**

This section illustrates the use of the shortest path algorithm to calculate shortest paths between a subset of source (or sink) nodes and all other sink (or source) nodes.

In this case, you designate the subset of source (or sink) nodes in the node set by specifying the SOURCE= (or SINK=) suboption. By specifying only one of the suboptions, you indicate that you want the network solver to calculate all pairs from a subset of source nodes (or to calculate all pairs to a subset of sink nodes).

For example, the following statements calculate all the shortest paths from nodes B and E:

```plaintext
proc optmodel;
  set <str,str> LINKS;
  num weight{LINKS};
  read data LinkSetIn into LINKS=[from to] weight;
  set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
  set SOURCES = / B E /;

  solve with NETWORK /
    links = (weight=weight)
    shortpath = (source=SOURCES)
    out = (sppaths=PATHS)
  ;

  put PATHS;
  create data ShortPath from [source sink order from to]=PATHS weight[from,to];
quit;
```

The data set ShortPath contains the shortest paths and is shown in Figure 15.56.
Conversely, the following statements calculate all the shortest paths to nodes B and E:

```sas
proc optmodel;
  set <str,str> LINKS;
  num weight{LINKS};
  read data LinkSetIn into LINKS=[from to] weight;
  set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
  set SINKS = / B E /;

  solve with NETWORK /
    links      = (weight=weight)
    shortpath  = (sink=SINKS)
    out        = (sppaths=PATHS)
  ;

  put PATHS;
  create data ShortPath from [source sink order from to]=PATHS weight[from,to];
quit;
```

The data set ShortPath contains the shortest paths and is shown in Figure 15.57.
This section illustrates the use of the shortest path algorithm to calculate shortest paths between one source-sink pair by using the SOURCE= and SINK= suboptions.

The following statements calculate a shortest path between node C and node F:

```plaintext
proc optmodel;
  set <str,str> LINKS;
  num weight{LINKS};
  read data LinkSetIn into LINKS=[from to] weight;
  set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
  set SOURCES = / C /;
  set SINKS = / F /;

  solve with NETWORK /
    links = (weight=weight)
    shortpath = (source=SOURCES sink=SINKS)
    out = (sppaths=PATHS)
  ;

  put PATHS;
  create data ShortPath from [source sink order from to]=PATHS weight[from,to];
quit;
```

The data set ShortPath contains this shortest path and is shown in Figure 15.58.
Figure 15.58  Shortest Paths for One Source-Sink Pair

The shortest path is shown graphically in Figure 15.59.

Figure 15.59  Shortest Path between Nodes C and F
Shortest Paths with Auxiliary Weight Calculation

This section illustrates the use of the shortest path algorithm, where auxiliary weights are used to calculate the shortest paths between all source-sink pairs.

Consider a links data set in which the auxiliary weight is a counter for each link:

```
data LinkSetIn;
  input from $ to $ weight count @@;
datalines;
A B 3 1  A C 2 1  A D 6 1  A E 4 1  B D 5 1
B F 5 1  C E 1 1  D E 2 1  D F 1 1  E F 4 1
;
```

The following statements calculate shortest paths for all source-sink pairs:

```
proc optmodel;
  set <str,str> LINKS;
  num weight{LINKS};
  num count{LINKS};
  read data LinkSetIn into LINKS=[from to] weight count;
  set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
  set NODES = union{<i,j> in LINKS} {i,j};
  num path_length{i in NODES, j in NODES: i ~= j};

  solve with NETWORK /
    links = (weight=weight)
    shortpath
    out = (sppaths=PATHS spweights=path_length)
  ;

  put PATHS;
  num path_weight2{source in NODES, sink in NODES: source ~= sink} =
    sum {<(source),(sink),order,from,to> in PATHS} count[from,to];
  print path_length path_weight2;
  create data ShortPathW from [source sink]
    path_weight=path_length path_weight2;
quit;
```

The data set ShortPathW contains the total path weight for shortest paths in each source-sink pair and is shown in Figure 15.60. Because the variable count in LinkSetIn has a value of 1 for all links, the value in the output data set variable path_weights2 contains the number of links in each shortest path.
The section “Getting Started: Network Solver” on page 660 shows an example of using the shortest path algorithm to minimize travel to and from work based on traffic conditions.

**Shortest Paths with Negative Link Weights**

This section illustrates the use of the shortest path algorithm on a simple directed graph $G$ with negative link weights, shown in Figure 15.61.
The following statements call PROC OPTMODEL and declare the directed graph $G$ by using set and array literals. For more information about literals, see the section “NUMBER, STRING, and SET Parameter Declarations” on page 292 in Chapter 10, “The OPTMODEL Procedure.”

```optmodel
proc optmodel;
    set LINKS = / <A B> <A C> <B C> <B D> <B E> <D B> <D C> <E D> /;
    num weight{LINKS} init [ -1 4 3 2 2 1 5 -3 ];

    set NODES = union{<i,j> in LINKS} {i,j};
    /* Use the type (in this case, STRING) of NODES but leave PATHS empty */
    set PATHS init {NODES,NODES,/0/,NODES,NODES:0};
    set SOURCE = /E/, SINK = /B/;

    solve with NETWORK /
        links = ( weight = weight )
        direction = directed
        shortpath = ( source = SOURCE sink = SINK )
        out = ( sppaths = PATHS )
    ;
    put "Path and Weight: " (setof{<s,t,i,u,v> in PATHS} <u,v,weight[u,v]> );
```

As shown in Figure 15.62, the network solver identifies a shortest path that has negative weights.
Transitive Closure

The transitive closure of a graph $G$ is a graph $G^T = (N, A^T)$ such that for all $i, j \in N$ there is a link $(i, j) \in A^T$ if and only if there exists a path from $i$ to $j$ in $G$.

The transitive closure of a graph can help you efficiently answer questions about reachability. Suppose you want to answer the question of whether you can get from node $i$ to node $j$ in the original graph $G$. Given the transitive closure $G^T$ of $G$, you can simply check for the existence of link $(i, j)$ to answer the question. Transitive closure has many applications, including speeding up the processing of structured query languages, which are often used in databases.

In the network solver, you can invoke the transitive closure algorithm by using the TRANSITIVECLOSURE option.

The results for the transitive closure algorithm are written to the set that is specified in the CLOSURE= suboption of the OUT= option.

The algorithm that the network solver uses to compute transitive closure is a sparse version of the Floyd-Warshall algorithm (Cormen, Leiserson, and Rivest 1990). This algorithm runs in time $O(|N|^3)$ and therefore might not scale to very large graphs.
Transitive Closure of a Simple Directed Graph

This example illustrates the use of the transitive closure algorithm on the simple directed graph $G$ that is shown in Figure 15.64.

**Figure 15.64 A Simple Directed Graph $G$**

The directed graph $G$ can be represented by the links data set LinkSetIn as follows:

```plaintext
data LinkSetIn;
  input from $ to $ @@;
  datalines;
  B C B D C B D A D C
;
```

The following statements calculate the transitive closure and output the results in the data set TransClosure:

```plaintext
proc optmodel;
    set<str,str> LINKS;
    read data LinkSetIn into LINKS=[from to];
    set<str,str> CAN_REACH;

    solve with NETWORK /
        graph_direction = directed
        links = ( include = LINKS )
        transcl
        out = ( closure = CAN_REACH )
    ;

    put CAN_REACH;
    create data TransClosure from [from to]=CAN_REACH;
quit;
```

The data set TransClosure contains the transitive closure of $G$ and is shown in Figure 15.65.
The transitive closure of $G$ is shown graphically in Figure 15.66.

For a more detailed example, see Example 15.6.
Traveling Salesman Problem

The traveling salesman problem (TSP) finds a minimum-cost tour in an undirected graph, $G$, that has a node set, $N$, and a link set, $A$. A path in a graph is a sequence of nodes, each of which has a link to the next node in the sequence. An elementary cycle is a path in which the start node and end node are the same and otherwise no node appears more than once in the sequence. A Hamiltonian cycle (or tour) is an elementary cycle that visits every node. In solving the TSP, then, the goal is to find a Hamiltonian cycle of minimum total cost, where the total cost is the sum of the costs of the links in the tour. Associated with each link $(i,j) \in A$ are a binary variable $x_{ij}$, which indicates whether link $x_{ij}$ is part of the tour, and a cost $c_{ij}$. Let $\delta(S) = \{(i,j) \in A \mid i \in S, j \notin S\}$. Then an integer linear programming formulation of the TSP (for an undirected graph $G$) is as follows:

$$\text{minimize} \quad \sum_{(i,j) \in A} c_{ij} x_{ij}$$

subject to

$$\sum_{(i,j) \in \delta(i)} x_{i,j} = 2 \quad i \in N \quad \text{(two_match)}$$

$$\sum_{(i,j) \in \delta(S)} x_{ij} \geq 2 \quad S \subseteq N, \quad 2 \leq |S| \leq |N| - 1 \quad \text{(subtour_elim)}$$

$$x_{ij} \in \{0, 1\} \quad (i,j) \in A$$

The equations (two_match) are the matching constraints, which ensure that each node has degree two in the subgraph. The inequalities (subtour_elim) are the subtour elimination constraints (SECs), which enforce connectivity.

For a directed graph, $G$, the same formulation and solution approach is used on an expanded graph $G'$, as described in Kumar and Li (1994). The network solver takes care of the construction of the expanded graph and returns the solution in terms of the original input graph.

In practical terms, you can think of the TSP in the context of a routing problem in which each node is a city and the links are roads that connect those cities. If you know the distance between each pair of cities, the goal is to find the shortest possible route that visits each city exactly once and returns to the starting city. The TSP has applications in planning, logistics, manufacturing, genomics, and many other areas.

In the network solver, you can invoke the traveling salesman problem solver by using the TSP= option.

The algorithm that the network solver uses for solving a TSP is based on a variant of the branch-and-cut process described in Applegate et al. (2006).

The resulting tour is represented in two ways: in the numeric array that is specified in the ORDER= suboption of the OUT= option, the tour is specified as a sequence of nodes; in the set that is specified in the TOUR= suboption of the OUT= option, the tour is specified as a list of links in the optimal tour.

**Traveling Salesman Problem Applied to a Simple Undirected Graph**

As a simple example, consider the weighted undirected graph in Figure 15.67.
You can represent the links data set as follows:

```plaintext
data LinkSetIn;
  input from $ to $ weight @@;
datalines;
  A B 1.0  A C 1.0  A D 1.5  B C 2.0  B D 4.0
  B E 3.0  C D 3.0  C F 3.0  C H 4.0  D E 1.5
  D F 3.0  D G 4.0  E F 1.0  E G 1.0  F G 2.0
  F H 4.0  H I 3.0  I J 1.0  C J 5.0  F J 3.0
  F I 1.0  H J 1.0;
```

The following statements calculate an optimal traveling salesman tour and output the results in the data sets TSPTour and NodeSetOut:

```plaintext
proc optmodel;
  set<str,str> EDGES;
  set<str> NODES = union{<i,j> in EDGES} {i,j};
  num weight{EDGES};
  read data LinkSetIn into EDGES=[from to] weight;
  num tsp_order{NODES};
  set<str,str> TOUR;
  solve with NETWORK /
    loglevel = moderate
    links = (weight=weight)
    tsp
    out = (order=tsp_order tour=TOUR);
```

Chapter 15: The Network Solver

```plaintext
put TOUR;
print {<i,j> in TOUR} weight;
print tsp_order;
create data NodeSetOut from [node] tsp_order;
create data TSPTour from [from to]=TOUR weight;
quit;
```

The progress of the procedure is shown in Figure 15.68.

**Figure 15.68** Network Solver Log: Optimal Traveling Salesman Tour of a Simple Undirected Graph

```
NOTE: There were 22 observations read from the data set WORK.LINKSETIN.
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 22.
NOTE: The network solver is called.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 16 using 0.00 (cpu: 0.02) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Branch and Cut algorithm is used.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sol</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>16.00000000</td>
<td>15.5005000</td>
<td>3.22%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>16.00000000</td>
<td>16.0000000</td>
<td>0.00%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Optimal.
NOTE: Objective = 16.
NOTE: Processing the traveling salesman problem used 0.01 (cpu: 0.02) seconds.

```

The data set NodeSetOut now contains a sequence of nodes in the optimal tour and is shown in Figure 15.69.

**Figure 15.69** Nodes in the Optimal Traveling Salesman Tour

```
<table>
<thead>
<tr>
<th>node tsp_order</th>
</tr>
</thead>
<tbody>
<tr>
<td>A     1</td>
</tr>
<tr>
<td>B     2</td>
</tr>
<tr>
<td>C     3</td>
</tr>
<tr>
<td>H     4</td>
</tr>
<tr>
<td>J     5</td>
</tr>
<tr>
<td>I     6</td>
</tr>
<tr>
<td>F     7</td>
</tr>
<tr>
<td>G     8</td>
</tr>
<tr>
<td>E     9</td>
</tr>
<tr>
<td>D     10</td>
</tr>
</tbody>
</table>
```

The data set TSPTour now contains the links in the optimal tour and is shown in Figure 15.70.
**Traveling Salesman Problem**

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>1.0</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>2.0</td>
</tr>
<tr>
<td>C</td>
<td>H</td>
<td>4.0</td>
</tr>
<tr>
<td>H</td>
<td>J</td>
<td>1.0</td>
</tr>
<tr>
<td>I</td>
<td>J</td>
<td>1.0</td>
</tr>
<tr>
<td>F</td>
<td>I</td>
<td>1.0</td>
</tr>
<tr>
<td>F</td>
<td>G</td>
<td>2.0</td>
</tr>
<tr>
<td>E</td>
<td>G</td>
<td>1.0</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
<td>1.5</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>1.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>16.0</strong></td>
</tr>
</tbody>
</table>

The minimum-cost links are shown in green in Figure 15.71.

**Traveling Salesman Problem Applied to a Simple Directed Graph**

As another simple example, consider the weighted directed graph in Figure 15.72. In this graph it might not be possible to travel directly between a pair of nodes in both directions, or the cost of traveling directly between two nodes might depend on the direction of travel.
You can represent the links data set as follows:

```plaintext
data LinkSetIn;
    input from $ to $ weight @@;
datalines;
A B 2.0  A C 1.0  A E 4.0
B A 1.0  B C 2.0  B D 1.0  B E 1.0
C B 2.0  C D 3.0
D A 1.0  D C 1.0  D E 2.0
E A 2.0  E D 1.0
;
```

The following statements, which are identical to those in the undirected example above except for the SOLVE statement clause DIRECTION=DIRECTED, calculate an optimal traveling salesman tour (on a directed graph) and output the results in the data sets TSPTour and NodeSetOut:

```plaintext
proc optmodel;
    set<str,str> EDGES;
    set<str> NODES = union{<i,j> in EDGES} {i,j};
    num weight{EDGES};
    read data LinkSetIn into EDGES=[from to] weight;
    num tsp_order{NODES};
    set<str,str> TOUR;

    solve with NETWORK /
        loglevel = moderate
        links = (weight=weight)
        direction = directed
        tsp
```
out = (order=tsp_order tour=TOUR);

put TOUR;
print {<i,j> in TOUR} weight;
print tsp_order;
create data NodeSetOut from [node] tsp_order;
create data TSPTour from [from to]=TOUR weight;
quit;

The progress of the procedure is shown in Figure 15.73.

**Figure 15.73** Network Solver Log: Optimal Traveling Salesman Tour of a Simple Directed Graph

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Solns</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>6.0000000</td>
<td>5.9001000</td>
<td>1.69%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>6.0000000</td>
<td>6.0000000</td>
<td>0.00%</td>
<td>0</td>
</tr>
</tbody>
</table>

The data set NodeSetOut now contains a sequence of nodes in the optimal tour and is shown in Figure 15.74.

**Figure 15.74** Nodes in the Optimal Traveling Salesman Tour

<table>
<thead>
<tr>
<th>node</th>
<th>tsp_order</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>5</td>
</tr>
</tbody>
</table>

The data set TSPTour now contains the links in the optimal tour and is shown in Figure 15.75.
Figure 15.75 Links in the Optimal Traveling Salesman Tour

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>1</td>
</tr>
</tbody>
</table>

The minimum-cost links are shown in green in Figure 15.76.

Figure 15.76 Optimal Traveling Salesman Tour

---

Macro Variable _OROPTMODEL_

PROC OPTMODEL always creates and initializes a SAS macro variable called _OROPTMODEL_, which contains a character string. After each PROC OPTMODEL run, you can examine this macro variable by specifying `%put &_OROPTMODEL_;` and check the execution of the most recently invoked solver from the value of the macro variable.

Each keyword and value pair in _OROPTMODEL_ also appears in two other places: the PROC OPTMODEL automatic arrays _OROPTMODEL_NUM_ and _OROPTMODEL_STR_; and the ODS tables ProblemSummary and SolutionSummary, which appear after a SOLVE statement, unless you set the PRINTLEVEL= option to NONE. You can use these variables to obtain details about the solution even if you do not specify an output destination in the OUT= option.

After the solver is called, the various keywords in the variable are interpreted as follows:
Macro Variable _OROPTMODEL_

**STATUS**
indicates the solver status at termination. It can take one of the following values:

- **OK** The solver terminated normally.
- **SYNTAX_ERROR** The use of syntax is incorrect.
- **DATA_ERROR** The input data is inconsistent.
- **OUT_OF_MEMORY** Insufficient memory was allocated to the procedure.
- **IO_ERROR** A problem in reading or writing of data has occurred.
- **SEMANTIC_ERROR** An evaluation error, such as an invalid operand type, has occurred.
- **ERROR** The status cannot be classified into any of the preceding categories.

**SOLUTION_STATUS**
indicates the solution status at termination. It can take one of the following values:

- **ABORT_NOSOL** The solver was stopped by the user and did not find a solution.
- **ABORT_SOL** The solver was stopped by the user but still found a solution.
- **BAD_PROBLEM_TYPE** The problem type is not supported by the solver.
- **CONDITIONAL_OPTIMAL** The optimality of the solution cannot be proven.
- **ERROR** The algorithm encountered an error.
- **FAIL_NOSOL** The solver stopped due to errors and did not find a solution.
- **FAIL_SOL** The solver stopped due to errors but still found a solution.
- **FAILED** The solver failed to converge, possibly due to numerical issues.
- **HEURISTIC_NOSOL** The solver used only heuristics and did not find a solution.
- **HEURISTIC_SOL** The solver used only heuristics and found a solution.
- **INFEASIBLE** The problem is infeasible.
- **INFEASIBLE_OR_UNBOUNDED** The problem is infeasible or unbounded.
- **INTERRUPTED** The solver was interrupted by the system or the user before completing its work.
- **ITERATION_LIMIT_REACHED** The solver reached the maximum number of iterations that is specified in the MAXITER= option.
- **NODE_LIM_NOSOL** The solver reached the maximum number of nodes specified in the MAXNODES= option and did not find a solution.
- **NODE_LIM_SOL** The solver reached the maximum number of nodes specified in the MAXNODES= option and found a solution.
- **NULL_GRAPH** The graph was null (it had 0 nodes) after PROC OPTMODEL processed the SUBGRAPH= option.
- **OK** The algorithm terminated normally.
**OPTIMAL**

The solution is optimal.

**OPTIMAL_AGAP**

The solution is optimal within the absolute gap that is specified in the **ABSOBJGAP** option.

**OPTIMAL_COND**

The solution is optimal, but some infeasibilities (primal, bound, or integer) exceed tolerances because of scaling.

**OPTIMAL_RGAP**

The solution is optimal within the relative gap that is specified in the **RELOBJGAP** option.

**OUTMEM_NOSOL**

The solver ran out of memory and either did not find a solution or failed to output the solution due to insufficient memory.

**OUTMEM_SOL**

The solver ran out of memory but still found a solution.

**SOLUTION_LIM**

The solver reached the maximum number of solutions specified in the **MAXCLIQUES**, **MAXCYCLES**, or **MAXSOLS** option.

**TARGET**

The solution is not worse than the target that is specified in the **TARGET** option.

**TIME_LIM_NOSOL**

The solver reached the execution time limit specified in the **MAXTIME** option and did not find a solution.

**TIME_LIM_SOL**

The solver reached the execution time limit specified in the **MAXTIME** option and found a solution.

**TIME_LIMIT_REACHED**

The solver reached its execution time limit.

**UNBOUNDED**

The problem is unbounded.

**PROBLEM_TYPE**

indicates the type of problem solved. It can take one of the following values:

**BICONCOMP**

Biconnected components

**CLIQUE**

Maximal cliques

**CONCOMP**

Connected components

**CYCLE**

Cycle detection

**LAP**

Linear assignment (matching)

**MCF**

Minimum-cost network flow

**MINCUT**

Minimum cut

**MST**

Minimum spanning tree

**SHORTPATH**

Shortest path

**TRANSCL**

Transitive closure

**TSP**

Traveling salesman

**NONE**

This value is used when you do not specify an algorithm to run.
OBJECTIVE
indicates the objective value that is obtained by the solver at termination. For problem classes that do not have an explicit objective, such as cycle, the value of this keyword within the _OROPTMODEL_ macro variable is missing (.).

RELATIVE_GAP
indicates the relative gap between the best integer objective (BestInteger) and the best bound on the objective function value (BestBound) upon termination of the MILP solver. The relative gap is equal to

\[
\frac{|\text{BestInteger} - \text{BestBound}|}{10 + |\text{BestBound}|}
\]

ABSOLUTE_GAP
indicates the absolute gap between the best integer objective (BestInteger) and the best bound on the objective function value (BestBound) upon termination of the MILP solver. The absolute gap is equal to \(|\text{BestInteger} - \text{BestBound}|\).

PRIMAL_INFEASIBILITY
indicates the maximum (absolute) violation of the primal constraints by the solution.

BOUND_INFEASIBILITY
indicates the maximum (absolute) violation by the solution of the lower or upper bounds (or both).

INTEGER_INFEASIBILITY
indicates the maximum (absolute) violation of the integrality of integer variables returned by the MILP solver.

BEST_BOUND
indicates the best bound on the objective function value at termination. A missing value indicates that the MILP solver was not able to obtain such a bound.

NODES
indicates the number of nodes enumerated by the MILP solver by using the branch-and-bound algorithm.

ITERATIONS
indicates the number of simplex iterations taken to solve the problem.

PRESOLVE_TIME
indicates the time (in seconds) used in preprocessing.

SOLUTION_TIME
indicates the time (in seconds) taken to solve the problem, including preprocessing time.

NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

When SOLUTION_STATUS has a value of OPTIMAL, CONDITIONAL_OPTIMAL, ITERATION_LIMIT_REACHED, or TIME_LIMIT_REACHED, all terms of the _OROPTMODEL_ macro variable are present; for other values of SOLUTION_STATUS, some terms do not appear.
The following keywords within the _OROPTMODEL_ macro variable appear only with certain algorithms. The keywords convey information about the number of solutions each algorithm found:

**NUM_ARTICULATION_POINTS**
indicates the number of articulation points found. This term appears only for biconnected components.

**NUM_CLIQUES**
indicates the number of cliques found. This term appears only for clique.

**NUM_COMPONENTS**
indicates the number of components that match the definitions of the corresponding problem class. This term appears only for connected components and biconnected components.

**NUM_CYCLES**
indicates the number of cycles found that satisfy the criteria you provide. This term appears only for cycles.

---

**Examples: Network Solver**

**Example 15.1: Articulation Points in a Terrorist Network**

This example considers the terrorist communications network from the attacks on the United States on September 11, 2001, described in Krebs 2002. Figure 15.77 shows this network, which was constructed after the attacks, based on collected intelligence information.
The full network data include 153 links. The following statements show a small subset to illustrate the use of the BICONNECTEDCOMPONENTS option in this context:

```
data LinkSetInTerror911;
    input from & $32. to & $32.;
    datalines;
    Abu Zubaida       Djamal Beghal
    Jean-Marc Grandvisir  Djamal Beghal
    Nizar Trabelsi      Djamal Beghal
    Abu Walid           Djamal Beghal
    Abu Qatada          Djamal Beghal
    Zacarias Moussaoui   Djamal Beghal
    Jerome Courtaillier  Djamal Beghal
    Kamel Daoudi        Djamal Beghal
    Abu Walid           Kamel Daoudi
    Abu Walid           Abu Qatada
    Kamel Daoudi        Zacarias Moussaoui
    Kamel Daoudi        Jerome Courtaillier
```

---

**Figure 15.77** Terrorist Communications Network from 9/11
Suppose that this communications network had been discovered before the attack on 9/11. If the investigators’ goal was to disrupt the flow of communication between different groups within the organization, then they would want to focus on the people who are articulation points in the network.

To find the articulation points, use the following statements:

```plaintext
proc optmodel;
  set<str,str> LINKS;
  read data LinkSetInTerror911 into LINKS=[from to];
  set NODES = union{<i,j> in LINKS} {i,j};
  set<str> ARTPOINTS;
  solve with NETWORK /
    links = (include=LINKS)
    biconcomp
    out = (artpoints=ARTPOINTS)
  ;
  put ARTPOINTS;
  create data ArtPoints from [node]=ARTPOINTS artpoint=1;
quit;
```

The data set ArtPoints contains members of the network who are articulation points. Focusing investigations on cutting off these particular members could have caused a great deal of disruption in the terrorists’ ability to communicate when formulating the attack.

**Output 15.1.1** Articulation Points of Terrorist Communications Network from 9/11

<table>
<thead>
<tr>
<th>node</th>
<th>artpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Djamal Beghal</td>
<td>1</td>
</tr>
<tr>
<td>Zacarias Moussaoui</td>
<td>1</td>
</tr>
<tr>
<td>Essid Sami Ben Khemais</td>
<td>1</td>
</tr>
<tr>
<td>Mohamed Atta</td>
<td>1</td>
</tr>
<tr>
<td>Mamoun Darkazanli</td>
<td>1</td>
</tr>
<tr>
<td>Nawaf Alhazmi</td>
<td>1</td>
</tr>
</tbody>
</table>

**Example 15.2: Cycle Detection for Kidney Donor Exchange**

This example looks at an application of cycle detection to help create a kidney donor exchange. Suppose someone needs a kidney transplant and a family member is willing to donate one. If the donor and recipient are incompatible (because of blood types, tissue mismatch, and so on), the transplant cannot happen. Now suppose two donor-recipient pairs \(i\) and \(j\) are in this situation, but donor \(i\) is compatible with recipient \(j\) and donor \(j\) is compatible with recipient \(i\). Then two transplants can take place in a two-way swap, shown graphically in Figure 15.78. More generally, an \(n\)-way swap can be performed involving \(n\) donors and \(n\) recipients (Willingham 2009).
To model this problem, define a directed graph as follows. Each node is an incompatible donor-recipient pair. Link \((i, j)\) exists if the donor from node \(i\) is compatible with the recipient from node \(j\), as shown in Figure 15.79. The link weight is a measure of the quality of the match. By introducing dummy links whose weight is 0, you can also include altruistic donors who have no recipients or recipients who have no donors. The idea is to find a maximum-weight node-disjoint union of directed cycles. You want the union to be node-disjoint so that no kidney is donated more than once, and you want cycles so that the donor from node \(i\) donates a kidney if and only if the recipient from node \(i\) receives a kidney.

Without any other constraints, the problem could be solved as a linear assignment problem, as described in the section “Linear Assignment (Matching)” on page 705. But doing so would allow arbitrarily long cycles in the solution. Because of practical considerations (such as travel) and to mitigate risk, each cycle must have no more than \(L\) links. The kidney exchange problem is to find a maximum-weight node-disjoint union of short directed cycles.

One way to solve this problem is to explicitly generate all cycles whose length is at most \(L\) and then solve a set packing problem. You can use PROC OPTMODEL to generate the cycles, formulate the set packing problem, call the mixed integer linear programming solver, and output the optimal solution.

The following DATA step sets up the problem, first creating a random graph on \(n\) nodes with link probability \(p\) and Uniform(0,1) weight:

```plaintext
/* create random graph on n nodes with arc probability p and uniform(0,1) weight */
%let n = 100;
%let p = 0.02;
data LinkSetIn;
call streaminit(1);
do from = 0 to &n - 1;
do to = 0 to &n - 1;
   if from eq to then continue;
   else if rand('UNIFORM') < &p then do;
      weight = rand('UNIFORM');
      output;
   end;
end;
end;
runtime;
```
The following statements declare parameters and then read the input data:

```plaintext
%let max_length = 10;
proc optmodel;
    /* declare index sets and parameters, and read data */
    set <num,num> ARCS;
    num weight {ARCS};
    read data LinkSetIn into ARCS=[from to] weight;
    set<num,num,num> ID_ORDER_NODE;
```

The following statements use the network solver to generate all cycles whose length is greater than or equal to 2 and less than or equal to 10:

```plaintext
/* generate all cycles with 2 <= length <= max_length */
solve with NETWORK /
    loglevel = moderate
    graph_direction = directed
    links = (include=ARCS)
    cycle = (maxcycles=all minlength=2 maxlength=&max_length)
    out = (cycles=ID_ORDER_NODE)
```

The network solver finds 395 cycles of the appropriate length, as shown in Output 15.2.1.

**Output 15.2.1** Cycles for Kidney Donor Exchange Network Solver Log

NOTE: There were 208 observations read from the data set WORK.LINKSETIN.
NOTE: The number of nodes in the input graph is 98.
NOTE: The number of links in the input graph is 208.
NOTE: The network solver is called.
NOTE: Processing cycle detection using 4 threads.
NOTE: Processing cycle detection using the build algorithm.
NOTE: The algorithm found 395 cycles.
NOTE: Processing cycle detection used 0.00 (cpu: 0.00) seconds.

From the resulting set ID_ORDER_NODE, use the following statements to convert to one tuple per cycle-arc combination:

```plaintext
/* extract <cid,from,to> triples from <cid,order,node> triples */
set <num,num,num> ID_FROM_TO init {};
num last init ., from, to;
for {<cid,order,node> in ID_ORDER_NODE} do;
    from = last;
    to = node;
    last = to;
    if order ne 1 then ID_FROM_TO = ID_FROM_TO union {<cid,from,to>};
end;
```
Given the set of cycles, you can now formulate a mixed integer linear program (MILP) to maximize the total cycle weight. Let $C$ be the set of cycles of appropriate length, $N_c$ be the set of nodes in cycle $c$, $A_c$ be the set of links in cycle $c$, and $w_{ij}$ be the link weight for link $(i, j)$. Define a binary decision variable $x_c$. Set $x_c$ to 1 if cycle $c$ is used in the solution; otherwise, set it to 0. Then, the following MILP defines the problem that you want to solve (to maximize the quality of the kidney exchange):

$$\text{maximize } \sum_{c \in C} \left( \sum_{(i,j) \in A_c} w_{ij} \right) x_c$$

subject to

$$\sum_{c \in C; i \in N_c} x_c \leq 1 \quad i \in N$$  \hspace{1cm} (incomp_pair)

$$x_c \in \{0, 1\} \quad c \in C$$

The constraint (incomp_pair) ensures that each node (incompatible pair) in the graph is intersected at most once. That is, a donor can donate a kidney only once. You can use PROC OPTMODEL to solve this mixed integer linear programming problem as follows:

```sql
/* solve set packing problem to find maximum weight node-disjoint union of short directed cycles */
set CYCLES = setof {<c,i,j> in ID_FROM_TO} c;
set ARCS_c {c in CYCLES} = setof {<(c),i,j> in ID_FROM_TO} <i,j>;
set NODES_c {c in CYCLES} = union {<i,j> in ARCS_c[c]} {i,j};
set NODES = union {c in CYCLES} NODES_c[c];
um num cycle_weight {c in CYCLES} = sum {<i,j> in ARCS_c[c]} weight[i,j];

/* UseCycle[c] = 1 if cycle c is used, 0 otherwise */
var UseCycle {CYCLES} binary;

/* declare objective */
max TotalWeight
   = sum {c in CYCLES} cycle_weight[c] * UseCycle[c];

/* each node appears in at most one cycle */
con node_packing {i in NODES}:
   sum {c in CYCLES: i in NODES_c[c]} UseCycle[c] <= 1;

/* call solver */
solve with milp;

/* output optimal solution */
create data Solution from [c]={c in CYCLES: UseCycle[c].sol > 0.5}
cycle_weight;
quit;
```

PROC OPTMODEL solves the problem by using the mixed integer linear programming solver. As shown in Output 15.2.2, it was able to find a total weight (quality level) of 24.85.
Chapter 15: The Network Solver

Output 15.2.2  Cycles for Kidney Donor Exchange MILP Solver Log

NOTE: Problem generation will use 4 threads.
NOTE: The problem has 395 variables (0 free, 0 fixed).
NOTE: The problem has 395 binary and 0 integer variables.
NOTE: The problem has 64 linear constraints (64 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 3431 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 125 variables and 30 constraints.
NOTE: The MILP presolver removed 1669 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 270 variables, 34 constraints, and 1762 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Soln</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>20.6710373</td>
<td>1147.4221881</td>
<td>98.20%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>20.6710373</td>
<td>25.4194215</td>
<td>18.68%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>20.6710373</td>
<td>25.2577476</td>
<td>18.16%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>20.6710373</td>
<td>25.0811160</td>
<td>17.58%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>20.6710373</td>
<td>24.9411612</td>
<td>17.12%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>24.8508554</td>
<td>24.8508554</td>
<td>0.00%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>4</td>
<td>24.8508554</td>
<td>24.8508554</td>
<td>0.00%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The MILP solver added 14 cuts with 1466 cut coefficients at the root.
NOTE: Objective = 24.850855395.
NOTE: The data set WORK.SOLUTION has 7 observations and 2 variables.
STATUS=OK ALGORITHM=B&C SOLUTION_STATUS=OPTIMAL OBJECTIVE=24.850855395
RELATIVE_GAP=0 ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=1.332268E-15
BOUND_INFEASIBILITY=1.332268E-15 INTEGER_INFEASIBILITY=1.332268E-15
BEST_BOUND=24.850855395 NODES=1 SOLUTIONS_FOUND=4 ITERATIONS=129
PRESOLVE_TIME=0.01 SOLUTION_TIME=0.03

The data set Solution, shown in Output 15.2.3, now contains the cycles that define the best exchange and their associated weight (quality).

Output 15.2.3  Maximum Quality Solution for Kidney Donor Exchange

<table>
<thead>
<tr>
<th>c</th>
<th>cycle_weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>4.35416</td>
</tr>
<tr>
<td>117</td>
<td>4.97483</td>
</tr>
<tr>
<td>213</td>
<td>4.34026</td>
</tr>
<tr>
<td>274</td>
<td>5.08435</td>
</tr>
<tr>
<td>289</td>
<td>1.72530</td>
</tr>
<tr>
<td>294</td>
<td>2.42954</td>
</tr>
<tr>
<td>388</td>
<td>1.94241</td>
</tr>
</tbody>
</table>
Example 15.3: Linear Assignment Problem for Minimizing Relay Times

A swimming coach needs to assign male and female swimmers to each leg of a medley relay team; each swimmer in the relay uses a different stroke (backstroke, breaststroke, butterfly, or freestyle). The swimmers’ best times for each stroke are stored in a SAS data set. The LINEARASSIGNMENT option evaluates the times and matches strokes and swimmers to find the lowest relay time.

The data are stored in matrix format, where the row identifier is the swimmer’s name (variable name) and each swimming stroke is a column (variables back, breast, fly, and free). In the following DATA step, the relay times are split into two categories, male (M) and female (F):

```sas
data RelayTimes;
  input name $ sex $ back breast fly free;
datalines;
Sue   F 35.1 36.7 28.3 36.1
Karen F 34.6 32.6 26.9 26.2
Jan   F 31.3 33.9 27.1 31.2
Andrea F 28.6 34.1 29.1 30.3
Carol F 32.9 32.2 26.6 24.0
Ellen F 27.8 32.5 27.8 27.0
Jim   M 26.3 27.6 23.5 22.4
Mike  M 29.0 24.0 27.9 25.4
Sam   M 27.2 33.8 25.2 24.1
Clayton M 27.0 29.2 23.0 21.9;
```

The following statements solve the linear assignment problem for both male and female relay teams:

```sas
proc contents data=RelayTimes
  out=stroke_data(rename=(name=stroke) where=(type=1)); run;
proc optmodel;
  set <str> STROKES;
  read data stroke_data into STROKES=[stroke];
  set <str> SWIMMERS;
  str sex {SWIMMERS};
  num time {SWIMMERS, STROKES};
  read data RelayTimes into SWIMMERS=[name] sex
    {stroke in STROKES} <time[name,stroke]=col(stroke)>
  set FEMALES = {i in SWIMMERS: sex[i] = 'F'};
  set FNODES = FEMALES union STROKES;
  set MALES = {i in SWIMMERS: sex[i] = 'M'};
  set MNODES = MALES union STROKES;
  set <str,str> PAIRS;
  solve with NETWORK /
    graph_direction = directed
    links = (weight=time)
    subgraph = (nodes=FNODES)
    lap
    out = (assignments=PAIRS);
  put PAIRS;
  create data LinearAssignF from [name assign]=PAIRS sex[name] cost=time;
```
solve with NETWORK /
  graph_direction = directed
  links = (weight=time)
  subgraph = (nodes=MNODES)
  lap
  out = (assignments=PAIRS)
;
put PAIRS;
create data LinearAssignM from [name assign]=PAIRS sex[name] cost=time;
quit;

The progress of the two SOLVE WITH NETWORK calls is shown in Output 15.3.1.

Output 15.3.1  Network Solver Log: Linear Assignment for Swim Times

NOTE: The data set WORK.STROKE_DATA has 4 observations and 41 variables.
NOTE: There were 4 observations read from the data set WORK.STROKE_DATA.
NOTE: There were 10 observations read from the data set WORK.RELAYTIMES.
NOTE: The SUBGRAPH= option filtered 16 elements from 'time.'
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 24.
NOTE: The network solver is called.
NOTE: Processing the linear assignment problem.
NOTE: Objective = 111.5.
NOTE: Processing the linear assignment problem used 0.00 (cpu: 0.00) seconds.
{<'Karen','breast'>,<'Jan','fly'>,<'Carol','free'>,<'Ellen','back'>}
NOTE: The data set WORK.LINEARASSIGNF has 4 observations and 4 variables.
NOTE: The SUBGRAPH= option filtered 24 elements from 'time.'
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 16.
NOTE: The network solver is called.
NOTE: Processing the linear assignment problem.
NOTE: Objective = 96.6.
NOTE: Processing the linear assignment problem used 0.00 (cpu: 0.00) seconds.
{<'Jim','free'>,<'Mike','breast'>,<'Sam','back'>,<'Clayton','fly'>}
NOTE: The data set WORK.LINEARASSIGNM has 4 observations and 4 variables.

The data sets LinearAssignF and LinearAssignM contain the optimal assignments. Note that in the case of the female data, there are more people (set $S$) than there are strokes (set $T$). Therefore, the solver allows for some members of $S$ to remain unassigned.
Example 15.4: Linear Assignment Problem, Sparse Format versus Dense Format

This example looks at the problem of assigning swimmers to legs of a medley relay based on their best stroke times. Certain swimmers are not eligible to perform certain strokes, because they do not excel at these strokes. A missing (.) value in the data matrix identifies an ineligible assignment. For example:

```sas
data RelayTimesMatrix;
  input name $ sex $ back breast fly free;
datalines;
Sue   F  .  36.7 28.3 36.1
Karen F 34.6  .   .  26.2
Jan   F 31.3  .   .  27.1
Andrea F 28.6  .  29.1  .
Carol F 32.9  .  26.6  .
;
```

The linear assignment problem can also be interpreted as the minimum-weight matching in a bipartite directed graph. The eligible assignments define links between the rows (swimmers) and the columns (strokes), as in Figure 15.80.
You can represent the same data in RelayTimesMatrix by using a links data set as follows:

```sas
data RelayTimesLinks;
  input name $ attr $ cost;
  datalines;
  Sue   breast 36.7
  Sue   fly     28.3
  Sue   free    36.1
  Karen back   34.6
  Karen free   26.2
  Jan    back   31.3
  Jan    fly     27.1
  Andrea back  28.6
  Andrea fly   29.1
  Carol   back  32.9
  Carol   fly    26.6
;
```

This graph must be bipartite (such that $S$ and $T$ are disjoint). If it is not, the network solver returns an error.

Now, you can use either input format to solve the same problem, as follows:
Example 15.4: Linear Assignment Problem, Sparse Format versus Dense Format

```plaintext
proc contents data=RelayTimesMatrix
  out=stroke_data(rename=(name=stroke) where=(type=1));
run;

proc optmodel;
  set <str> STROKES;
  read data stroke_data into STROKES=[stroke];
  set <str> SWIMMERS;
  str sex {SWIMMERS};
  num time {SWIMMERS, STROKES};
  read data RelayTimesMatrix into SWIMMERS=[name]
    sex
    {stroke in STROKES} <time[name,stroke]=col(stroke)>;
  set SWIMMERS_STROKES =
    {name in SWIMMERS, stroke in STROKES: time[name,stroke] ne .};
  set <str,str> PAIRS;
  solve with NETWORK /
    graph_direction = directed
    links = (weight=time)
    subgraph = (links=SWIMMERS_STROKES)
    lap
    out = (assignments=PAIRS)
  ;
  put PAIRS;
  create data LinearAssignMatrix from [name assign]=PAIRS
    sex[name] cost=time;
quit;

proc sql;
  create table stroke_data as
    select distinct attr as stroke
    from RelayTimesLinks;
quit;

proc optmodel;
  set <str> STROKES;
  read data stroke_data into STROKES=[stroke];
  set <str> SWIMMERS;
  str sex {SWIMMERS};
  set <str,str> SWIMMERS_STROKES;
  num time {SWIMMERS_STROKES};
  read data RelayTimesLinks into SWIMMERS_STROKES=[name attr] time=cost;
  set <str,str> PAIRS;
  solve with NETWORK /
    graph_direction = directed
    links = (weight=time)
    lap
    out = (assignments=PAIRS)
  ;
  put PAIRS;
  create data LinearAssignLinks from [name attr]=PAIRS cost=time;
quit;
```
The data sets LinearAssignMatrix and LinearAssignLinks now contain the optimal assignments, as shown in Output 15.4.1 and Output 15.4.2.

**Output 15.4.1** Optimal Assignments for Swim Times (Dense Input)

<table>
<thead>
<tr>
<th>name</th>
<th>assign</th>
<th>sex</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>breast</td>
<td>F</td>
<td>36.7</td>
</tr>
<tr>
<td>Karen</td>
<td>free</td>
<td>F</td>
<td>26.2</td>
</tr>
<tr>
<td>Andrea</td>
<td>back</td>
<td>F</td>
<td>28.6</td>
</tr>
<tr>
<td>Carol</td>
<td>fly</td>
<td>F</td>
<td>26.6</td>
</tr>
</tbody>
</table>

The optimal assignments are shown graphically in Figure 15.81.

**Output 15.4.2** Optimal Assignments for Swim Times (Sparse Input)

<table>
<thead>
<tr>
<th>name</th>
<th>attr</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>breast</td>
<td>36.7</td>
</tr>
<tr>
<td>Karen</td>
<td>free</td>
<td>26.2</td>
</tr>
<tr>
<td>Andrea</td>
<td>back</td>
<td>28.6</td>
</tr>
<tr>
<td>Carol</td>
<td>fly</td>
<td>26.6</td>
</tr>
</tbody>
</table>

The optimal assignments are shown graphically in Figure 15.81.
For large problems where a number of links are forbidden, the sparse format can be faster and can save a great deal of memory. Consider an example that uses the dense format with 15,000 columns ($|S| = 15,000$) and 4,000 rows ($|T| = 4,000$). To store the dense matrix in memory, the network solver needs to allocate approximately $|S| \times |T| \times 8/1,024/1,024 = 457$ MB. If the data have mostly ineligible links, then the sparse (graph) format is much more efficient with respect to memory. For example, if the data have only 5% of the eligible links ($15,000 \times 4,000 \times 0.05 = 3,000,000$), then the dense storage would still need 457 MB. The sparse storage for the same example needs approximately $|S| \times |T| \times 0.05 \times 12/1,024/1,024 = 34$ MB. If the problem is fully dense (all links are eligible), then the dense format is more efficient.

**Example 15.5: Minimum Spanning Tree for Computer Network Topology**

Consider the problem of designing a small network of computers in an office. In designing the network, the goal is to make sure that each machine in the office can reach every other machine. To accomplish this goal, Ethernet lines must be constructed and run between the machines. The construction costs for each possible link are based approximately on distance and are shown in Figure 15.82. Besides distance, the costs also reflect some restrictions due to physical boundaries. To connect all the machines in the office at minimal cost, you need to find a minimum spanning tree on the network of possible links.

**Figure 15.82 Potential Office Computer Network**

![Potential Office Computer Network](image)

Define the link data set as follows:

```plaintext
data LinkSetInCompNet;
  input from $ to $ weight @@;
datalines;
  A B 1.0 A C 1.0 A D 1.5 B C 2.0 B D 4.0
  B E 3.0 C D 3.0 C F 3.0 C H 4.0 D E 1.5
  D F 3.0 D G 4.0 E F 1.0 E G 1.0 F G 2.0
  F H 4.0 H I 1.0 I J 1.0
;
```

The following statements find a minimum spanning tree:
proc optmodel;
set<str,str> LINKS;
num weight{LINKS};
read data LinkSetInCompNet into LINKS=[from to] weight;
set<str,str> FOREST;
solve with NETWORK /
   links = (weight=weight)
   minspantree
   out = (forest=FOREST)
;
put FOREST;
put (sum {<i,j> in FOREST} weight[i,j]);
create data MinSpanTree from [from to]=FOREST weight;
quit;

Output 15.5.1 shows the resulting data set MinSpanTree, which is displayed graphically in Figure 15.83 with the minimal cost links shown in green.

Figure 15.83 Minimum Spanning Tree for Office Computer Network

Output 15.5.1 Minimum Spanning Tree of a Computer Network Topology

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>J</td>
<td>1.0</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>1.0</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>1.0</td>
</tr>
<tr>
<td>E</td>
<td>G</td>
<td>1.0</td>
</tr>
<tr>
<td>H</td>
<td>I</td>
<td>1.0</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>1.0</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
<td>1.5</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>1.5</td>
</tr>
<tr>
<td>C</td>
<td>H</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13.0</td>
</tr>
</tbody>
</table>
Example 15.6: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System

Most software bug tracking systems have some notion of duplicate bugs in which one bug is declared to be the same as another bug. If bug A is considered a duplicate (DUP) of bug B, then a fix for B would also fix A. You can represent the DUPs in a bug tracking system as a directed graph where you add a link \( A \rightarrow B \) if A is a DUP of B.

The bug tracking system needs to check for two situations when users declare a bug to be a DUP. The first situation is called a circular dependence. Consider bugs A, B, C, and D in the tracking system. The first user declares that A is a DUP of B and that C is a DUP of D. Then, a second user declares that B is a DUP of C, and a third user declares that D is a DUP of A. You now have a circular dependence, and no primary bug is defined on which the development team should focus. You can easily see this circular dependence in the graph representation, because \( A \rightarrow B \rightarrow C \rightarrow D \rightarrow A \). Finding such circular dependencies can be done using cycle detection, which is described in the section “Cycle” on page 699. However, the second situation that needs to be checked is more general. If a user declares that A is a DUP of B and another user declares that B is a DUP of C, this chain of duplicates is already an issue. The bug tracking system needs to provide one primary bug to which the rest of the bugs are duplicated. The existence of these chains can be identified by calculating the transitive closure of the directed graph that is defined by the DUP links.

Given the original directed graph \( G \) (defined by the DUP links) and its transitive closure \( G^T \), any link in \( G^T \) that is not in \( G \) exists because of one chain that is present in \( G \).

Consider the following data that define some duplicated bugs (called defects) in a small sample of the bug tracking system:

```sas
data DefectLinks;
  input defectId $ linkedDefect $ linkType $ when datetime16.;
  format when datetime16.;
datalines;
   D0096978 S0711218 DUPTO 20OCT10:00:00:00
   S0152674 S0153280 DUPTO 30MAY02:00:00:00
   S0153280 S0153307 DUPTO 30MAY02:00:00:00
   S0153307 S0152674 DUPTO 30MAY02:00:00:00
   S0162973 S0162978 DUPTO 29NOV10:16:13:16
   S0162978 S0165405 DUPTO 29NOV10:16:13:16
   S0325026 S0575748 DUPTO 01JUN10:00:00:00
   S0347945 S0346582 DUPTO 03MAR06:00:00:00
   S0350596 S0346582 DUPTO 21MAR06:00:00:00
   S0539744 S0643230 DUPTO 10MAY10:00:00:00
   S0575748 S0643230 DUPTO 15JUN10:00:00:00
   S0629984 S0643230 DUPTO 01JUN10:00:00:00
;
```

The following statements calculate cycles in addition to the transitive closure of the graph \( G \) that is defined by the duplicated defects in DefectLinks. The output data set Cycles contains any circular dependencies, and the data set TransClosure contains the transitive closure \( G^T \). To identify the chains, you can use PROC SQL to identify the links in \( G^T \) that are not in \( G \).
proc optmodel;
    set<str,str> LINKS;
    read data DefectLinks into LINKS=[defectId linkedDefect];
    set<num,num,str> CYCLES;
    set<str,str> CLOSURE;

    solve with NETWORK / 
        loglevel = moderate 
        graph_direction = directed 
        links = (include=LINKS) 
        cycle = (maxcycles=1) 
        out = (cycles=CYCLES) 
    ;

    put CYCLES;
    create data Cycles from [cycle order node]=CYCLES;

    solve with NETWORK / 
        loglevel = moderate 
        graph_direction = directed 
        links = (include=LINKS) 
        transitive_closure 
        out = (closure=CLOSURE) 
    ;

    put CLOSURE;
    create data TransClosure from [defectId linkedDefect]=CLOSURE;
quit;

proc sql;
    create table Chains as 
        select defectId, linkedDefect from TransClosure 
        except 
            select defectId, linkedDefect from DefectLinks;
quit;

The progress of the procedure is shown in Output 15.6.1.
Example 15.6: Transitive Closure for Identification of Circular Dependencies

**Output 15.6.1** Network Solver Log: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System

NOTE: There were 12 observations read from the data set WORK.DEFECTLINKS.
NOTE: The number of nodes in the input graph is 16.
NOTE: The number of links in the input graph is 12.
NOTE: The network solver is called.
NOTE: Processing cycle detection.
NOTE: Processing cycle detection using the backtrack algorithm.
NOTE: Required number of cycles found. You can increase this value using the MAXCYCLES= option.
NOTE: The algorithm found 1 cycles.
NOTE: Processing cycle detection used 0.00 (cpu: 0.00) seconds.
{<1,1,'S0152674'>,<1,2,'S0153280'>,<1,3,'S0153307'>,<1,4,'S0152674'>}
NOTE: The data set WORK.CYCLES has 4 observations and 3 variables.
NOTE: The number of nodes in the input graph is 16.
NOTE: The number of links in the input graph is 12.
NOTE: The network solver is called.
NOTE: Processing the transitive closure.
NOTE: Processing the transitive closure used 0.00 (cpu: 0.00) seconds.
{<D0096978','S0711218'>,<S0152674',S0153280'>,<S0153280',S0153307'>,<
'S0153307',S0152674'>,<S0162973',S0162978'>,<S0162978',S0165405'>,<
'S0325026',S0575748'>,<S0347945',S0346582'>,<S0350596',S0346582'>,<
'S0539744',S0643230'>,<S0575748',S0643230'>,<S0629984',S0643230'>,<
'S0153280',S0152674'>,<S0162973',S0165405'>,<S0325026',S0643230'>,<
'S0153307',S0153280'>,<S0153280',S0153307'>,<S0152674',S0152674'>,<
'S0152674',S0153307'>,<S0153307',S0153307'>}
NOTE: The data set WORK.TRANSCLUSION has 20 observations and 2 variables.
NOTE: Table WORK.CHAINS created, with 8 rows and 2 columns.
The data set Cycles contains one case of a circular dependence in which the DUPs start and end at S0152674.

**Output 15.6.2** Cycle in Bug Tracking System

<table>
<thead>
<tr>
<th>cycle</th>
<th>order</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>S0152674</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>S0153280</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>S0153307</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>S0152674</td>
</tr>
</tbody>
</table>

The data set Chains contains the chains in the bug tracking system that come from the links in $G^T$ that are not in $G$.

**Output 15.6.3** Chains in Bug Tracking System

<table>
<thead>
<tr>
<th>defectId</th>
<th>linkedDefect</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0152674</td>
<td>S0152674</td>
</tr>
<tr>
<td>S0152674</td>
<td>S0153307</td>
</tr>
<tr>
<td>S0153280</td>
<td>S0152674</td>
</tr>
<tr>
<td>S0153280</td>
<td>S0153280</td>
</tr>
<tr>
<td>S0153307</td>
<td>S0153280</td>
</tr>
<tr>
<td>S0153307</td>
<td>S0153307</td>
</tr>
<tr>
<td>S0162973</td>
<td>S0165405</td>
</tr>
<tr>
<td>S0325026</td>
<td>S0643230</td>
</tr>
</tbody>
</table>

**Example 15.7: Traveling Salesman Tour of US Capital Cities**

Consider a cross-country trip where you want to travel the fewest miles to visit all the capital cities in all US states (and the District of Columbia) except Alaska and Hawaii. Finding the optimal route is an instance of the traveling salesman problem, which is described in the section “Traveling Salesman Problem” on page 734.

The following data set provides a list of the capital cities and their latitude and longitude:

```r
data Cities;
  input city $ 20. lat long;
datalines;
Albany, NY 42.652552778 -73.75732222
Annapolis, MD 38.978611111 76.49111111
Atlanta, GA 33.749272222 -84.38826111
Augusta, ME 44.307236111 69.78167778
Baton Rouge, LA 30.457072222 -91.18740556
Bismarck, ND 46.820813889 100.7827417
Boise, ID 43.617697222 -116.1996139
Boston, MA 42.357708333 71.06356389
Carson City, NV 39.164075 -119.74055556
Charleston, WV 38.336388889 81.61222222
Cheyenne, WY 41.140277778 -104.8197222
Columbia, SC 34.000433333 81.03314722
Columbus, OH 39.961388889 -82.99888888
Concord, NH 43.206747222 71.53812778
```

```r
```
Example 15.7: Traveling Salesman Tour of US Capital Cities

<table>
<thead>
<tr>
<th>City</th>
<th>Latitude</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Denver, CO</td>
<td>39.7390944444</td>
<td>-104.9848972222</td>
</tr>
<tr>
<td>Des Moines, IA</td>
<td>41.5917777778</td>
<td>-93.6038694416</td>
</tr>
<tr>
<td>Dover, DE</td>
<td>39.1573055556</td>
<td>-75.5197222222</td>
</tr>
<tr>
<td>Frankfort, KY</td>
<td>38.1867777778</td>
<td>-84.8753333333</td>
</tr>
<tr>
<td>Harrisburg, PA</td>
<td>40.2644444444</td>
<td>-76.8666666667</td>
</tr>
<tr>
<td>Hartford, CT</td>
<td>41.7641361111</td>
<td>-72.6827777778</td>
</tr>
<tr>
<td>Helena, MT</td>
<td>46.5857</td>
<td>-112.0184</td>
</tr>
<tr>
<td>Indianapolis, IN</td>
<td>39.7686111111</td>
<td>-86.1625</td>
</tr>
<tr>
<td>Jackson, MS</td>
<td>32.303888889</td>
<td>-90.1822222222</td>
</tr>
<tr>
<td>Jefferson City, MO</td>
<td>38.5791194444</td>
<td>-92.1729916667</td>
</tr>
<tr>
<td>Lansing, MI</td>
<td>42.7337277778</td>
<td>-84.55558889</td>
</tr>
<tr>
<td>Lincoln, NE</td>
<td>40.808088889</td>
<td>-96.6995861111</td>
</tr>
<tr>
<td>Little Rock, AR</td>
<td>34.7467583333</td>
<td>-92.2887611111</td>
</tr>
<tr>
<td>Madison, WI</td>
<td>43.0744444444</td>
<td>-89.3847222222</td>
</tr>
<tr>
<td>Montgomery, AL</td>
<td>32.3774472222</td>
<td>-86.30094167</td>
</tr>
<tr>
<td>Montpelier, VT</td>
<td>44.2622222222</td>
<td>-72.5808333333</td>
</tr>
<tr>
<td>Nashville, TN</td>
<td>36.1658333333</td>
<td>-86.78416667</td>
</tr>
<tr>
<td>Oklahoma City, OK</td>
<td>35.4922805556</td>
<td>-97.503372222</td>
</tr>
<tr>
<td>Olympia, WA</td>
<td>47.0352777778</td>
<td>-122.90638889</td>
</tr>
<tr>
<td>Phoenix, AZ</td>
<td>33.4480972222</td>
<td>-112.09709444</td>
</tr>
<tr>
<td>Pierre, SD</td>
<td>44.3671666667</td>
<td>-100.3463528</td>
</tr>
<tr>
<td>Providence, RI</td>
<td>41.8308333333</td>
<td>-71.415</td>
</tr>
<tr>
<td>Raleigh, NC</td>
<td>35.7802777778</td>
<td>-78.63916667</td>
</tr>
<tr>
<td>Richmond, VA</td>
<td>37.5387583333</td>
<td>-77.43359444</td>
</tr>
<tr>
<td>Sacramento, CA</td>
<td>38.5765722222</td>
<td>-121.49341111</td>
</tr>
<tr>
<td>Saint Paul, MN</td>
<td>44.9551472222</td>
<td>-93.10223611</td>
</tr>
<tr>
<td>Salem, OR</td>
<td>44.9387305556</td>
<td>-123.0300972</td>
</tr>
<tr>
<td>Salt Lake City, UT</td>
<td>40.7772222222</td>
<td>-111.88805556</td>
</tr>
<tr>
<td>Santa Fe, NM</td>
<td>35.6822805556</td>
<td>-105.9396583</td>
</tr>
<tr>
<td>Springfield, IL</td>
<td>39.7985166667</td>
<td>-89.65488889</td>
</tr>
<tr>
<td>Tallahassee, FL</td>
<td>30.4381111111</td>
<td>-84.2816</td>
</tr>
<tr>
<td>Topeka, KS</td>
<td>39.0480083333</td>
<td>-95.67815556</td>
</tr>
<tr>
<td>Trenton, NJ</td>
<td>40.2204361111</td>
<td>-74.76990278</td>
</tr>
<tr>
<td>Washington, DC</td>
<td>38.8898027778</td>
<td>-77.00911389</td>
</tr>
</tbody>
</table>

From this list, you can generate a links data set, CitiesDist, that contains the distances (in miles) between each pair of cities. The distances are calculated by using the SAS function GEODIST.

```sas
/* create a list of all the possible pairs of cities */
proc sql;
create table CitiesDist as
select
   a.city as city1, a.lat as lat1, a.long as long1,
   b.city as city2, b.lat as lat2, b.long as long2,
   geodist(lat1, long1, lat2, long2, 'DM') as distance
from Cities as a, Cities as b
   where a.city < b.city;
quit;
```

The following PROC OPTMODEL statements find the optimal tour of all the capital cities:
/* find optimal tour by using the network solver */
proc optmodel;
  set<str,str> CAPPAIRS;
  set<str> CAPITALS = union {<i,j> in CAPPAIRS} {i,j};
  num distance{i in CAPITALS, j in CAPITALS: i < j};
  read data CitiesDist into CAPPAIRS=[city1 city2] distance;
  set<str,str> TOUR;
  num order{CAPITALS};

  solve with NETWORK /
    loglevel = moderate
    links = (weight=distance)
    tsp
    out = (order=order tour=TOUR)
  ;

  put (sum{<i,j> in TOUR} distance[i,j]);
/* create tour-ordered pairs (rather than input-ordered pairs) */
  str CAPbyOrder[1..card(CAPITALS)];
  for {i in CAPITALS} CAPbyOrder[order[i]] = i;
  set TSPEDGES init
    setof{i in 2..card(CAPITALS)} <CAPbyOrder[i-1],CAPbyOrder[i]>
    union {<CAPbyOrder[card(CAPITALS)],CAPbyOrder[1]>};
  num distance2{<i,j> in TSPEDGES} =
    if i < j then distance[i,j] else distance[j,i];
  create data TSPTourNodes from [node] tsp_order=order;
  create data TSPTourLinks from [city1 city2]=TSPEDGES distance=distance2;
quit;

The progress of the procedure is shown in Output 15.7.1. The total mileage that is needed to optimally tour the capital cities is 10,637.36 miles.
The following statements produce a graphical display of the solution:

```sql
/* merge latitude and longitude */
proc sql;
/* merge in the lat & long for city1 */
create table TSPTourLinksAnno1 as
select unique TSPTourLinks.*, cities.lat as lat1, cities.long as long1
from TSPTourLinks left join cities
on TSPTourLinks.city1=cities.city;
/* merge in the lat & long for city2 */
create table TSPTourLinksAnno2 as
select unique TSPTourLinksAnno1.*, cities.lat as lat2, cities.long as long2
from TSPTourLinksAnno1 left join cities
on TSPTourLinksAnno1.city2=cities.city;
quit;
```
The minimal-cost tour of the capital cities is shown in Figure 15.7.2.

The data set TSPTourLinks contains the links in the optimal tour. To display the links in the order in which they are to be visited, you can use the following DATA step:

```sas
/* create the directed optimal tour */
data TSPTourLinksDirected(drop=next);
  set TSPTourLinks;
  retain next;
  if _N_ ne 1 and city1 ne next then do;
    city2 = city1;
    city1 = next;
  end;
  next = city2;
run;
```
The data set TSPTourLinksDirected is shown in Figure 15.84.

### Figure 15.84
Links in the Optimal Traveling Salesman Tour

<table>
<thead>
<tr>
<th>city1</th>
<th>city2</th>
<th>distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Albany, NY</td>
<td>Harrisburg, PA</td>
<td>230.71</td>
</tr>
<tr>
<td>Harrisburg, PA</td>
<td>Charleston, WV</td>
<td>287.04</td>
</tr>
<tr>
<td>Charleston, WV</td>
<td>Columbus, OH</td>
<td>134.61</td>
</tr>
<tr>
<td>Columbus, OH</td>
<td>Lansing, MI</td>
<td>207.78</td>
</tr>
<tr>
<td>Lansing, MI</td>
<td>Madison, WI</td>
<td>246.20</td>
</tr>
<tr>
<td>Madison, WI</td>
<td>Saint Paul, MN</td>
<td>226.22</td>
</tr>
<tr>
<td>Saint Paul, MN</td>
<td>Bismarck, ND</td>
<td>392.09</td>
</tr>
<tr>
<td>Bismarck, ND</td>
<td>Pierre, SD</td>
<td>170.80</td>
</tr>
<tr>
<td>Pierre, SD</td>
<td>Cheyenne, WY</td>
<td>318.40</td>
</tr>
<tr>
<td>Cheyenne, WY</td>
<td>Denver, CO</td>
<td>97.09</td>
</tr>
<tr>
<td>Denver, CO</td>
<td>Salt Lake City, UT</td>
<td>371.84</td>
</tr>
<tr>
<td>Salt Lake City, UT</td>
<td>Helena, MT</td>
<td>401.15</td>
</tr>
<tr>
<td>Helena, MT</td>
<td>Boise, ID</td>
<td>289.50</td>
</tr>
<tr>
<td>Boise, ID</td>
<td>Olympia, WA</td>
<td>402.89</td>
</tr>
<tr>
<td>Olympia, WA</td>
<td>Salem, OR</td>
<td>144.95</td>
</tr>
<tr>
<td>Salem, OR</td>
<td>Sacramento, CA</td>
<td>446.28</td>
</tr>
<tr>
<td>Sacramento, CA</td>
<td>Carson City, NV</td>
<td>101.59</td>
</tr>
<tr>
<td>Carson City, NV</td>
<td>Phoenix, AZ</td>
<td>581.52</td>
</tr>
<tr>
<td>Phoenix, AZ</td>
<td>Santa Fe, NM</td>
<td>383.38</td>
</tr>
<tr>
<td>Santa Fe, NM</td>
<td>Oklahoma City, OK</td>
<td>475.23</td>
</tr>
<tr>
<td>Oklahoma City, OK</td>
<td>Austin, TX</td>
<td>359.89</td>
</tr>
<tr>
<td>Austin, TX</td>
<td>Baton Rouge, LA</td>
<td>391.67</td>
</tr>
<tr>
<td>Baton Rouge, LA</td>
<td>Jackson, MS</td>
<td>140.45</td>
</tr>
<tr>
<td>Jackson, MS</td>
<td>Little Rock, AR</td>
<td>207.71</td>
</tr>
<tr>
<td>Little Rock, AR</td>
<td>Jefferson City, MO</td>
<td>264.39</td>
</tr>
<tr>
<td>Jefferson City, MO</td>
<td>Topeka, KS</td>
<td>191.94</td>
</tr>
<tr>
<td>Topeka, KS</td>
<td>Lincoln, NE</td>
<td>133.03</td>
</tr>
<tr>
<td>Lincoln, NE</td>
<td>Des Moines, IA</td>
<td>170.19</td>
</tr>
<tr>
<td>Des Moines, IA</td>
<td>Springfield, IL</td>
<td>241.50</td>
</tr>
<tr>
<td>Springfield, IL</td>
<td>Indianapolis, IN</td>
<td>185.93</td>
</tr>
<tr>
<td>Indianapolis, IN</td>
<td>Frankfort, KY</td>
<td>129.29</td>
</tr>
<tr>
<td>Frankfort, KY</td>
<td>Nashville, TN</td>
<td>174.72</td>
</tr>
<tr>
<td>Nashville, TN</td>
<td>Atlanta, GA</td>
<td>215.06</td>
</tr>
<tr>
<td>Atlanta, GA</td>
<td>Montgomery, AL</td>
<td>145.82</td>
</tr>
<tr>
<td>Montgomery, AL</td>
<td>Tallahassee, FL</td>
<td>179.17</td>
</tr>
<tr>
<td>Tallahassee, FL</td>
<td>Columbia, SC</td>
<td>310.59</td>
</tr>
<tr>
<td>Columbia, SC</td>
<td>Raleigh, NC</td>
<td>183.17</td>
</tr>
<tr>
<td>Raleigh, NC</td>
<td>Richmond, VA</td>
<td>138.55</td>
</tr>
<tr>
<td>Richmond, VA</td>
<td>Washington, DC</td>
<td>96.03</td>
</tr>
<tr>
<td>Washington, DC</td>
<td>Annapolis, MD</td>
<td>28.58</td>
</tr>
<tr>
<td>Annapolis, MD</td>
<td>Dover, DE</td>
<td>53.68</td>
</tr>
<tr>
<td>Dover, DE</td>
<td>Trenton, NJ</td>
<td>83.54</td>
</tr>
<tr>
<td>Trenton, NJ</td>
<td>Hartford, CT</td>
<td>152.52</td>
</tr>
<tr>
<td>Hartford, CT</td>
<td>Providence, RI</td>
<td>65.65</td>
</tr>
<tr>
<td>Providence, RI</td>
<td>Boston, MA</td>
<td>40.61</td>
</tr>
<tr>
<td>Boston, MA</td>
<td>Concord, NH</td>
<td>63.39</td>
</tr>
<tr>
<td>Concord, NH</td>
<td>Augusta, ME</td>
<td>116.20</td>
</tr>
<tr>
<td>Augusta, ME</td>
<td>Montpelier, VT</td>
<td>138.89</td>
</tr>
<tr>
<td>Montpelier, VT</td>
<td>Albany, NY</td>
<td>125.92</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>city1</th>
<th>city2</th>
<th>distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>10,637.36</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### References


Chapter 16
The Nonlinear Programming Solver

Contents

Overview: NLP Solver ................................................. 769
Getting Started: NLP Solver ........................................ 771
Syntax: NLP Solver ..................................................... 781
  Functional Summary .............................................. 781
  NLP Solver Options ............................................... 782
Details: NLP Solver ................................................... 789
  Basic Definitions and Notation ................................. 789
  Constrained Optimization ..................................... 789
  Interior Point Algorithm ....................................... 791
  Active-Set Method ............................................... 792
  Multistart .......................................................... 794
  Covariance Matrix ............................................... 795
  Iteration Log for the Local Solver ............................ 798
  Iteration Log for Multistart .................................... 798
  Solver Termination Criterion .................................. 799
  Solver Termination Messages ................................. 800
  Macro Variable _OROPTMODEL_ ............................... 801
Examples: NLP Solver ................................................. 803
  Example 16.1: Solving Highly Nonlinear Optimization Problems 803
  Example 16.2: Solving Unconstrained and Bound-Constrained Optimization Problems 805
  Example 16.3: Solving NLP Problems with Range Constraints 807
  Example 16.4: Solving Large-Scale NLP Problems .................. 810
  Example 16.5: Solving NLP Problems That Have Several Local Minima 811
  Example 16.6: Maximum Likelihood Weibull Estimation .................. 818
  Example 16.7: Finding an Irreducible Infeasible Set .................. 820
References ....................................................... 824

Overview: NLP Solver

The sparse nonlinear programming (NLP) solver is a component of the OPTMODEL procedure that can solve optimization problems containing both nonlinear equality and inequality constraints. The general nonlinear
optimization problem can be defined as

\[
\begin{align*}
\text{minimize} & \quad f(x) \\
\text{subject to} & \quad h_i(x) = 0, \quad i \in \mathcal{E} = \{1, 2, \ldots, p\} \\
 & \quad g_i(x) \geq 0, \quad i \in \mathcal{I} = \{1, 2, \ldots, q\} \\
 & \quad l \leq x \leq u
\end{align*}
\]

where \( x \in \mathbb{R}^n \) is the vector of the decision variables; \( f : \mathbb{R}^n \mapsto \mathbb{R} \) is the objective function; \( h : \mathbb{R}^n \mapsto \mathbb{R}^p \) is the vector of equality constraints—that is, \( h = (h_1, \ldots, h_p) \); \( g : \mathbb{R}^n \mapsto \mathbb{R}^q \) is the vector of inequality constraints—that is, \( g = (g_1, \ldots, g_q) \); and \( l, u \in \mathbb{R}^n \) are the vectors of the lower and upper bounds, respectively, on the decision variables.

It is assumed that the functions \( f, h_i, \text{ and } g_i \) are twice continuously differentiable. Any point that satisfies the constraints of the NLP problem is called a feasible point, and the set of all those points forms the feasible region of the NLP problem—that is, \( \mathcal{F} = \{ x \in \mathbb{R}^n : h(x) = 0, \, g(x) \geq 0, \, l \leq x \leq u \} \).

The NLP problem can have a unique minimum or many different minima, depending on the type of functions involved. If the objective function is convex, the equality constraint functions are linear, and the inequality constraint functions are concave, then the NLP problem is called a convex program and has a unique minimum. All other types of NLP problems are called nonconvex and can contain more than one minimum, usually called local minima. The solution that achieves the lowest objective value of all local minima is called the global minimum or global solution of the NLP problem. The NLP solver can find the unique minimum of convex programs and a local minimum of a general NLP problem. In addition, the solver is equipped with specific options that enable it to locate the global minimum or a good approximation of it, for those problems that contain many local minima.

The NLP solver implements the following primal-dual methods for finding a local minimum:

- interior point trust-region line-search algorithm
- active-set trust-region line-search algorithm

Both methods can solve small-, medium-, and large-scale optimization problems efficiently and robustly. These methods use exact first and second derivatives to calculate search directions. The memory requirements of both algorithms are reduced dramatically because only nonzero elements of matrices are stored. Convergence of both algorithms is achieved by using a trust-region line-search framework that guides the iterations towards the optimal solution. If a trust-region subproblem fails to provide a suitable step of improvement, a line-search is then used to fine tune the trust-region radius and ensure sufficient decrease in objective function and constraint violations.

The interior point technique implements a primal-dual interior point algorithm in which barrier functions are used to ensure that the algorithm remains feasible with respect to the bound constraints. Interior point methods are extremely useful when the optimization problem contains many inequality constraints and you suspect that most of these constraints will be satisfied as strict inequalities at the optimal solution.

The active-set technique implements an active-set algorithm in which only the inequality constraints that are satisfied as equalities, together with the original equality constraints, are considered. Once that set of constraints is identified, active-set algorithms typically converge faster than interior point algorithms. They converge faster because the size and the complexity of the original optimization problem can be reduced if only few constraints need to be considered.
For optimization problems that contain many local optima, the NLP solver can be run in multistart mode. If the multistart mode is specified, the solver samples the feasible region and generates a number of starting points. Then the local solvers can be called from each of those starting points to converge to different local optima. The local minimum with the smallest objective value is then reported back to the user as the optimal solution.

The NLP solver implements many powerful features that are obtained from recent research in the field of nonlinear optimization algorithms (Akrotirianakis and Rustem 2005; Armand, Gilbert, and Jan-Jégou 2002; Erway, Gill, and Griffin 2007; Forsgren and Gill 1998; Vanderbei 1999; Wächter and Biegler 2006; Yamashita 1998). The term primal-dual means that the algorithm iteratively generates better approximations of the decision variables $x$ (usually called primal variables) in addition to the dual variables (also referred to as Lagrange multipliers). At every iteration, the algorithm uses a modified Newton’s method to solve a system of nonlinear equations. The modifications made to Newton’s method are implicitly controlled by the current trust-region radius. The solution of that system provides the direction and the steps along which the next approximation of the local minimum is searched. The active-set algorithm ensures that the primal iterations are always within their bounds—that is, $l \leq x^k \leq u$, for every iteration $k$. However, the interior approach relaxes this condition by using slack variables, and intermediate iterations might be infeasible.

Finally, for parameter estimation problems such as least squares, maximum likelihood, or Bayesian estimation problems, the NLP solver can calculate the covariance matrix after it successfully obtains parameter estimates.

**Getting Started: NLP Solver**

The NLP solver consists of two techniques that can solve a wide class of optimization problems efficiently and robustly. In this section two examples that introduce the two techniques of NLP are presented. The examples also introduce basic features of the modeling language of PROC OPTMODEL that is used to define the optimization problem.

The NLP solver can be invoked using the SOLVE statement,

```
SOLVE WITH NLP </options> ;
```

where options specify the technique name, termination criteria, and how to display the results in the iteration log. For a detailed description of the options, see the section “NLP Solver Options” on page 782.

**A Simple Problem**

Consider the following simple example of a nonlinear optimization problem:

\[
\begin{align*}
\text{minimize} & \quad f(x) = (x_1 + 3x_2 + x_3)^2 + 4(x_1 - x_2)^2 \\
\text{subject to} & \quad x_1 + x_2 + x_3 = 1 \\
& \quad 6x_2 + 4x_3 - x_1^3 - 3 \geq 0 \\
& \quad x_i \geq 0, i = 1, 2, 3
\end{align*}
\]

The problem consists of a quadratic objective function, a linear equality constraint, and a nonlinear inequality constraint. The goal is to find a local minimum, starting from the point $x^0 = (0.1, 0.7, 0.2)$. You can use the following call to PROC OPTMODEL to find a local minimum:
proc optmodel;
  var x{1..3} >= 0;

  con constr1: sum{i in 1..3}x[i] = 1;
  con constr2: 6*x[2] + 4*x[3] - x[1]**3 - 3 >= 0;

  /* starting point */
  x[1] = 0.1;
  x[2] = 0.7;
  x[3] = 0.2;

  solve with NLP;
  print x;
quit;

Because no options have been specified, the default solver (INTERIORPOINT) is used to solve the problem. The SAS output displays a detailed summary of the problem along with the status of the solver at termination, the total number of iterations required, and the value of the objective function at the best feasible solution that was found. The summaries and the returned solution are shown in Figure 16.1.

**Figure 16.1** Problem Summary, Solution Summary, and the Returned Solution

### The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objective Sense</strong></td>
</tr>
<tr>
<td><strong>Objective Function</strong></td>
</tr>
<tr>
<td><strong>Objective Type</strong></td>
</tr>
</tbody>
</table>

| Number of Variables | 3 |
| Bounded Above | 0 |
| Bounded Below | 3 |
| Bounded Below and Above | 0 |
| Free | 0 |
| Fixed | 0 |

| Number of Constraints | 2 |
| Linear LE (<=) | 0 |
| Linear EQ (=) | 1 |
| Linear GE (>=) | 0 |
| Linear Range | 0 |
| Nonlinear LE (<=) | 0 |
| Nonlinear EQ (=) | 0 |
| Nonlinear GE (>=) | 1 |
| Nonlinear Range | 0 |
The SAS log shown in Figure 16.2 displays a brief summary of the problem being solved, followed by the iterations that are generated by the solver.

**Figure 16.2 Progress of the Algorithm as Shown in the Log**

<table>
<thead>
<tr>
<th>Iter</th>
<th>Objective Value</th>
<th>Infeasibility</th>
<th>Optimality Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>7.20000000</td>
<td>0.00000000</td>
<td>0.0000000000</td>
</tr>
<tr>
<td>1</td>
<td>1.22115550</td>
<td>0.00004385</td>
<td>0.0005000000</td>
</tr>
<tr>
<td>2</td>
<td>1.00188693</td>
<td>0.00003290</td>
<td>0.00480263</td>
</tr>
<tr>
<td>3</td>
<td>1.00275609</td>
<td>0.00002123</td>
<td>0.000050000</td>
</tr>
<tr>
<td>4</td>
<td>1.00001702</td>
<td>0.00000000252254</td>
<td>0.0000187172</td>
</tr>
<tr>
<td>5</td>
<td>1.00001738</td>
<td>0.00000000250883</td>
<td>0.00000005000000</td>
</tr>
</tbody>
</table>

**NOTE:** Problem generation will use 4 threads.

**NOTE:** The problem has 3 variables (0 free, 0 fixed).

**NOTE:** The problem has 1 linear constraints (0 LE, 1 EQ, 0 GE, 0 range).

**NOTE:** The problem has 3 linear constraint coefficients.

**NOTE:** The problem has 1 nonlinear constraints (0 LE, 0 EQ, 1 GE, 0 range).

**NOTE:** The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0 nonlinear constraints.

**NOTE:** Using analytic derivatives for objective.

**NOTE:** Using analytic derivatives for nonlinear constraints.

**NOTE:** The NLP solver is called.

**NOTE:** The Interior Point algorithm is used.

**NOTE:** Optimal.

**NOTE:** Objective = 1.000017384.

**NOTE:** Objective of the best feasible solution found = 1.0000158715.

**NOTE:** The best feasible solution found is returned.

**NOTE:** To return the local optimal solution found, set the SOLTYPE= option to 0.
A Larger Optimization Problem

Consider the following larger optimization problem:

\[
\begin{align*}
\text{minimize} & \quad f(x) = \sum_{i=1}^{1000} x_i y_i + \frac{1}{2} \sum_{j=1}^{5} z_j^2 \\
\text{subject to} & \quad x_k + y_k + \sum_{j=1}^{5} z_j = 5, \text{ for } k = 1, 2, \ldots, 1000 \\
& \quad \sum_{i=1}^{1000}(x_i + y_i) + \sum_{j=1}^{5} z_j \geq 6 \\
& \quad -1 \leq x_i \leq 1, i = 1, 2, \ldots, 1000 \\
& \quad -1 \leq y_i \leq 1, i = 1, 2, \ldots, 1000 \\
& \quad 0 \leq z_i \leq 2, i = 1, 2, \ldots, 5
\end{align*}
\]

The problem consists of a quadratic objective function, 1,000 linear equality constraints, and a linear inequality constraint. There are also 2,005 variables. The goal is to find a local minimum by using the ACTIVESET technique. This can be accomplished by issuing the following call to PROC OPTMODEL:

```sas
proc optmodel;
    number n = 1000;
    number b = 5;
    var x{1..n} >= -1 <= 1 init 0.99;
    var y{1..n} >= -1 <= 1 init -0.99;
    var z{1..b} >= 0 <= 2 init 0.5;
    minimize f = sum{i in 1..n} x[i]*y[i] + 0.5*sum{j in 1..b} z[j]^2;
    con cons1{k in 1..n}: x[k] + y[k] + sum{j in 1..b} z[j] = b;
    con cons2: sum{i in 1..n} (x[i] + y[i]) + sum{j in 1..b} z[j] >= b + 1;
    solve with NLP / algorithm=activeset logfreq=2;
quit;
```

The SAS output displays a detailed summary of the problem along with the status of the solver at termination, the total number of iterations required, and the value of the objective function at the local minimum. The summaries are shown in Figure 16.3.
Figure 16.3 Problem Summary and Solution Summary

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
<tr>
<td>Number of Variables</td>
</tr>
<tr>
<td>Bounded Above</td>
</tr>
<tr>
<td>Bounded Below</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
</tr>
<tr>
<td>Free</td>
</tr>
<tr>
<td>Fixed</td>
</tr>
<tr>
<td>Number of Constraints</td>
</tr>
<tr>
<td>Linear LE (≤)</td>
</tr>
<tr>
<td>Linear EQ (=)</td>
</tr>
<tr>
<td>Linear GE (≥)</td>
</tr>
<tr>
<td>Linear Range</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Optimality Error</td>
</tr>
<tr>
<td>Infeasibility</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

The SAS log shown in Figure 16.4 displays a brief summary of the problem that is being solved, followed by the iterations that are generated by the solver.
An Optimization Problem with Many Local Minima

Consider the following optimization problem:

\[
\begin{align*}
\text{minimize } \quad f(x) &= e^{\sin(50x)} + \sin(60e^y) + \sin(70\sin(x)) + \sin(80\sin(y)) \\
&\quad - \sin(10(x+y)) + (x^2 + y^2)/4 \\
\text{subject to } \\
&\quad -1 \leq x \leq 1 \\
&\quad -1 \leq y \leq 1
\end{align*}
\]

The objective function is highly nonlinear and contains many local minima. The NLP solver provides you with the option of searching the feasible region and identifying local minima of better quality. This is achieved by writing the following SAS program:

```sas
proc optmodel;
var x >= -1 <= 1;
var y >= -1 <= 1;
min f = exp(sin(50*x)) + sin(60*exp(y)) + sin(70*sin(x)) + sin(sin(80*y)) \\
    - sin(10*(x+y)) + (x^2 + y^2)/4;
solve with nlp / multistart=(maxstarts=30) seed=94245;
quit;
```

The `MULTISTART=()` option is specified, which directs the algorithm to start the local solver from many different starting points. The SAS log is shown in Figure 16.5.
Figure 16.5  Progress of the Algorithm as Shown in the Log

NOTE: Problem generation will use 4 threads.
NOTE: The problem has 2 variables (0 free, 0 fixed).
NOTE: The problem has 0 linear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0 nonlinear constraints.
NOTE: Using analytic derivatives for objective.
NOTE: The NLP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The MULTISTART option is enabled.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Multistart algorithm is executing in single-machine mode.
NOTE: The Multistart algorithm is using up to 4 threads.
NOTE: Random number seed 94245 is used.

<table>
<thead>
<tr>
<th>Best Start</th>
<th>Objective</th>
<th>Objective</th>
<th>Local Optimality</th>
<th>Infesiti-</th>
<th>Local</th>
<th>Local</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-1.8567821</td>
<td>-1.8567821</td>
<td>SE-7</td>
<td>0</td>
<td>3</td>
<td>Optimal</td>
</tr>
<tr>
<td>2</td>
<td>-2.9390086</td>
<td>-2.9390086</td>
<td>SE-7</td>
<td>0</td>
<td>3</td>
<td>Optimal</td>
</tr>
<tr>
<td>3</td>
<td>-3.3068686</td>
<td>-3.3068686</td>
<td>SE-7</td>
<td>0</td>
<td>3</td>
<td>Optimal</td>
</tr>
<tr>
<td>4</td>
<td>-3.3068686</td>
<td>-2.6561783</td>
<td>SE-7</td>
<td>0</td>
<td>3</td>
<td>Optimal</td>
</tr>
<tr>
<td>5</td>
<td>-3.3068686</td>
<td>-1.5173191</td>
<td>SE-7</td>
<td>0</td>
<td>4</td>
<td>Optimal</td>
</tr>
<tr>
<td>6</td>
<td>-3.3068686</td>
<td>-1.9050116</td>
<td>SE-7</td>
<td>0</td>
<td>5</td>
<td>Optimal</td>
</tr>
<tr>
<td>7</td>
<td>-3.3068686</td>
<td>-1.9014527</td>
<td>6.14439E-7</td>
<td>6.14439E-7</td>
<td>4</td>
<td>Optimal</td>
</tr>
<tr>
<td>8</td>
<td>-3.3068686</td>
<td>-2.2355863</td>
<td>SE-7</td>
<td>0</td>
<td>3</td>
<td>Optimal</td>
</tr>
<tr>
<td>9</td>
<td>-3.3068686</td>
<td>-2.1930943</td>
<td>SE-7</td>
<td>0</td>
<td>5</td>
<td>Optimal</td>
</tr>
<tr>
<td>10</td>
<td>-3.3068686</td>
<td>-0.6723829</td>
<td>SE-7</td>
<td>0</td>
<td>4</td>
<td>Optimal</td>
</tr>
<tr>
<td>11</td>
<td>-3.3068686</td>
<td>-0.4033346</td>
<td>SE-7</td>
<td>0</td>
<td>3</td>
<td>Optimal</td>
</tr>
<tr>
<td>12</td>
<td>-3.3068686</td>
<td>-2.9525781</td>
<td>SE-7</td>
<td>0</td>
<td>4</td>
<td>Optimal</td>
</tr>
<tr>
<td>13</td>
<td>-3.3068686</td>
<td>-2.0402058</td>
<td>SE-7</td>
<td>0</td>
<td>4</td>
<td>Optimal</td>
</tr>
<tr>
<td>14</td>
<td>-3.3068686</td>
<td>-0.7857749</td>
<td>SE-7</td>
<td>0</td>
<td>5</td>
<td>Optimal</td>
</tr>
<tr>
<td>15 *</td>
<td>-3.3068686</td>
<td>-1.3289057</td>
<td>SE-7</td>
<td>0</td>
<td>4</td>
<td>Optimal</td>
</tr>
<tr>
<td>16</td>
<td>-3.3068686</td>
<td>-1.404132</td>
<td>SE-7</td>
<td>0</td>
<td>3</td>
<td>Optimal</td>
</tr>
<tr>
<td>17</td>
<td>-3.3068686</td>
<td>-2.4632393</td>
<td>SE-7</td>
<td>0</td>
<td>3</td>
<td>Optimal</td>
</tr>
<tr>
<td>18</td>
<td>-3.3068686</td>
<td>-2.4541355</td>
<td>SE-7</td>
<td>0</td>
<td>4</td>
<td>Optimal</td>
</tr>
<tr>
<td>19</td>
<td>-3.3068686</td>
<td>-1.3559281</td>
<td>8.74345E-7</td>
<td>8.74345E-7</td>
<td>3</td>
<td>Optimal</td>
</tr>
<tr>
<td>20</td>
<td>-3.3068686</td>
<td>-2.5171432</td>
<td>SE-7</td>
<td>0</td>
<td>5</td>
<td>Optimal</td>
</tr>
<tr>
<td>21</td>
<td>-3.3068686</td>
<td>-2.076195</td>
<td>SE-7</td>
<td>0</td>
<td>4</td>
<td>Optimal</td>
</tr>
<tr>
<td>22</td>
<td>-3.3068686</td>
<td>-2.387082</td>
<td>SE-7</td>
<td>0</td>
<td>4</td>
<td>Optimal</td>
</tr>
<tr>
<td>23</td>
<td>-3.3068686</td>
<td>-1.2265443</td>
<td>SE-7</td>
<td>0</td>
<td>4</td>
<td>Optimal</td>
</tr>
<tr>
<td>24</td>
<td>-3.3068686</td>
<td>-0.4808983</td>
<td>SE-7</td>
<td>0</td>
<td>3</td>
<td>Optimal</td>
</tr>
<tr>
<td>25</td>
<td>-3.3068686</td>
<td>-3.3068686</td>
<td>SE-7</td>
<td>0</td>
<td>4</td>
<td>Optimal</td>
</tr>
<tr>
<td>26</td>
<td>-3.3068686</td>
<td>-0.5926643</td>
<td>SE-7</td>
<td>0</td>
<td>4</td>
<td>Optimal</td>
</tr>
<tr>
<td>27</td>
<td>-3.3068686</td>
<td>-0.559164</td>
<td>SE-7</td>
<td>0</td>
<td>3</td>
<td>Optimal</td>
</tr>
<tr>
<td>28</td>
<td>-3.3068686</td>
<td>-2.8503167</td>
<td>SE-7</td>
<td>0</td>
<td>4</td>
<td>Optimal</td>
</tr>
<tr>
<td>29</td>
<td>-3.3068686</td>
<td>-2.7726003</td>
<td>SE-7</td>
<td>0</td>
<td>3</td>
<td>Optimal</td>
</tr>
<tr>
<td>30</td>
<td>-3.3068686</td>
<td>-2.311349</td>
<td>SE-7</td>
<td>0</td>
<td>3</td>
<td>Optimal</td>
</tr>
</tbody>
</table>

NOTE: The Multistart algorithm generated 480 sample points.
NOTE: 29 distinct local optima were found.
NOTE: The best objective value found by local solver = -3.306868647.
The SAS log presents additional information when the MULTISTART=() option is specified. The first column counts the number of restarts of the local solver. The second column records the best local optimum that has been found so far, and the third through sixth columns record the local optimum to which the solver has converged. The final column records the status of the local solver at every iteration.

The SAS output is shown in Figure 16.6.

**Figure 16.6 Problem Summary and Solution Summary**

**The OPTMODEL Procedure**

<table>
<thead>
<tr>
<th>Problem Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
<td>Minimization</td>
</tr>
<tr>
<td>Objective Function</td>
<td>f</td>
</tr>
<tr>
<td>Objective Type</td>
<td>Nonlinear</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Variables</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded Above</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
<td>2</td>
</tr>
<tr>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>Fixed</td>
<td>0</td>
</tr>
</tbody>
</table>

| Number of Constraints | 0 |

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>Multistart NLP</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Interior Point</td>
</tr>
<tr>
<td>Objective Function</td>
<td>f</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Optimal</td>
</tr>
<tr>
<td>Objective Value</td>
<td>-3.306868647</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Starts</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Sample Points</td>
<td>480</td>
</tr>
<tr>
<td>Number of Distinct Optima</td>
<td>29</td>
</tr>
<tr>
<td>Random Seed Used</td>
<td>94245</td>
</tr>
<tr>
<td>Optimality Error</td>
<td>5E-7</td>
</tr>
<tr>
<td>Infeasibility</td>
<td>0</td>
</tr>
</tbody>
</table>

| Presolve Time     | 0.00 |
| Solution Time     | 0.01 |
A Least Squares Estimation Problem for a Regression Model

The following data are used to build a regression model:

```
data samples;
    input x1 x2 y;
datalines;
   4  8   43.71
  62  5   351.29
  81 62   2878.91
  85 75   3591.59
  65 54   2058.71
  96 84   4487.87
  98 29   1773.52
  36 33   767.57
  30 91   1637.66
   3 59   215.28
  62 57   2067.42
  11 48   394.11
  66 21   932.84
  68 24   1069.21
  95 30   1770.78
  34 14   368.51
  86 81   3902.27
  37 49   1115.67
  46 80   2136.92
  87 72   3537.84
;
```

Suppose you want to compute the parameters in your regression model based on the preceding data, and the model is

$$L(a, b, c) = a \cdot x_1 + b \cdot x_2 + c \cdot x_1 \cdot x_2$$

where \( a, b, c \) are the parameters that need to be found.

The following PROC OPTMODEL call specifies the least squares problem for the regression model:

```
/* Regression model with interactive term: \( y = a \cdot x_1 + b \cdot x_2 + c \cdot x_1 \cdot x_2 \) */
proc optmodel;
    set obs;
    num x1{obs}, x2{obs}, y{obs};
    num mycov{i in 1.._nvar_, j in 1..i};
    var a, b, c;
    read data samples into obs=[_n_] x1 x2 y;
    impvar Err{i in obs} = y[i] - (a*x1[i]+b*x2[i]+c*x1[i]*x2[i]);
    min f = sum{i in obs} Err[i]^2;
    solve with nlp/covest=(cov=5 covout=mycov);
    print mycov;
    print a b c;
quit;
```

The solution is displayed in Figure 16.7.
Figure 16.7  Least Squares Problem Estimation Results

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
<tr>
<td>Number of Variables</td>
</tr>
<tr>
<td>Bounded Above</td>
</tr>
<tr>
<td>Bounded Below</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
</tr>
<tr>
<td>Free</td>
</tr>
<tr>
<td>Fixed</td>
</tr>
<tr>
<td>Number of Constraints</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Optimality Error</td>
</tr>
<tr>
<td>Infeasibility</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>mycov</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>0.0000047825</td>
</tr>
<tr>
<td>-.0000000996 0.0000032426</td>
</tr>
<tr>
<td>-.00000000764 -.00000000442 0.0000000017</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.0113</td>
<td>2.0033</td>
<td>0.4998</td>
</tr>
</tbody>
</table>
Syntax: NLP Solver

The following PROC OPTMODEL statement is available for the NLP solver:

```
SOLVE WITH NLP <options>;
```

Functional Summary

Table 16.1 summarizes the options that can be used with the SOLVE WITH NLP statement.

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Covariance Matrix Options and Suboptions</strong></td>
<td></td>
</tr>
<tr>
<td>Requests that the NLP solver compute a covariance matrix</td>
<td>COVEST=()</td>
</tr>
<tr>
<td>Specifies an absolute singularity criterion for matrix inversion</td>
<td>ASINGULAR=</td>
</tr>
<tr>
<td>Specifies the type of covariance matrix</td>
<td>COV=</td>
</tr>
<tr>
<td>Specifies the name of the output covariance matrix</td>
<td>COVOUT=</td>
</tr>
<tr>
<td>Specifies the tolerance for deciding whether a matrix is singular</td>
<td>COVSING=</td>
</tr>
<tr>
<td>Specifies a relative singularity criterion for matrix inversion</td>
<td>MSINGULAR=</td>
</tr>
<tr>
<td>Specifies a number for calculating the divisor for the covariance matrix when VARDEF=DF</td>
<td>NDF=</td>
</tr>
<tr>
<td>Specifies a number for calculating the scale factor for the covariance matrix</td>
<td>NTERMS=</td>
</tr>
<tr>
<td>Specifies a scalar factor for computing the covariance matrix</td>
<td>SIGSQ=</td>
</tr>
<tr>
<td>Specifies the divisor for calculating the covariance matrix</td>
<td>VARDEF=</td>
</tr>
<tr>
<td><strong>Miscellaneous Option</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the seed to use to generate random numbers</td>
<td>SEED=</td>
</tr>
<tr>
<td><strong>Multistart Options</strong></td>
<td></td>
</tr>
<tr>
<td>Directs the local solver to start from multiple initial points</td>
<td>MULTISTART=()</td>
</tr>
<tr>
<td>Specifies the maximum range of values that each variable can take during the sampling process</td>
<td>BNDRANGE=</td>
</tr>
<tr>
<td>Specifies the tolerance for local optima to be considered distinct</td>
<td>DISTTOL=</td>
</tr>
<tr>
<td>Specifies the amount of printing solution progress in multistart mode</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the time limit in multistart mode</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Specifies the maximum number of starting points to be used by the multistart algorithm</td>
<td>MAXSTARTS=</td>
</tr>
<tr>
<td><strong>Optimization Option</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the optimization technique</td>
<td>ALGORITHM=</td>
</tr>
</tbody>
</table>
Table 16.1  Options for the NLP Solver (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Output Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the frequency of printing solution progress (local solvers)</td>
<td>LOGFREQ=</td>
</tr>
<tr>
<td>Specifies the allowable types of output solution</td>
<td>SOLTYPE=</td>
</tr>
<tr>
<td><strong>Solver Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the feasibility tolerance</td>
<td>FEASTOL=</td>
</tr>
<tr>
<td>Specifies the type of Hessian used by the solver</td>
<td>HESSTYPE=</td>
</tr>
<tr>
<td>Enables or disables IIS detection with respect to linear constraints and variable bounds</td>
<td>IIS=</td>
</tr>
<tr>
<td>Specifies the maximum number of iterations</td>
<td>MAXITER=</td>
</tr>
<tr>
<td>Specifies the time limit for the optimization process</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Specifies the number of threads to use for threaded processing</td>
<td>NTHREADS=</td>
</tr>
<tr>
<td>Specifies the upper limit on the objective</td>
<td>OBJLIMIT=</td>
</tr>
<tr>
<td>Specifies the convergence tolerance</td>
<td>OPTTOL=</td>
</tr>
<tr>
<td>Specifies units of CPU time or real time</td>
<td>TIMETYPE=</td>
</tr>
</tbody>
</table>

**NLP Solver Options**

This section describes the options that are recognized by the NLP solver. These options can be specified after a forward slash (/) in the SOLVE statement, provided that the NLP solver is explicitly specified using a WITH clause.

**Covariance Matrix Options**

COVEST=(*suboptions*)

requests that the NLP solver produce a covariance matrix. When this option is applied, the following PROC OPTMODEL options are automatically set: PRESOLVER=NONE and SOLTYPE=0. For more information, see the section “Covariance Matrix” on page 795.

You can specify the following *suboptions*:

ASINGULAR=asing

specifies an absolute singularity criterion for measuring the singularity of the Hessian and crossproduct Jacobian and their projected forms, which might have to be inverted to compute the covariance matrix. The value of asing can be any number between the machine precision and the largest positive number representable in your operating environment. The default is the square root of the machine precision. For more information, see the section “Covariance Matrix” on page 795.

COV=number | string

specifies one of six formulas for computing the covariance matrix. The formula that is used depends on the type of objective (MIN or LSQ) that is specified. Table 16.2 describes the valid values for this option and their corresponding formulas, where nterms is the value of the
NTERMS= option and MIN, LSQ, and other symbols are defined in the section “Covariance Matrix” on page 795.

### Table 16.2 Values of COV= Option

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>MIN Objective</th>
<th>LSQ Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>M</td>
<td>((n_{\text{terms}}/d)G^{-1}JJ(f)G^{-1})</td>
<td>((n_{\text{terms}}/d)G^{-1}VG^{-1})</td>
</tr>
<tr>
<td>2</td>
<td>H</td>
<td>((n_{\text{terms}}/d)G^{-1})</td>
<td>(\sigma^2G^{-1})</td>
</tr>
<tr>
<td>3</td>
<td>J</td>
<td>((1/d)W^{-1})</td>
<td>(\sigma^2JJ(f)^{-1})</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>((1/d)G^{-1}WG^{-1})</td>
<td>(\sigma^2G^{-1}JJ(f)G^{-1})</td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td>((n_{\text{terms}}/d)JJ(f)^{-1})</td>
<td>((1/d)V^{-1})</td>
</tr>
<tr>
<td>6</td>
<td>U</td>
<td>((n_{\text{terms}}/d)W^{-1}JJ(f)W^{-1})</td>
<td>((n_{\text{terms}}/d)JJ(f)^{-1}VJJ(f)^{-1})</td>
</tr>
</tbody>
</table>

For MAX type problems, the covariance matrix is converted to MIN type by using negative Hessian, Jacobian, and function values in the computation. For more information, see the section “Covariance Matrix” on page 795.

By default, COV=2.

**COVOUT=** specifies the name of the parameter that contains the output covariance matrix. Because a covariance matrix is symmetric, you should declare the covariance matrix as either a lower-triangular matrix or a square matrix with indexes starting from 1. For example:

```plaintext
num mycov{i in 1..N, j in 1..i}; /* a lower triangular matrix */
```

or

```plaintext
num mycov{i in 1..N, j in 1..N}; /* a square matrix */
```

where N is the number of variables.

Depending on the type of output covariance matrix, the solver updates either the lower-triangular matrix or the full square matrix. If you declare the covariance matrix as neither a lower-triangular matrix nor a square matrix, or if the indexes do not start from 1, the NLP solver issues an error message. You can use the CREATE DATA statement to output the results to a SAS data set. For more information, see the section “Covariance Matrix” on page 795.

**COVSING=** specifies a threshold, \(civosing > 0\), that determines whether to consider the eigenvalues of a matrix to be 0. The value of \(civosing\) can be any number between the machine precision and the largest positive number representable in your operating environment. The default is set internally by the algorithm. For more information, see the section “Covariance Matrix” on page 795.
**Chapter 16: The Nonlinear Programming Solver**

**MSINGULAR=**

Specifies a relative singularity criterion \( m_{sing} > 0 \) for measuring the singularity of the Hessian and crossproduct Jacobian and their projected forms. The value of \( m_{sing} \) can be any number between machine precision and the largest positive number representable in your operating environment. The default is \( 1 \times 10^{-12} \). For more information, see the section “Covariance Matrix” on page 795.

**NDF=**

Specifies a number to be used in calculating the divisor \( d \), which is used in calculating the covariance matrix when \( \text{VARDEF}=\text{DF} \). The value of \( n_{df} \) can be any positive integer up to the largest four-byte signed integer, which is \( 2^{31} - 1 \). The default is the number of optimization variables in the objective function. For more information, see the section “Covariance Matrix” on page 795.

**NTERMS=**

Specifies a number to be used in calculating the scale factor for the covariance matrix, as shown in Table 16.2. The value of \( n_{terms} \) can be any positive integer up to the largest four-byte signed integer, which is \( 2^{31} - 1 \). The default is the number of nonconstant terms in the objective function. For more information, see the section “Covariance Matrix” on page 795.

**SIGSQ=**

Specifies a real scalar factor, \( s_{q} > 0 \), for computing the covariance matrix. The value of \( s_{q} \) can be any number between the machine precision and the largest positive number representable in your operating environment. For more information, see the section “Covariance Matrix” on page 795.

**VARDEF=DF | N**

Controls how the divisor \( d \) is calculated. This divisor is used in calculating the covariance matrix and approximate standard errors. The value of \( d \) also depends on the values of the \( \text{NDF=} \) and \( \text{NTERMS=} \) options, \( n_{df} \) and \( n_{terms} \), respectively, as follows:

\[
d = \begin{cases} 
\max(1, n_{terms} - n_{df}) & \text{for } \text{VARDEF}=\text{DF} \\
n_{terms} & \text{for } \text{VARDEF}=\text{N}
\end{cases}
\]

By default, \( \text{VARDEF}=\text{DF} \) if the \( \text{SIGSQ=} \) option is not specified; otherwise, by default \( \text{VARDEF}=\text{N} \). For more information, see the section “Covariance Matrix” on page 795.

**Miscellaneous Option**

**SEED=**

Specifies a positive integer to be used as the seed for generating random number sequences. You can use this option to replicate results from different runs.

**Multistart Options**

**MULTISTART=**

Enables multistart mode. In this mode, the local solver solves the problem from multiple starting points, possibly finding a better local minimum as a result. This option is disabled by default. For more information about multistart mode, see the section “Multistart” on page 794.

You can specify the following **suboptions**:
**BNDRANGE=M**

defines the range from which each variable can take values during the sampling process. This option affects only the sampling process that determines starting points for the local solver. It does not affect the bounds of the original nonlinear optimization problem. More specifically, if the \( i \)th variable \( x_i \) has lower and upper bounds \( \ell_i \) and \( u_i \), respectively (that is, \( \ell_i \leq x_i \leq u_i \)), then an initial point is generated by a sampling process as follows:

For each sample point \( x \), the \( i \)th coordinate \( x_i \) is generated so that the following bounds hold, where \( x_i^0 \) is the default starting point or a specified starting point:

\[
\begin{align*}
    l_i &\leq x_i \leq u_i & &\text{if } l_i \text{ and } u_i \text{ are both finite} \\
    l_i &\leq x_i \leq l_i + M & &\text{if only } l_i \text{ is finite} \\
    u_i - M &\leq x_i \leq u_i & &\text{if only } u_i \text{ is finite} \\
    x_i^0 - M/2 &\leq x_i \leq x_i^0 + M/2 & &\text{otherwise}
\end{align*}
\]

The default value is 200 in a shared-memory computing environment and 1,000 in a distributed computing environment.

**DISTTOL=\( \epsilon \)**

defines the tolerance by which two optimal points are considered distinct. Optimal points are considered distinct if the Euclidean distance between them is at least \( \epsilon \). The default is 1.0E–6.

**LOGLEVEL=number**

**PRINTLEVEL=number**

defines the amount of information that the multistart algorithm displays in the SAS log. Table 16.3 describes the valid values of this suboption.

<table>
<thead>
<tr>
<th>number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Turns off all solver-related messages to SAS log</td>
</tr>
<tr>
<td>1</td>
<td>Displays multistart summary information when the algorithm terminates</td>
</tr>
<tr>
<td>2</td>
<td>Displays multistart iteration log and summary information when the algorithm terminates</td>
</tr>
<tr>
<td>3</td>
<td>Displays the same information as LOGLEVEL=2 and might display additional information</td>
</tr>
</tbody>
</table>

By default, LOGLEVEL=2.

**MAXTIME=\( T \)**

defines the maximum allowable time \( T \) (in seconds) for the NLP solver to locate the best local optimum in multistart mode. The value of the TIMETYPE= option determines the type of units that are used. The time that is specified by the MAXTIME= suboption is checked only once after the completion of the local solver. Because the local solver might be called many times, the maximum time that is specified for multistart is recommended to be greater than the maximum time specified for the local solver. If you do not specify this option, the multistart algorithm does not stop based on the amount of time elapsed.
\textbf{MAXSTARTS=N}

defines the maximum number of starting points to be used for local optimization. That is, there will be no more than \( N \) local optimization calls in the multistart algorithm. You can specify \( N \) to be any nonnegative integer. When \( N = 0 \), the algorithm uses the default value of this option. In a shared-memory computing environment, the default value is 100. In a distributed computing environment, the default value is usually greater than 100 and proportional to the number of grid nodes.

\textbf{Optimization Options}

\begin{verbatim}
ALGORITHM=keyword
TECHNIQUE=keyword
TECH=keyword
SOLVER=keyword
\end{verbatim}

specifies the optimization technique to be used to solve the problem. The following \textit{keywords} are valid:

\textbf{INTERIORPOINT}

uses a primal-dual interior point method. This technique is recommended for both small- and large-scale nonlinear optimization problems. This is the preferred solver if the problem includes a large number of inactive constraints.

\textbf{ACTIVESET}

uses a primal-dual active-set method. This technique is recommended for both small- and large-scale nonlinear optimization problems. This is the preferred solver if the problem includes only bound constraints or if the optimal active set can be quickly determined by the solver.

\textbf{CONCURRENT (experimental)}

runs the INTERIORPOINT and ACTIVESET techniques in parallel, with one thread using the INTERIORPOINT technique and the other thread using the ACTIVESET technique. The solution is returned by the first method that terminates.

The default is INTERIORPOINT.

\textbf{Output Options}

\begin{verbatim}
LOGFREQ=N
PRINTFREQ=N
SOLTYPE=0 | 1
\end{verbatim}

specifies how often the iterations are displayed in the SAS log. \( N \) should be an integer between zero and the largest four-byte, signed integer, which is \( 2^{31} - 1 \). If \( N \geq 1 \), the solver prints only those iterations that are a multiple of \( N \). If \( N = 0 \), no iteration is displayed in the log. The default value is 1.

\textbf{SOLTYPE=0 | 1}

specifies whether the NLP solver should return only a solution that is locally optimal. If SOLTYPE=0, the solver returns a locally optimal solution, provided it locates one. If SOLTYPE=1, the solver returns the best feasible solution found, provided its objective value is better than that of the locally optimal solution found. The default is 1.
Solver Options

**FEASTOL=** \( \varepsilon \)
defines the feasible tolerance. The solver will exit if the constraint violation is less than FEASTOL and the scaled optimality conditions are less than OPTTOL. The default is \( \varepsilon = 1 \times 10^{-6} \).

**HESSTYPE=** FULL | PRODUCT
specifies the type of Hessian to be used by the solver. The valid keywords for this option are FULL and PRODUCT. If HESSTYPE=FULL, the solver uses a full Hessian. If HESSTYPE=PRODUCT, the solver uses only Hessian-vector products, not the full Hessian. When the solver uses only Hessian-vector products to find a search direction, it usually uses much less memory, especially when the problem is large and the Hessian is not sparse. On the other hand, when the full Hessian is used, the algorithm can create a better preconditioner to solve the problem in less CPU time. The default is FULL.

**IIS=** TRUE | FALSE
specifies whether the NLP solver attempts to identify a set of linear constraints and variables that form an irreducible infeasible set (IIS). Table 16.4 describes the valid values of the IIS= option.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRUE</td>
<td>Enables IIS detection.</td>
</tr>
<tr>
<td>FALSE</td>
<td>Disables IIS detection.</td>
</tr>
</tbody>
</table>

The default is FALSE.

Note that when the IIS= option is enabled, all the other NLP solver options are ignored except the following:

<table>
<thead>
<tr>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>FEASTOL=</td>
</tr>
<tr>
<td>LOGFREQ=</td>
</tr>
<tr>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>MAXITER=</td>
</tr>
<tr>
<td>MAXTIME=</td>
</tr>
<tr>
<td>TIMETYPE=</td>
</tr>
</tbody>
</table>

The NLP solver ignores nonlinear constraints, if any, and invokes the LP solver’s algorithm to attempt to identify an IIS. If an IIS is found, information about the infeasibilities can be found in the .status suffix values of the constraints and variables. For more information about the IIS= option, see the section "Irreducible Infeasible Set" on page 519 of Chapter 12, “The Linear Programming Solver.” Also see Example 16.7 for an example that demonstrates the use of the IIS= option of the NLP solver.

**MAXITER=** \( N \)
specifies that the solver take at most \( N \) major iterations to determine an optimum of the NLP problem. The value of \( N \) is an integer between zero and the largest four-byte, signed integer, which is \( 2^{31} - 1 \). A major iteration in NLP consists of finding a descent direction and a step size along which the next approximation of the optimum resides. The default is 5,000 iterations.
MAXTIME=$t$

specifies an upper limit of $t$ units of time for the optimization process, including problem generation time and solution time. The value of the TIMETYPE= option determines the type of units used. If you do not specify the MAXTIME= option, the solver does not stop based on the amount of time elapsed. The value of $t$ can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

NTHREADS=$number$

when the MULTISTART= option is not used, specifies the number of threads that the interior point, active-set, or concurrent algorithm can use to perform linear algebra operations. When the MULTISTART= option is used, it specifies the number of threads for the multistart algorithm to use to concurrently start local solves. The value of $number$ must be an integer between 1 and 256, inclusive. The default is the value of the NTHREADS= option in PROC OPTMODEL.

OBJLIMIT=$M$

specifies an upper limit on the magnitude of the objective value. For a minimization problem, the algorithm terminates when the objective value becomes less than $-M$; for a maximization problem, the algorithm stops when the objective value exceeds $M$. The algorithm stopping implies that either the problem is unbounded or the algorithm diverges. If optimization were allowed to continue, numerical difficulty might be encountered. The default is $M=1E+20$. The minimum acceptable value of $M$ is $1E+8$. If the specified value of $M$ is less than $1E+8$, the value is reset to the default value $1E+20$.

OPTTOL=$\epsilon$

RELOPTTOL=$\epsilon$

defines the measure by which you can decide whether the current iterate is an acceptable approximation of a local minimum. The value of this option is a positive real number. The NLP solver determines that the current iterate is a local minimum when the norm of the scaled vector of the optimality conditions is less than $\epsilon$ and the true constraint violation is less than FEASTOL. The default is $\epsilon=1E–6$.

TIMETYPE=CPU | REAL

specifies the units of time used by the MAXTIME= option and reported by the PRESOLVE_TIME and SOLUTION_TIME terms in the _OROPTMODEL_ macro variable. Table 16.6 describes the valid values of the TIMETYPE= option.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Specifies units of CPU time</td>
</tr>
<tr>
<td>REAL</td>
<td>Specifies units of real time</td>
</tr>
</tbody>
</table>

The “Optimization Statistics” table, an output of PROC OPTMODEL if you specify PRINTLEVEL=2 in the PROC OPTMODEL statement, also includes the same time units for Presolver Time and Solver Time. The other times (such as Problem Generation Time) in the “Optimization Statistics” table are also in the same units.

The default value of the TIMETYPE= option is REAL.
This section presents a brief discussion about the algorithmic details of the NLP solver. First, the notation is defined. Next, an introduction to the fundamental ideas in constrained optimization is presented; the main point of the second section is to present the necessary and sufficient optimality conditions, which play a central role in all optimization algorithms. The section concludes with a general overview of primal-dual interior point and active-set algorithms for nonlinear optimization. A detailed treatment of the preceding topics can be found in Nocedal and Wright (1999), Wright (1997), and Forsgren, Gill, and Wright (2002).

Basic Definitions and Notation

The gradient of a function \( f : \mathbb{R}^n \rightarrow \mathbb{R} \) is the vector of all the first partial derivatives of \( f \) and is denoted by
\[
\nabla f(x) = \left( \frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)^T
\]
where the superscript \( T \) denotes the transpose of a vector.

The Hessian matrix of \( f \), denoted by \( \nabla^2 f(x) \), or simply by \( H(x) \), is an \( n \times n \) symmetric matrix whose \((i, j)\) element is the second partial derivative of \( f(x) \) with respect to \( x_i \) and \( x_j \). That is, \( H_{i,j}(x) = \frac{\partial^2 f(x)}{\partial x_i \partial x_j} \).

Consider the vector function, \( c : \mathbb{R}^n \rightarrow \mathbb{R}^{p+q} \), whose first \( p \) elements are the equality constraint functions \( h_i(x), i = 1, 2, \ldots, p \), and whose last \( q \) elements are the inequality constraint functions \( g_i(x), i = 1, 2, \ldots, q \). That is,
\[
c(x) = (h(x):g(x))^T = (h_1(x), \ldots, h_p(x):g_1(x), \ldots, g_q(x))^T
\]
The \((p + q) \times n \) matrix whose \( i \)th row is the gradient of the \( i \)th element of \( c(x) \) is called the Jacobian matrix of \( c(x) \) (or simply the Jacobian of the NLP problem) and is denoted by \( J(x) \). You can also use \( J_h(x) \) to denote the \( p \times n \) Jacobian matrix of the equality constraints and use \( J_g(x) \) to denote the \( q \times n \) Jacobian matrix of the inequality constraints.

Constrained Optimization

A function that plays a pivotal role in establishing conditions that characterize a local minimum of an NLP problem is the Lagrangian function \( \mathcal{L}(x, y, z) \), which is defined as
\[
\mathcal{L}(x, y, z) = f(x) - \sum_{i \in \mathcal{E}} y_i h_i(x) - \sum_{i \in \mathcal{I}} z_i g_i(x)
\]
Note that the Lagrangian function can be seen as a linear combination of the objective and constraint functions. The coefficients of the constraints, \( y_i, i \in \mathcal{E} \), and \( z_i, i \in \mathcal{I} \), are called the Lagrange multipliers or dual variables. At a feasible point \( \hat{x} \), an inequality constraint is called active if it is satisfied as an equality—that is, \( g_i(\hat{x}) = 0 \). The set of active constraints at a feasible point \( \hat{x} \) is then defined as the union of the index set of the equality constraints, \( \mathcal{E} \), and the indices of those inequality constraints that are active at \( \hat{x} \); that is,
\[
\mathcal{A}(\hat{x}) = \mathcal{E} \cup \{ i \in \mathcal{I} : g_i(\hat{x}) = 0 \}
\]
An important condition that is assumed to hold in the majority of optimization algorithms is the so-called linear independence constraint qualification (LICQ). The LICQ states that at any feasible point $\hat{x}$, the gradients of all the active constraints are linearly independent. The main purpose of the LICQ is to ensure that the set of constraints is well-defined in a way that there are no redundant constraints or in a way that there are no constraints defined such that their gradients are always equal to zero.

**The First-Order Necessary Optimality Conditions**

If $x^*$ is a local minimum of the NLP problem and the LICQ holds at $x^*$, then there are vectors of Lagrange multipliers $y^*$ and $z^*$, with components $y^*_i, i \in \mathcal{E}$, and $z^*_i, i \in \mathcal{I}$, respectively, such that the following conditions are satisfied:

$$
\begin{align*}
\nabla_x L(x^*, y^*, z^*) &= 0 \\
h_i(x^*) &= 0, \quad i \in \mathcal{E} \\
g_i(x^*) &\geq 0, \quad i \in \mathcal{I} \\
z^*_i &\geq 0, \quad i \in \mathcal{I} \\
z^*_i g_i(x^*) &= 0, \quad i \in \mathcal{I}
\end{align*}
$$

where $\nabla_x L(x^*, y^*, z^*)$ is the gradient of the Lagrangian function with respect to $x$, defined as

$$
\nabla_x L(x^*, y^*, z^*) = \nabla f(x) - \sum_{i \in \mathcal{E}} y_i \nabla h_i(x) - \sum_{i \in \mathcal{I}} z_i \nabla g_i(x)
$$

The preceding conditions are often called the Karush-Kuhn-Tucker (KKT) conditions. The last group of equations ($z_i g_i(x) = 0, i \in \mathcal{I}$) is called the complementarity condition. Its main aim is to try to force the Lagrange multipliers, $z^*_i$, of the inactive inequalities (that is, those inequalities with $g_i(x^*) > 0$) to zero.

The KKT conditions describe the way the first derivatives of the objective and constraints are related at a local minimum $x^*$. However, they are not enough to fully characterize a local minimum. The second-order optimality conditions attempt to fulfill this aim by examining the curvature of the Hessian matrix of the Lagrangian function at a point that satisfies the KKT conditions.

**The Second-Order Necessary Optimality Condition**

Let $x^*$ be a local minimum of the NLP problem, and let $y^*$ and $z^*$ be the corresponding Lagrange multipliers that satisfy the first-order optimality conditions. Then $d^T \nabla^2_x L(x^*, y^*, z^*) d \geq 0$ for all nonzero vectors $d$ that satisfy the following conditions:

1. $\nabla h_i^T(x^*) d = 0, \quad \forall i \in \mathcal{E}$
2. $\nabla g_i^T(x^*) d = 0, \quad \forall i \in \mathcal{A}(x^*) \cap \mathcal{I}$, such that $z^*_i > 0$
3. $\nabla g_i^T(x^*) d \geq 0, \quad \forall i \in \mathcal{A}(x^*) \cap \mathcal{I}$, such that $z^*_i = 0$

The second-order necessary optimality condition states that, at a local minimum, the curvature of the Lagrangian function along the directions that satisfy the preceding conditions must be nonnegative.
Interior Point Algorithm

Primal-dual interior point methods can be classified into two categories: feasible and infeasible. The first category requires that the starting point and all subsequent iterations of the algorithm strictly satisfy all the inequality constraints. The second category relaxes those requirements and allows the iterations to violate some or all of the inequality constraints during the course of the minimization procedure. The NLP solver implements an infeasible algorithm; this section concentrates on that type of algorithm.

To make the notation less cluttered and the fundamentals of interior point methods easier to understand, consider without loss of generality the following simpler NLP problem:

\[
\begin{align*}
\text{minimize} & \quad f(x) \\
\text{subject to} & \quad g_i(x) \geq 0, i \in \mathcal{I} = \{1, 2, \ldots, q\}
\end{align*}
\]

Note that the equality and bound constraints have been omitted from the preceding problem. Initially, slack variables are added to the inequality constraints, giving rise to the problem

\[
\begin{align*}
\text{minimize} & \quad f(x) \\
\text{subject to} & \quad g_i(x) - s_i = 0, i \in \mathcal{I} \\
& \quad s \geq 0
\end{align*}
\]

where \( s = (s_1, \ldots, s_q)^T \) is the vector of slack variables, which are required to be nonnegative. Next, all the nonnegativity constraints on the slack variables are eliminated by being incorporated into the objective function, by means of a logarithmic function. This gives rise to the equality-constrained NLP problem

\[
\begin{align*}
\text{minimize} & \quad B(x, s) = f(x) - \mu \sum_{i \in \mathcal{I}} \ln(s_i) \\
\text{subject to} & \quad g_i(x) - s_i = 0, i \in \mathcal{I}
\end{align*}
\]

where \( \mu \) is a positive parameter. The nonnegativity constraints on the slack variables are implicitly enforced by the logarithmic functions, since the logarithmic function prohibits \( s \) from taking zero or negative values.

Next, the equality constraints can be absorbed by using a quadratic penalty function to obtain the following:

\[
\begin{align*}
\text{minimize} & \quad M(x, s) = f(x) + \frac{1}{2\mu} \|g(x) - s\|^2 - \mu \sum_{i \in \mathcal{I}} \ln(s_i)
\end{align*}
\]

The preceding unconstrained problem is often called the penalty-barrier subproblem. Depending on the size of the parameter \( \mu \), a local minimum of the barrier problem provides an approximation to the local minimum of the original NLP problem. The smaller the size of \( \mu \), the better the approximation becomes. Infeasible primal-dual interior point algorithms repeatedly solve the penalty-barrier problem for different values of \( \mu \) that progressively go to zero, in order to get as close as possible to a local minimum of the original NLP problem.

An unconstrained minimizer of the penalty-barrier problem must satisfy the equations

\[
\begin{align*}
\nabla f(x) - J(x)^T z &= 0 \\
-\mu S^{-1} e &= 0
\end{align*}
\]

where \( z = -(g(x) - s)/\mu \), \( J(x) \) is the Jacobian matrix of the vector function \( g(x) \), \( S \) is the diagonal matrix whose elements are the elements of the vector \( s \) (that is, \( S = \text{diag}\{s_1, \ldots, s_q\} \)), and \( e \) is a vector of all ones.
Multiplying the second equation by $S$ and adding the definition of $z$ as a third equation produces the following equivalent nonlinear system:

$$F^\mu(x, s, z) = \begin{pmatrix} \nabla f(x) - J(x)^T z \\ S z - e \\ g(x) - s + \mu z \end{pmatrix} = 0$$

Note that if $\mu = 0$, the preceding conditions represent the optimality conditions of the original optimization problem, after adding slack variables. One of the main aims of the algorithm is to gradually reduce the value of $\mu$ to zero, so that it converges to a local optimum of the original NLP problem. The rate by which $\mu$ approaches zero affects the overall efficiency of the algorithm. Algorithms that treat $z$ as an additional variable are considered primal-dual, while those that enforce the definition of $z = -(g(x) - s)/\mu$ at each iteration are consider purely primal approaches.

At iteration $k$, the infeasible primal-dual interior point algorithm approximately solves the preceding system by using Newton’s method. The Newton system is

$$\begin{bmatrix} H_L(x^k, z^k) & 0 & -J(x^k)^T \\ 0 & Z^k & S^k \\ J(x^k) & -I & \mu I \end{bmatrix} \begin{bmatrix} \Delta x^k \\ \Delta s^k \\ \Delta z^k \end{bmatrix} = -\begin{bmatrix} \nabla f(x^k) - J(x^k)^T z \\ -\mu e + S^k z^k \\ g(x^k) - s^k + \mu z^k \end{bmatrix}$$

where $H_L$ is the Hessian matrix of the Lagrangian function $L(x, z) = f(x) - z^T g(x)$ of the original NLP problem; that is,

$$H_L(x, z) = \nabla^2 f(x) - \sum_{i \in I} z_i \nabla^2 g_i(x)$$

The solution $(\Delta x^k, \Delta s^k, \Delta z^k)$ of the Newton system provides a direction to move from the current iteration $(x^k, s^k, z^k)$ to the next,

$$(x^{k+1}, s^{k+1}, z^{k+1}) = (x^k, s^k, z^k) + \alpha (\Delta x^k, \Delta s^k, \Delta z^k)$$

where $\alpha$ is the step length along the Newton direction. The step length is determined through a line-search procedure that ensures sufficient decrease of a merit function based on the augmented Lagrangian function of the barrier problem. The role of the merit function and the line-search procedure is to ensure that the objective and the infeasibility reduce sufficiently at every iteration and that the iterations approach a local minimum of the original NLP problem.

**Active-Set Method**

Active-set methods differ from interior point methods in that no barrier term is used to ensure that the algorithm remains interior with respect to the inequality constraints. Instead, attempts are made to learn the true active set. For simplicity, use the same initial slack formulation used by the interior point method description,

$$\begin{align*}
\text{minimize} \quad & f(x) \\
\text{subject to} \quad & g_i(x) - s_i = 0, \ i \in I \\
& s \geq 0
\end{align*}$$
where $s = (s_1, \ldots, s_q)^T$ is the vector of slack variables, which are required to be nonnegative. Begin by absorbing the equality constraints as before into a penalty function, but keep the slack bound constraints explicitly:

$$\text{minimize} \quad \mathcal{M}(x, s) = f(x) + \frac{1}{2\mu} \| g(x) - s \|^2_2$$
subject to $s \geq 0$

where $\mu$ is a positive parameter. Given a solution pair $(x(\mu), s(\mu))$ for the preceding problem, you can define the active-set projection matrix $P$ as follows:

$$P_{ij} = \begin{cases} 1 & \text{if } i = j \text{ and } s_i(\mu) = 0 \\ 0 & \text{otherwise.} \end{cases}$$

Then $(x(\mu), s(\mu))$ is also a solution of the equality constraint subproblem:

$$\text{minimize} \quad \mathcal{M}(x, s) = f(x) + \frac{1}{2\mu} \| g(x) - s \|^2_2$$
subject to $Ps = 0$.

The minimizer of the preceding subproblem must be a stationary point of the Lagrangian function

$$L^\mu(x, s, z) = f(x) + \frac{1}{2\mu} \| g(x) - s \|^2_2 - z^T Ps$$

which gives the optimality equations

$$\begin{align*}
\nabla_x L^\mu(x, s, z) &= \nabla f(x) - J(x)^T y = 0 \\
\nabla_s L^\mu(x, s, z) &= y - P^T z = 0 \\
&= Ps = 0
\end{align*}$$

where $y = -(g(x) - s)/\mu$. Using the second equation, you can simplify the preceding equations to get the following optimality conditions for the bound-constrained penalty subproblem:

$$\begin{align*}
\nabla f(x) - J(x)^T P^T z &= 0 \\
P(g(x) - s) + \mu z &= 0 \\
Ps &= 0
\end{align*}$$

Using the third equation directly, you can reduce the system further to

$$\begin{align*}
\nabla f(x) - J(x)^T P^T z &= 0 \\
Pg(x) + \mu z &= 0
\end{align*}$$

At iteration $k$, the primal-dual active-set algorithm approximately solves the preceding system by using Newton’s method. The Newton system is

$$\begin{bmatrix} H_C(x^k, z^k) & -J_A^T \\ J_A & \mu I \end{bmatrix} \begin{bmatrix} \Delta x^k \\ \Delta z^k \end{bmatrix} = -\begin{bmatrix} \nabla_x f(x^k) - J_A^T z^k \\ Pg(x^k) + \mu z^k \end{bmatrix}$$

where $J_A = PJ(x^k)$ and $H_C$ denotes the Hessian of the Lagrangian function $f(x) - z^T Pg(x)$. The solution $(\Delta x^k, \Delta z^k)$ of the Newton system provides a direction to move from the current iteration $(x^k, s^k, z^k)$ to the next,

$$(x^{k+1}, z^{k+1}) = (x^k, z^k) + \alpha(\Delta x^k, \Delta z^k)$$
where $\alpha$ is the step length along the Newton direction. The corresponding slack variable update $s^{k+1}$ is defined as the solution to the following subproblem whose solution can be computed analytically:

$$
\min_{\mathcal{M}} \mathcal{M}(x^{k+1}, s) = f(x) + \frac{1}{2\mu} \| g(x^{k+1}) - s \|_2^2 \\
\text{subject to } s \geq 0
$$

The step length $\alpha$ is then determined in a similar manner to the preceding interior point approach. At each iteration, the definition of the active-set projection matrix $P$ is updated with respect to the new value of the constraint function $g(x^{k+1})$. For large-scale NLP, the computational bottleneck typically arises in seeking to solve the Newton system. Thus active-set methods can achieve substantial computational savings when the size of $J_A$ is much smaller than $J(x)$; however, convergence can be slow if the active-set estimate changes combinatorially. Further, the active-set algorithm is often the superior algorithm when only bound constraints are present. In practice, both the interior point and active-set approach incorporate more sophisticated merit functions than those described in the preceding sections; however, their description is beyond the scope of this document. See Gill and Robinson (2010) for further reading.

### Multistart

Frequently, nonlinear optimization problems contain many local minima because the objective or the constraints are nonconvex functions. The quality of different local minima is measured by the objective value achieved at those points. For example, if $x_1^*$ and $x_2^*$ are two distinct local minima and $f(x_1^*) \leq f(x_2^*)$, then $x_1^*$ is said to be of better quality than $x_2^*$. The NLP solver provides a mechanism that can locate local minima of better quality by starting the local solver multiple times from different initial points. By doing so, the local solver can converge to different local minima. The local minimum with the lowest objective value is then reported back to the user.

The multistart feature consists of two phases. In the first phase, the entire feasible region is explored by generating sample points from a uniform distribution. The aim of this phase is to place at least one sample point in the region of attraction of every local minimum. Here the region of attraction of a local minimum is defined as the set of feasible points that, when used as starting points, enable a local solver to converge to that local minimum.

During the second phase, a subset of the sample points generated in the first phase is chosen by applying a clustering technique. The goal of the clustering technique is to group the initial sample points around the local minima and allow only a single local optimization to start from each cluster or group. The clustering technique aims to reduce computation time by sparing the work of unnecessarily starting multiple local optimizations within the region of attraction of the same local minimum.

The number of starting points is critical to the time spent by the solver to find a good local minimum. You can specify the maximum number of starting points by using the MAXSTARTS= suboption. If this option is not specified, the solver determines the minimum number of starting points that can provide reasonable evidence that a good local minimum will be found.

Many optimization problems contain variables with infinite upper or lower bounds. These variables can cause the sampling procedure to generate points that are not useful for locating different local minima. The efficiency of the sampling procedure can be increased by reducing the range of these variables by using the BNDRANGE= suboption. This option forces the sampling procedure to generate points that are in a smaller interval, thereby increasing the efficiency of the solver to converge to a local optimum.
The multistart feature currently supports deterministic parallel processing in a shared-memory computing environment and in single-machine mode of a distributed computing environment. To ensure reproducible results, specify a nonzero value for the SEED= option. In a distributed computing environment, the multistart feature is nondeterministic.

**Accessing the Starting Point That Leads to the Best Local Optimum**

The starting point that leads to the best local optimum can be accessed by using the .msinit suffix in PROC OPTMODEL. In some cases, the knowledge of that starting point might be useful. For example, you can run the local solver again but this time providing as initial point the one that is stored in .msinit. This way the multistart explores a different part of the feasible region and might discover a local optimum of better quality than those found in previous runs. The use of the suffix .msinit is demonstrated in Example 16.5. For more information about suffixes in PROC OPTMODEL, see “Suffixes” on page 380 in Chapter 10, “The OPTMODEL Procedure.”

**Covariance Matrix**

You must specify the COVEST=() option to compute an approximate covariance matrix for the parameter estimates under asymptotic theory for least squares, maximum likelihood, or Bayesian estimation, with or without corrections for degrees of freedom as specified in the VARDEF= option.

The standard form of this class of the problems is one of following:

- **least squares (LSQ):** \( \min f(x) = s \sum_{i=1}^{m} f_i^2(x) \)
- **minimum or maximum (MIN/MAX):** \( \operatorname{opt} f(x) = s \sum_{i=1}^{m} f_i(x) \)

For example, two groups of six different forms of covariance matrices (and therefore approximate standard errors) can be computed corresponding to the following two situations, where TERMS is an index set.

- **LSQ:** The objective function consists solely of a positively scaled sum of squared terms, which means that least squares estimates are being computed:
  \[
  \min f(x) = s \sum_{(i,j) \in \text{TERMS}} f_{ij}^2(x)
  \]
  where \( s > 0 \).

- **MIN or MAX:** The MIN or MAX declaration is specified, and the objective is not in least squares form. Together, these characteristics mean that maximum likelihood or Bayesian estimates are being computed:
  \[
  \operatorname{opt} f(x) = s \sum_{(i,j) \in \text{TERMS}} f_{ij}(x)
  \]
  where \( \operatorname{opt} \) is either \( \min \) or \( \max \) and \( s \) is arbitrary.

In the preceding section, TERMS is used to denote an arbitrary index set. For example, if your problem is

\[
\min z = 0.5 \sum_{i \in I} (g_1^2[i] + g_2^2[i])
\]
then \( \text{TERMS} = \{(i, j) : i \in \mathcal{I} \text{ and } j \in \{1, 2\}\} \), where \( \mathcal{I} \) is the index set of input data. The following rules apply when you specify your objective function:

- The terms \( f_{ij}(x) \) can be either IMPVAR expressions or constant expressions (expressions that do not depend on variables). The \( i \) and \( j \) values can be partitioned among observation and function indices as needed. Any number of indices can be used, including non-array indices to implicit variables.

- The nonconstant terms are defined by using the IMPVAR declaration. Each nonconstant IMPVAR element can be referenced at most once in the objective.

- The objective consists of a scaled sum of terms (or squared terms for least squares). The scaling, shown as \( s \) in the preceding equations, consists of outer multiplication or division by constants of the unscaled sum of terms (or squared terms for least squares). The unary + or − operators can also be used for scaling.

- Least squares objectives require the scaling to be positive \( (s > 0) \). The individual \( f_{ij} \) values are scaled by \( \sqrt{s} \) by PROC OPTMODEL.

- Objectives that are not least squares allow arbitrary scaling. The scale value is distributed to the \( f_{ij} \) values.

- The summation of terms (or squared terms for least squares) is constructed with the binary +, SUM, and IF-THEN-ELSE operators (where IF-THEN-ELSE must have a first operand that does not depend on variables). The operands can be terms or a summation of terms (or squared terms for least squares).

- A squared term is specified as \( \text{term}^2 \) or \( \text{term}^{**2} \).

- The default value of the NTERMS= option is determined by counting the nonconstant terms. The constant terms do not contribute to the covariance matrix.

The following PROC OPTMODEL statements demonstrate these rules:

```plaintext
var x{VARS};
impvar g{OBS} = ...; /* expression involves x */
impvar h{OBS} = ...; /* expression involves x */
/* This objective is okay. */
min z1 = sum{i in OBS} (g[i] + h[i]);
/* This objective is okay. */
min z2 = 0.5*sum{i in OBS} (g[i]^2 + h[i]^2);
/* This objective is okay. It demonstrates multiple levels of scaling. */
min z3 = 3*(sum{i in OBS} (g[i]^2 + h[i]^2))/2;
/* This objective is okay. */
min z4 = (sum{i in OBS} (g[i]^2 + h[i]^2))/2;
```

Note that the following statements are not accepted:
Covariance Matrix

/* This objective causes an error because individual scaling is not allowed. */
/* (division applies to inner term) */
min z5 = sum{i in OBS} (g[i]^2 + h[i]^2)/2;

/* This objective causes an error because individual scaling is not allowed. */
min z6 = sum{i in OBS} 0.5* g[i]^2;

/* This objective causes an error because the element g[1] is repeated. */
min z7 = g[1] + sum{i in OBS} g[i];

The covariance matrix is always positive semidefinite. For MAX type problems, the covariance matrix is converted to MIN type by using negative Hessian, Jacobian, and function values in the computation. You can use the following options to check for a rank deficiency of the covariance matrix:

- The ASINGULAR= and MSINGULAR= options enable you to set two singularity criteria for the inversion of the matrix $A$ that is needed to compute the covariance matrix, when $A$ is either the Hessian or one of the crossproduct Jacobian matrices. The singularity criterion that is used for the inversion is

$$|d_{j,j}| \leq \max(\text{asing}, \text{msing} \times \max(|A_{1,1}|, \ldots, |A_{n,n}|))$$

where $d_{j,j}$ is the diagonal pivot of the matrix $A$, and $\text{asing}$ and $\text{msing}$ are the specified values of the ASINGULAR= and MSINGULAR= options, respectively.

- If the matrix $A$ is found to be singular, the NLP solver computes a generalized inverse that satisfies Moore-Penrose conditions. The generalized inverse is computed using the computationally expensive but numerically stable eigenvalue decomposition, $A = Z \Lambda Z^T$, where $Z$ is the orthogonal matrix of eigenvectors and $\Lambda$ is the diagonal matrix of eigenvalues, $\Lambda = \text{diag}(\lambda_1, \ldots, \lambda_n)$. The generalized inverse of $A$ is set to

$$A^{-} = Z \Lambda^{-} Z^T$$

where the diagonal matrix $\Lambda^{-} = \text{diag}(\lambda_1^{-}, \ldots, \lambda_n^{-})$ is defined as follows, where $\text{covsing}$ is the specified value of the COVSING= option:

$$\lambda_i^{-} = \begin{cases} 1/\lambda_i & \text{if } |\lambda_i| > \text{covsing} \\ 0 & \text{if } |\lambda_i| \leq \text{covsing} \end{cases}$$

If the COVSING= option is not specified, then the default is $\max(\text{asing}, \text{msing} \times \max(|A_{1,1}|, \ldots, |A_{n,n}|))$, where $\text{asing}$ and $\text{msing}$ are the specified values of the ASINGULAR= and MSINGULAR= options, respectively.

For problems of the MIN or LSQ type, the matrices that are used to compute the covariance matrix are

$$G = \nabla^2 f(x)$$

$$J(f) = (\nabla f_1, \ldots, \nabla f_m) = \left( \frac{\partial f_i}{\partial x_j} \right)$$

$$JJ(f) = J(f)^T J(f)$$

$$V = J(f)^T \text{diag}(f_i^2) J(f)$$
where \( f_i \) is defined in the standard form of the covariance matrix problem. Note that when some \( f_i \) are 0, \((\text{diag}(f_i))^{-1}\) is computed as a generalized inverse.

For unconstrained minimization, the formulas of the six types of covariance matrices are given in Table 16.2. The value of \( d \) in the table depends on the VARDEF= option and the values of the NDF= and NTERMS= options, \( ndf \) and \( nterms \), respectively, as follows:

\[
d = \begin{cases} 
\max(1, nterms - ndf) & \text{for VARDEF=DF} \\
nterms & \text{for VARDEF=N}
\end{cases}
\]

The value of \( \sigma^2 \) depends on the specification of the SIGSQ= option and on the value of \( d \),

\[
\sigma^2 = \begin{cases} 
sq \times nterms/d & \text{if SIGSQ=sq is specified} \\
2f(x^*)/d & \text{if SIGSQ= is not specified}
\end{cases}
\]

where \( f(x^*) \) is the value of the objective function at the optimal solution \( x^* \).

Because of the analytic definition, in exact arithmetic the covariance matrix is positive semidefinite at the solution. A warning message is issued if numerical computation does not result in a positive semidefinite matrix. This can happen because round-off error is introduced or the incorrect type of covariance matrix for a specified problem is selected.

**Iteration Log for the Local Solver**

The iteration log for the local solver provides detailed information about progress towards a locally optimal solution. This log appears when multistart mode is disabled.

The following information is displayed in the log:

- **Iter** indicates the iteration number.
- **Objective Value** indicates the objective function value.
- **Infeasibility** indicates the maximum value out of all constraint violations.
- **Optimality Error** indicates the relative optimality error (see the section “Solver Termination Criterion” on page 799).

**Iteration Log for Multistart**

When the MULTISTART=() option is specified, the iteration log differs from that of the local solver. More specifically, when a value of 2 is specified for the LOGLEVEL= suboption, the following information is displayed in the log:

- **Start** indicates the index number of each local optimization run. The following indicator can appear beside this number to provide additional information about the run:
Solver Termination Criterion

Because badly scaled problems can lead to slow convergence, the NLP solver dynamically rescales both the objective and constraint functions adaptively as needed. The optimality conditions are always stated with respect to the rescaled NLP. However, because typically you are most interested in the constraint violation of the original NLP, and not the internal scaled variant, you always work with respect to the true constraint violation. Thus, the solver terminates when both of the following conditions are true:

* indicates the local optimization started from a user-supplied point.

Best Objective indicates the value of the objective function at the best local solution found so far.

Local Objective indicates the value of the objective function obtained at the solution returned by the local solver.

Infeasibility indicates the infeasibility error at the solution returned by the local solver.

Optimality Error indicates the optimality error at the solution returned by the local solver.

Local Iter indicates the number of iterations taken by the local solver.

Local Status indicates the solution status of the local solver. Several different values can appear in this column:

OPTIMAL indicates that the local solver found a locally optimal solution.

BESTFEASIBLE indicates that the local solver returned the best feasible point found. See the SOLTYPE= option for more information.

INFEASIBLE indicates that the local solver converged to a point that might be infeasible.

LOCALINFEAS indicates that the local solver converged to a point of minimal local infeasibility.

UNBOUNDED indicates that the local solver determined that the problem is unbounded.

ITERLIMIT indicates that the local solver reached the maximum number of iterations and could not find a locally optimal solution.

TIMELIMIT indicates that the local solver reached the maximum allowable time and could not find a locally optimal solution.

ABORTED indicates that the local solver terminated due to a user interrupt.

FUNEVALERR indicates that the local solver encountered a function evaluation error.

NUMERICERR indicates that the local solver encountered a numerical error other than a function evaluation error.

INTERNALERR indicates that the local solver encountered a solver system error.

OUTMEMORY indicates that the local solver ran out of memory.

FAILED indicates a general failure of the local solver in the absence of any other error.
The norm of the optimality conditions of the scaled NLP is less than the user-defined or default tolerance (OPTTOL= option).

- The norm of the constraint violation of the original NLP is less than the user-defined feasibility tolerance (FEASTOL= option).

More specifically, if

$$F(x, s, z) = (\nabla_x f(x) - J(x)^T z, \quad S z, \quad g(x) - s)^T$$

is the vector of the optimality conditions of the rescaled NLP problem, then the solver terminates when

$$\| F(x, s, z) \| \leq \text{OPTTOL}(1 + \|(x, s)\|)$$

and the maximum constraint violation is less than FEASTOL.

---

**Solver Termination Messages**

Upon termination, the solver produces the following messages in the log:

**Optimal**

The solver has successfully found a local solution to the optimization problem.

**Conditionally optimal solution found**

The solver is sufficiently close to a local solution, but it has difficulty in completely satisfying the user-defined optimality tolerance. This can happen when the line search finds very small steps that result in very slight progress of the algorithm. It can also happen when the prespecified tolerance is too strict for the optimization problem at hand.

**Maximum number of iterations reached**

The solver could not find a local optimum in the prespecified number of iterations.

**Maximum specified time reached**

The solver could not find a local optimum in the prespecified maximum real time for the optimization process.

**Did not converge**

The solver could not satisfy the optimality conditions and failed to converge.

**Problem might be unbounded**

The objective function takes arbitrarily large values, and the optimality conditions fail to be satisfied. This can happen when the problem is unconstrained or when the problem is constrained and the feasible region is not bounded.

**Problem might be infeasible**

The solver cannot identify a point in the feasible region.
**Problem is infeasible**
The solver detects that the problem is infeasible.

**Out of memory**
The problem is so large that the solver requires more memory to solve the problem.

**Problem solved by the OPTMODEL presolver**
The problem was solved by the OPTMODEL presolver.

---

**Macro Variable _OROPTMODEL_**

The OPTMODEL procedure always creates and initializes a SAS macro variable called _OROPTMODEL_, which contains a character string. After each PROC OPTMODEL run, you can examine this macro variable by specifying `put &_OROPTMODEL_;` and check the execution of the most recently invoked solver from the value of the macro variable. After the NLP solver is called, the various terms of the variable are interpreted as follows:

**STATUS**
indicates the solver status at termination. It can take one of the following values:

- **OK** The solver terminated normally.
- **SYNTAX_ERROR** The use of syntax is incorrect.
- **DATA_ERROR** The input data are inconsistent.
- **OUT_OF_MEMORY** Insufficient memory was allocated to the procedure.
- **IO_ERROR** A problem in reading or writing of data has occurred.
- **SEMANTIC_ERROR** An evaluation error, such as an invalid operand type, has occurred.
- **ERROR** The status cannot be classified into any of the preceding categories.

**ALGORITHM**
indicates the algorithm that produced the solution data in the macro variable. This term only appears when STATUS=OK. It can take one of the following values:

- **IP** The interior point algorithm produced the solution data.
- **AS** The active-set algorithm produced the solution data.

When running algorithms concurrently (ALGORITHM=CONCURRENT), this term indicates which algorithm was the first to terminate.

**SOLUTION_STATUS**
indicates the solution status at termination. It can take one of the following values:

- **OPTIMAL** The solution is optimal.
- **CONDITIONAL_OPTIMAL** The optimality of the solution cannot be proven.
BEST_FEASIBLE

The solution returned is the best feasible solution. This solution status indicates that the algorithm has converged to a local optimum but a feasible (not locally optimal) solution with a better objective value has been found and hence is returned.

INFEASIBLE

The problem is infeasible.

UNBOUNDED

The problem might be unbounded.

INFEASIBLE_OR_UNBOUNDED

The problem is infeasible or unbounded.

BAD_PROBLEM_TYPE

The problem type is not supported by the solver.

ITERATION_LIMIT_REACHED

The maximum allowable number of iterations has been reached.

TIME_LIMIT_REACHED

The solver reached its execution time limit.

FAILED

The solver failed to converge, possibly due to numerical issues.

OBJECTIVE

indicates the objective value that is obtained by the solver at termination.

NUMSTARTS

indicates the number of starting points. This term appears only in multistart mode.

SAMPLE_POINTS

indicates the number of points that are evaluated in the sampling phase. This term appears only in multistart mode.

DISTINCT_OPTIMA

indicates the number of distinct local optima that the solver finds. This term appears only in multistart mode.

SEED

indicates the seed value that is used to initialize the solver. This term appears only in multistart mode.

INFEASIBILITY

indicates the level of infeasibility of the constraints at the solution.

OPTIMALITY_ERROR

indicates the norm of the optimality conditions at the solution. See the section “Solver Termination Criterion” on page 799 for details.

ITERATIONS

indicates the number of iterations required to solve the problem.

PRESOLVE_TIME

indicates the real time taken for preprocessing (seconds).

SOLUTION_TIME

indicates the real time taken by the solver to perform iterations for solving the problem (seconds).

NOTE: The time that is reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.
Example 16.1: Solving Highly Nonlinear Optimization Problems

This example demonstrates the use of the NLP solver to solve the following highly nonlinear optimization problem, which appears in Hock and Schittkowski (1981):

\[
\begin{align*}
\text{minimize} & \quad f(x) = 0.4(x_1/x_7)^{0.67} + 0.4(x_2/x_8)^{0.67} + 10 - x_1 - x_2 \\
\text{subject to} & \quad 1 - 0.0588x_5x_7 - 0.1x_1 \geq 0 \\
& \quad 1 - 0.0588x_6x_8 - 0.1x_1 - 0.1x_2 \geq 0 \\
& \quad 1 - 4x_3/x_5 - 2/(x_3^{0.71}x_5) - 0.0588x_7/x_3^{1.3} \geq 0 \\
& \quad 1 - 4x_4/x_6 - 2/(x_4^{0.71}x_6) - 0.0588x_8/x_4^{1.3} \geq 0 \\
& \quad 0.1 \leq f(x) \leq 4.2 \\
& \quad 0.1 \leq x_i \leq 10, i = 1, 2, \ldots, 8
\end{align*}
\]

The initial point used is \( x^0 = (6, 3, 0.4, 0.2, 6, 6, 1, 0.5) \). You can call the NLP solver within PROC OPTMODEL to solve the problem by writing the following SAS statements:

```sas
proc optmodel;
  var x{1..8} >= 0.1 <= 10;
  min f = 0.4*(x[1]/x[7])^0.67 + 0.4*(x[2]/x[8])^0.67 + 10 - x[1] - x[2];
  con c1: 1 - 0.0588*x[5]*x[7] - 0.1*x[1] >= 0;
  con c2: 1 - 0.0588*x[6]*x[8] - 0.1*x[1] - 0.1*x[2] >= 0;
  con c5: 0.1 <= f <= 4.2;
/* starting point */
  x[1] = 6;
  x[2] = 3;
  x[3] = 0.4;
  x[4] = 0.2;
  x[5] = 6;
  x[6] = 6;
  x[7] = 1;
  x[8] = 0.5;
  solve with nlp / algorithm=activeset;
  print x;
quit;
```

The summaries and the solution are shown in Output 16.1.1.
### Output 16.1.1  Summaries and the Returned Solution

#### The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
<td>Minimization</td>
</tr>
<tr>
<td>Objective Function</td>
<td>f</td>
</tr>
<tr>
<td>Objective Type</td>
<td>Nonlinear</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Variables</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded Above</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
<td>8</td>
</tr>
<tr>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>Fixed</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Constraints</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear LE (&lt;=)</td>
<td>0</td>
</tr>
<tr>
<td>Linear EQ (=)</td>
<td>0</td>
</tr>
<tr>
<td>Linear GE (&gt;=)</td>
<td>0</td>
</tr>
<tr>
<td>Linear Range</td>
<td>0</td>
</tr>
<tr>
<td>Nonlinear LE (&lt;=)</td>
<td>0</td>
</tr>
<tr>
<td>Nonlinear EQ (=)</td>
<td>0</td>
</tr>
<tr>
<td>Nonlinear GE (&gt;=)</td>
<td>4</td>
</tr>
<tr>
<td>Nonlinear Range</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>NLP</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Active Set</td>
</tr>
<tr>
<td>Objective Function</td>
<td>f</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Best Feasible</td>
</tr>
<tr>
<td>Objective Value</td>
<td>3.9511579677</td>
</tr>
<tr>
<td>Optimality Error</td>
<td>0.0001050714</td>
</tr>
<tr>
<td>Infeasibility</td>
<td>7.7302351E-7</td>
</tr>
</tbody>
</table>

| Iterations | 26 |
| Presolve Time | 0.00 |
| Solution Time | 0.00 |

<table>
<thead>
<tr>
<th>[1]</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6.46332</td>
</tr>
<tr>
<td>2</td>
<td>2.23453</td>
</tr>
<tr>
<td>3</td>
<td>0.66746</td>
</tr>
<tr>
<td>4</td>
<td>0.59582</td>
</tr>
<tr>
<td>5</td>
<td>5.93298</td>
</tr>
<tr>
<td>6</td>
<td>5.52723</td>
</tr>
<tr>
<td>7</td>
<td>1.01379</td>
</tr>
<tr>
<td>8</td>
<td>0.40066</td>
</tr>
</tbody>
</table>
Example 16.2: Solving Unconstrained and Bound-Constrained Optimization Problems

Although the NLP techniques are suited for solving generally constrained nonlinear optimization problems, these techniques can also be used to solve unconstrained and bound-constrained problems efficiently. This example considers the relatively large nonlinear optimization problems

\[
\text{minimize } f(x) = \sum_{i=1}^{n-1} (-4x_i + 3.0) + \sum_{i=1}^{n-1} (x_i^2 + x_n^2)^2
\]

and

\[
\text{minimize } f(x) = \sum_{i=1}^{n-1} \cos(-.5x_{i+1} - x_i^2)
\]

subject to \( 1 \leq x_i \leq 2, \ i = 1, \ldots, n \)

with \( n = 100,000 \). These problems are unconstrained and bound-constrained, respectively.

For large-scale problems, the default memory limit might be too small, which can lead to out-of-memory status. To prevent this occurrence, it is recommended that you set a larger memory size. See the section “Memory Limit” on page 21 in Chapter 4, “Shared Concepts,” for more information.

To solve the first problem, you can write the following statements:

```plaintext
proc optmodel;
   number N=100000;
   var x{1..N} init 1.0;

   minimize f = sum {i in 1..N - 1} (-4 * x[i] + 3.0) +
       sum {i in 1..N - 1} (x[i]^2 + x[N]^2)^2;

   solve with nlp;
quit;
```

The problem and solution summaries are shown in Output 16.2.1.
To solve the second problem, you can write the following statements (here the active-set method is specifically selected):

```plaintext
proc optmodel;
  number N=100000;
  var x{1..N} >= 1 <= 2;
  minimize f = sum {i in 1..N - 1} cos(-0.5*x[i+1] - x[i]^2);
  solve with nlp / algorithm=activeset;
quit;
```

The problem and solution summaries are shown in **Output 16.2.2**.
Some constraints have both lower and upper bounds (that is, $a \leq g(x) \leq b$). These constraints are called *range constraints*. The NLP solver can handle range constraints in an efficient way. Consider the following NLP problem, taken from Hock and Schittkowski (1981),

$$\begin{align*}
\text{minimize} & \quad f(x) = 5.35(x_3)^2 + 0.83x_1x_5 + 37.29x_1 - 40792.141 \\
\text{subject to} & \quad 0 \leq a_1 + a_2x_2x_5 + a_3x_1x_4 - a_4x_3x_5 \leq 92 \\
& \quad 0 \leq a_5 + a_6x_2x_5 + a_7x_1x_2 + a_8x_3^2 - 90 \leq 20 \\
& \quad 0 \leq a_9 + a_{10}x_3x_5 + a_{11}x_1x_3 + a_{12}x_3x_4 - 20 \leq 5 \\
& \quad 78 \leq x_1 \leq 102 \\
& \quad 33 \leq x_2 \leq 45 \\
& \quad 27 \leq x_i \leq 45, \quad i = 3, 4, 5
\end{align*}$$

where the values of the parameters $a_i, i = 1, 2, \ldots, 12$, are shown in Table 16.7.
The initial point used is $x^0 = (78, 33, 27, 27, 27)$. You can call the NLP solver within PROC OPTMODEL to solve this problem by writing the following statements:

```plaintext
proc optmodel;
    number l {1..5} = [78 33 27 27 27];
    number u {1..5} = [102 45 45 45 45];

    number a {1..12} =
        [85.334407 0.0056858 0.0006262 0.0022053 80.51249 0.0071317 0.0029955 0.0021813 9.300961 0.0047026 0.0012547 0.0019085];

    var x {j in 1..5} >= l[j] <= u[j];

    minimize f = 5.35*x[3]^2 + 0.83*x[1]*x[5] + 37.29*x[1] - 40792.141;

    con constr1:
    con constr2:
    con constr3:

    x[1] = 78;
    x[2] = 33;
    x[3] = 27;
    x[4] = 27;
    x[5] = 27;

    solve with nlp / algorithm=activeset;
    print x;
quit;
```

The summaries and solution are shown in Output 16.3.1.
Output 16.3.1  Summaries and the Optimal Solution

**The OPTMODEL Procedure**

<table>
<thead>
<tr>
<th>Problem Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
<td>Minimization</td>
</tr>
<tr>
<td>Objective Function</td>
<td>f</td>
</tr>
<tr>
<td>Objective Type</td>
<td>Quadratic</td>
</tr>
</tbody>
</table>

| Number of Variables | 5 |
| Bounded Above       | 0 |
| Bounded Below       | 0 |
| Bounded Below and Above | 5 |
| Free               | 0 |
| Fixed              | 0 |

| Number of Constraints | 3 |
| Linear LE (≤)        | 0 |
| Linear EQ (=)        | 0 |
| Linear GE (≥)        | 0 |
| Linear Range         | 0 |
| Nonlinear LE (≤)     | 0 |
| Nonlinear EQ (=)     | 0 |
| Nonlinear GE (≥)     | 0 |
| Nonlinear Range      | 3 |

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>NLP</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Active Set</td>
</tr>
<tr>
<td>Objective Function</td>
<td>f</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Best Feasible</td>
</tr>
<tr>
<td>Objective Value</td>
<td>-30689.17758</td>
</tr>
</tbody>
</table>

| Optimality Error | 0.0000100695 |
| Infeasibility    | 2.2694156E-8 |

| Iterations | 17 |
| Presolve Time | 0.00 |
| Solution Time | 0.00 |

<table>
<thead>
<tr>
<th>x</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>78.000</td>
</tr>
<tr>
<td>2</td>
<td>33.000</td>
</tr>
<tr>
<td>3</td>
<td>29.995</td>
</tr>
<tr>
<td>4</td>
<td>45.000</td>
</tr>
<tr>
<td>5</td>
<td>36.776</td>
</tr>
</tbody>
</table>
Example 16.4: Solving Large-Scale NLP Problems

The following example is a selected large-scale problem from the CUTEr test set (Gould, Orban, and Toint 2003) that has 20,400 variables, 20,400 lower bounds, and 9,996 linear equality constraints. This problem was selected to provide an idea of the size of problem that the NLP solver is capable of solving. In general, the maximum size of nonlinear optimization problems that can be solved with the NLP solver is controlled less by the number of variables and more by the density of the first and second derivatives of the nonlinear objective and constraint functions.

For large-scale problems, the default memory limit might be too small, which can lead to out-of-memory status. To prevent this occurrence, it is recommended that you set a larger memory size. See the section “Memory Limit” on page 21 in Chapter 4, “Shared Concepts,” for more information.

```
proc optmodel;
num nx = 100;
num ny = 100;

var x {1..nx, 0..ny+1} >= 0;
var y {0..nx+1, 1..ny} >= 0;

min f =
    sum {i in 1..nx-1, j in 1..ny-1} (x[i,j] - 1)^2
    + sum {i in 1..nx-1, j in 1..ny-1} (y[i,j] - 1)^2
    + sum {i in 1..nx-1} (x[i,ny] - 1)^2
    + sum {j in 1..ny-1} (y[nx,j] - 1)^2
    ) / 2;

con con1 {i in 2..nx-1, j in 2..ny-1}:
    (x[i,j] - x[i-1,j]) + (y[i,j] - y[i,j-1]) = 1;
con con2 {i in 2..nx-1}:
    x[i,0] + (x[i,1] - x[i-1,1]) + y[i,1] = 1;
con con3 {i in 1..nx-1}:
    x[i,ny+1] + (x[i,ny] - x[i-1,ny]) - y[i,ny-1] = 1;
con con4 {j in 2..ny-1}:
    y[0,j] + (y[1,j] - y[1,j-1]) + x[1,j] = 1;
con con5 {j in 2..ny-1}:
    y[nx+1,j] + (y[nx,j] - y[nx,j-1]) - x[nx-1,j] = 1;

for {i in 1..nx-1} x[i,ny].lb = 1;
for {j in 1..ny-1} y[nx,j].lb = 1;

solve with nlp;
quit;
```

The problem and solution summaries are shown in Output 16.4.1.
Example 16.5: Solving NLP Problems That Have Several Local Minima

Some NLP problems contain many local minima. By default, the NLP solver converges to a single local minimum. However, the NLP solver can search the feasible region for other local minima. After it completes the search, it returns the point where the objective function achieves its minimum value. (This point might not be a local minimum; see the SOLTYPE= option for more details.) Consider the following example, taken from Hock and Schittkowski (1981):
minimize \( f(x) = (x_1 - 1)^2 + (x_1 - x_2)^2 + (x_2 - x_3)^3 + (x_3 - x_4)^4 + (x_4 - x_5)^4 \)
subject to \[
\begin{align*}
    x_1 + x_2^2 + x_3^3 &= 2 + 3\sqrt{2} \\
    x_2 + x_4^2 - x_3^3 &= -2 + 2\sqrt{2} \\
    x_1x_5 &= 2 \\
    -5 \leq x_i \leq 5, & i = 1, \ldots, 5
\end{align*}
\]

The following statements call the NLP solver to search the feasible region for different local minima. The NTHREADS= option in the SOLVE statement requests that the multistart algorithm use up to four threads. The SEED= option is specified for reproducibility, but it is not required in running the multistart algorithm.

```sas
proc optmodel;
    var x{i in 1..5} >= -5 <= 5 init -2;
    con g1: x[1] + x[2]^2 + x[3]^3 = 2 + 3*sqrt(2);
    con g3: x[1]*x[5] = 2;
    solve with nlp/multistart=(maxstarts=10) seed=1234 nthreads=4;
    print x.msinit x;
quit;
```

The PRINT statement prints the starting point (x.msinit) that led to the best local solution and the best local solution (x) that the NLP solver found in multistart mode. The SAS log is shown in Output 16.5.1.
### Example 16.5: Solving NLP Problems That Have Several Local Minima

Output 16.5.1: Progress of the Algorithm as Shown in the Log

<table>
<thead>
<tr>
<th>Start</th>
<th>Best Objective</th>
<th>Local Optimality</th>
<th>Objective Error</th>
<th>Infeasibility</th>
<th>Local Iter</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>607.035512</td>
<td>607.035512</td>
<td>2.81991E-7</td>
<td>5.88394E-9</td>
<td>7</td>
<td>Optimal</td>
</tr>
<tr>
<td>2</td>
<td>52.9025715</td>
<td>52.9025715</td>
<td>1.19566E-7</td>
<td>9.15988E-8</td>
<td>7</td>
<td>Optimal</td>
</tr>
<tr>
<td>3</td>
<td>52.9025715</td>
<td>607.0355801</td>
<td>8.62585E-7</td>
<td>8.62585E-7</td>
<td>7</td>
<td>Optimal</td>
</tr>
<tr>
<td>4</td>
<td>52.9025715</td>
<td>52.9025732</td>
<td>2.36966E-7</td>
<td>6.49828E-8</td>
<td>12</td>
<td>Optimal</td>
</tr>
<tr>
<td>5 *</td>
<td>52.9025715</td>
<td>607.035845</td>
<td>5.38983E-7</td>
<td>5.38983E-7</td>
<td>11</td>
<td>Optimal</td>
</tr>
<tr>
<td>6</td>
<td>27.8719058</td>
<td>27.8719058</td>
<td>7.17391E-7</td>
<td>1.85419E-7</td>
<td>6</td>
<td>Optimal</td>
</tr>
<tr>
<td>7</td>
<td>27.8719058</td>
<td>27.8719059</td>
<td>5E-7</td>
<td>7.34792E-8</td>
<td>6</td>
<td>Optimal</td>
</tr>
<tr>
<td>8</td>
<td>27.8719058</td>
<td>64.8740102</td>
<td>5E-7</td>
<td>3.27268E-7</td>
<td>7</td>
<td>Optimal</td>
</tr>
<tr>
<td>9</td>
<td>0.0293107</td>
<td>0.0293107</td>
<td>5.0539E-7</td>
<td>5.0539E-7</td>
<td>9</td>
<td>Optimal</td>
</tr>
<tr>
<td>10</td>
<td>0.0293107</td>
<td>0.02931081</td>
<td>5.59562E-7</td>
<td>5.59562E-7</td>
<td>13</td>
<td>Optimal</td>
</tr>
</tbody>
</table>

The first column in the log indicates the index of the current starting point. An additional indicator (*) can appear after the index to provide more information about the optimization run that started from the current point. For more information, see the section “Iteration Log for Multistart” on page 798. The second column records the best objective that has been found so far. Columns 3 to 6 report the objective value, optimality error, infeasibility, and number of iterations that the local solver returned when it was started from the current starting point. Finally, the last column records the status of the local solver—namely, whether it was able to converge to a local optimum from the current starting point.

The summaries and solution are shown in Output 16.5.2. Note that the best local solution was found by starting the local solver from a point at x.msinit.
Output 16.5.2  Summaries and the Optimal Solution

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
</tbody>
</table>

| Number of Variables  | 5                |
| Bounded Above        | 0                |
| Bounded Below        | 0                |
| Bounded Below and Above | 5             |
| Free                 | 0                |
| Fixed                | 0                |

| Number of Constraints | 3                |
| Linear LE (<=)        | 0                |
| Linear EQ (=)         | 0                |
| Linear GE (>=)        | 0                |
| Linear Range          | 0                |
| Nonlinear LE (<=)     | 0                |
| Nonlinear EQ (=)      | 3                |
| Nonlinear GE (>=)     | 0                |
| Nonlinear Range       | 0                |

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
</tbody>
</table>

| Number of Starts      | 10               |
| Number of Sample Points | 800             |
| Number of Distinct Optima | 6              |
| Random Seed Used      | 1234             |
| Optimality Error      | 8.2635936E-7     |
| Infeasibility         | 2.5864754E-7     |

| Presolve Time         | 0.00             |
| Solution Time         | 0.02             |

<table>
<thead>
<tr>
<th>[1] x.MSINIT x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1  2.87632  1.1167</td>
</tr>
<tr>
<td>2  3.26949  1.2205</td>
</tr>
<tr>
<td>3  -1.32653  1.5378</td>
</tr>
<tr>
<td>4  -0.49448  1.9726</td>
</tr>
<tr>
<td>5  1.27877  1.7910</td>
</tr>
</tbody>
</table>
Alternatively, the following SAS statements show how you can solve this problem in a distributed computing environment. After you create a CAS session named mysess, you specify the SESSREF="MYSESS" option to run the OPTMODEL procedure on CAS. For more information about creating a CAS session, see the section “Using CAS Sessions and CAS Engine Librefs” on page 19 in Chapter 4, “Shared Concepts.” Note that the NTHREADS= option in PROC OPTMODEL is used to specify the number of threads to use on each CAS node.

    proc optmodel sessref="mysess" nthreads=16;
    var x{i in 1..5} >= -5 <= 5 init -2;
    con g1: x[1] + x[2]^2 + x[3]^3 = 2 + 3*sqrt(2);
    con g3: x[1]*x[5] = 2;
    solve with nlp/multistart=(maxstarts=200) seed=12345;
    print x;
    quit;

The SAS log is displayed in Output 16.5.3. It shows that 120 worker nodes with 16 threads on each node are used in distributed mode.
Output 16.5.3 Progress of the Algorithm as Shown in the Log

NOTE: Problem generation will use 4 threads.
NOTE: The problem has 5 variables (0 free, 0 fixed).
NOTE: The problem has 0 linear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 3 nonlinear constraints (0 LE, 3 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver removed 0 variables, 0 linear constraints, and 0 nonlinear constraints.
NOTE: The SOLVE statement is executing in the distributed computing environment with 120 worker nodes.
NOTE: Using analytic derivatives for objective.
NOTE: Using analytic derivatives for nonlinear constraints.
NOTE: The NLP solver is called.
NOTE: The Interior Point algorithm is used.
NOTE: The MULTISTART option is enabled.
NOTE: The nondeterministic parallel mode is enabled.
NOTE: The Multistart algorithm is executing in the distributed computing environment with 120 worker nodes.
NOTE: The Multistart algorithm is using up to 16 threads.
WARNING: The solution of the current run may not be reproducible even if you specify SEED=12345 in a subsequent SOLVE statement.

<table>
<thead>
<tr>
<th>Start</th>
<th>Objective</th>
<th>Objective</th>
<th>Error</th>
<th>Infeasibility</th>
<th>Iters</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>607.035511</td>
<td>607.035511</td>
<td>2.70665E-7</td>
<td>7.21823E-9</td>
<td>9</td>
<td>Optimal</td>
</tr>
<tr>
<td>2</td>
<td>607.035511</td>
<td>607.035673</td>
<td>8.29095E-7</td>
<td>8.02736E-7</td>
<td>8</td>
<td>Optimal</td>
</tr>
<tr>
<td>3</td>
<td>607.035383</td>
<td>607.035383</td>
<td>5E-7</td>
<td>2.80981E-7</td>
<td>6</td>
<td>Optimal</td>
</tr>
<tr>
<td>4</td>
<td>52.9025752</td>
<td>52.9025752</td>
<td>1.32438E-7</td>
<td>1.32438E-7</td>
<td>7</td>
<td>Optimal</td>
</tr>
<tr>
<td>5</td>
<td>52.9025752</td>
<td>52.902665</td>
<td>8.6181E-7</td>
<td>8.6181E-7</td>
<td>6</td>
<td>Optimal</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>194</td>
<td>0.02931082</td>
<td>52.9025778</td>
<td>9.74497E-8</td>
<td>2.49519E-8</td>
<td>7</td>
<td>Optimal</td>
</tr>
<tr>
<td>195</td>
<td>0.02931082</td>
<td>64.873964</td>
<td>5.17354E-7</td>
<td>5.17354E-7</td>
<td>10</td>
<td>Optimal</td>
</tr>
<tr>
<td>196</td>
<td>0.02931082</td>
<td>52.9025666</td>
<td>8.48688E-7</td>
<td>8.48688E-7</td>
<td>6</td>
<td>Optimal</td>
</tr>
<tr>
<td>197</td>
<td>0.02931082</td>
<td>52.902585</td>
<td>1.30951E-7</td>
<td>1.30951E-7</td>
<td>8</td>
<td>Optimal</td>
</tr>
<tr>
<td>198</td>
<td>0.02931082</td>
<td>64.8739864</td>
<td>1.07745E-7</td>
<td>1.07745E-7</td>
<td>6</td>
<td>Optimal</td>
</tr>
<tr>
<td>199</td>
<td>0.02931082</td>
<td>52.902589</td>
<td>5E-7</td>
<td>4.19274E-7</td>
<td>8</td>
<td>Optimal</td>
</tr>
<tr>
<td>200</td>
<td>0.02931082</td>
<td>52.9025419</td>
<td>5E-7</td>
<td>3.34178E-7</td>
<td>5</td>
<td>Optimal</td>
</tr>
</tbody>
</table>

NOTE: The Multistart algorithm generated 192000 sample points.
NOTE: 168 distinct local optima were found.
NOTE: The best objective value found by local solver = 0.0293108196.
NOTE: The solution found by local solver with objective = 0.0293108196 was returned.

Output 16.5.4 shows the summaries and solution.
### Output 16.5.4 Summaries and the Optimal Solution

#### The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
<tr>
<td>Number of Variables</td>
</tr>
<tr>
<td>Bounded Above</td>
</tr>
<tr>
<td>Bounded Below</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
</tr>
<tr>
<td>Free</td>
</tr>
<tr>
<td>Fixed</td>
</tr>
</tbody>
</table>

| Number of Constraints    | 3            |
| Linear LE (<=)           | 0            |
| Linear EQ (=)            | 0            |
| Linear GE (>=)           | 0            |
| Linear Range             | 0            |
| Nonlinear LE (<=)        | 0            |
| Nonlinear EQ (=)         | 3            |
| Nonlinear GE (>=)        | 0            |
| Nonlinear Range          | 0            |

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
</tbody>
</table>

| Number of Starts         | 200           |
| Number of Sample Points  | 192000        |
| Number of Distinct Optima| 168           |
| Random Seed Used         | 12345         |
| Optimality Error         | 9.0123528E-8  |
| Infeasibility            | 9.0123528E-8  |

| Presolve Time            | 0.00          |
| Solution Time            | 0.08          |

\[
\begin{array}{c|c}
[1] & x \\ 
1  & 1.1166 \\ 
2  & 1.2204 \\ 
3  & 1.5378 \\ 
4  & 1.9728 \\ 
5  & 1.7911 \\
\end{array}
\]
Example 16.6: Maximum Likelihood Weibull Estimation

The following data are taken from Lawless (1982, p. 193) and represent the number of days that it took rats that were painted with a carcinogen to develop carcinoma. The last two observations are censored data from a group of 19 rats.

```
data pike;
  input days cens @@;
datalines;
  143 0 164 0 188 0 188 0
  190 0 192 0 206 0 209 0
  213 0 216 0 220 0 227 0
  230 0 234 0 246 0 265 0
  304 0 216 1 244 1;
```

Suppose you want to compute the maximum likelihood estimates of the scale parameter \( \sigma \) (\( \alpha \) in Lawless), the shape parameter \( c \) (\( \beta \) in Lawless), and the location parameter \( \theta \) (\( \mu \) in Lawless). The observed likelihood function of the three-parameter Weibull transformation (Lawless 1982, p. 191) is

\[
L(\theta, \sigma, c) = \frac{c^n}{\sigma^m} \prod_{i \in D} \left( \frac{t_i - \theta}{\sigma} \right)^{c-1} \prod_{i=1}^{n} \exp \left( -\frac{(t_i - \theta)}{\sigma} \right)^c
\]

where \( n \) is the number of individuals involved in the experiment, \( D \) is the set of individuals whose lifetimes are observed, \( m = |D| \), and \( t_i \) is defined by the data set. Then the log-likelihood function is

\[
l(\theta, \sigma, c) = m \log c - mc \log \sigma + (c - 1) \sum_{i \in D} \log(t_i - \theta) - \sum_{i=1}^{n} \left( \frac{t_i - \theta}{\sigma} \right)^c
\]

For \( c < 1 \), the logarithmic terms become infinite as \( \theta \uparrow \min_{i \in D}(t_i) \). That is, \( l(\theta, \sigma, c) \) is unbounded. Thus our interest is restricted to \( c \) values greater than or equal to 1. Further, for the logarithmic terms to be defined, it is required that \( \sigma > 0 \) and \( \theta < \min_{i \in D}(t_i) \).

The following PROC OPTMODEL call specifies the maximization of the log-likelihood function for the three-parameter Weibull estimation:

```
proc optmodel;
  set OBS;
  num days {OBS};
  num cens {OBS};
  read data pike into OBS=[_N_] days cens;
  var sig >= 1.0e-6 init 10;
  var c >= 1.0e-6 init 10;
  var theta >= 0 <= min {i in OBS: cens[i] = 0} days[i] init 10;
  impvar fi {i in OBS} =
    (if cens[i] = 0 then
      log(c) - c * log(sig) + (c - 1) * log(days[i] - theta)
    )
    - ((days[i] - theta) / sig)^c;
```
max logf = sum {i in OBS} fi[i];

set VARS = 1.._NVAR_;
num mycov {i in VARS, j in 1..i};
solve with NLP / algorithm=activeset covest=(cov=2 covout=mycov);
print sig c theta;
print mycov;
create data covdata from [i j]={(i in VARS, j in 1..i)
v_i=VAR_[i].name v_j=VAR_[j].name mycov;
quit;
The solution is displayed in Output 16.6.1. The solution that the NLP solver obtains closely matches the local maximum $\theta^* = 122$, $\sigma^* = 108.4$, and $c^* = 2.712$ that are given in Lawless (1982, p. 193).

Output 16.6.1 Three-Parameter Weibull Estimation Results

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
<tr>
<td>Number of Variables</td>
</tr>
<tr>
<td>Bounded Above</td>
</tr>
<tr>
<td>Bounded Below</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
</tr>
<tr>
<td>Free</td>
</tr>
<tr>
<td>Fixed</td>
</tr>
<tr>
<td>Number of Constraints</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Optimality Error</td>
</tr>
<tr>
<td>Infeasibility</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sig</th>
<th>c</th>
<th>theta</th>
</tr>
</thead>
<tbody>
<tr>
<td>107.41</td>
<td>2.6802</td>
<td>122.94</td>
</tr>
</tbody>
</table>
Example 16.7: Finding an Irreducible Infeasible Set

This example demonstrates the use of the IIS= option to locate an irreducible infeasible set. Suppose you have the following nonlinear programming problem:

\[
\begin{align*}
\text{minimize} & \quad x_1^4 + x_2^4 + x_3^4 \\
\text{subject to} & \quad x_1 + x_2 \geq 10 \quad (c1) \\
& \quad x_1 + x_3 \leq 4 \quad (c2) \\
& \quad 4 \leq x_2 + x_3 \leq 5 \quad (c3) \\
& \quad x_1^2 + x_3 \leq 5 \quad (c4) \\
& \quad x_1, x_2 \geq 0 \quad (c5) \\
& \quad 0 \leq x_3 \leq 3
\end{align*}
\]

It is easy to verify that the following three linear constraints and one variable bound form an IIS for this problem:

\[
\begin{align*}
x_1 + x_2 & \geq 10 \quad (c1) \\
x_1 + x_3 & \leq 4 \quad (c2) \\
x_2 + x_3 & \leq 5 \quad (c3) \\
x_3 & \geq 0
\end{align*}
\]

You can formulate the problem and call the NLP solver by using the following statements:

```plaintext
proc optmodel;
   /* declare variables */
   var x{1..3} >= 0;

   /* upper bound on variable x[3] */
   x[3].ub = 3;

   /* objective function */

   /* constraints */
   con c4: x[1]^2 + x[3] <= 5;

   solve with nlp / iis = true;

   print x.status;
```

---

**Output 16.6.1 continued**

<table>
<thead>
<tr>
<th>mycov</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1137.2528</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>32.0116</td>
<td>1.2259</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-940.9583</td>
<td>-28.3624</td>
<td>868.4785</td>
</tr>
</tbody>
</table>
Example 16.7: Finding an Irreducible Infeasible Set

```
print c1.status c2.status c3.status;
expand / IIS;
quit;
```

The SAS log output is shown in Output 16.7.1. Note that the PROC OPTMODEL presolver is disabled because the IIS= option is enabled. Also, a warning message is displayed to alert the user that the nonlinear constraints are ignored for the purpose of detecting an IIS.

**Output 16.7.1** Finding an IIS: Original Problem

<table>
<thead>
<tr>
<th>Phase</th>
<th>Iteration</th>
<th>Value</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>1</td>
<td>6.000000E+00</td>
<td>0</td>
</tr>
<tr>
<td>P</td>
<td>3</td>
<td>9.998343E-01</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The OPTMODEL presolver is disabled when the IIS= option is enabled.

NOTE: Problem generation will use 4 threads.

NOTE: The problem has 3 variables (0 free, 0 fixed).

NOTE: The problem has 3 linear constraints (1 LE, 0 EQ, 1 GE, 1 range).

NOTE: The problem has 6 linear constraint coefficients.

NOTE: The problem has 1 nonlinear constraints (1 LE, 0 EQ, 0 GE, 0 range).

WARNING: The nonlinear constraints are ignored because the IIS= option is enabled.

NOTE: The NLP solver is called.

NOTE: The LP solver is called.

NOTE: The IIS= option is enabled.

| Objective
<table>
<thead>
<tr>
<th>Phase</th>
<th>Iteration</th>
<th>Value</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P 1</td>
<td>1</td>
<td>6.000000E+00</td>
<td>0</td>
</tr>
<tr>
<td>P 1</td>
<td>3</td>
<td>9.998343E-01</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Applying the IIS sensitivity filter.

NOTE: The sensitivity filter removed 1 constraints and 3 variable bounds.

NOTE: Applying the IIS deletion filter.

NOTE: Processing constraints.

<table>
<thead>
<tr>
<th>Processed</th>
<th>Removed</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Processing variable bounds.

<table>
<thead>
<tr>
<th>Processed</th>
<th>Removed</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The deletion filter removed 0 constraints and 0 variable bounds.

NOTE: The IIS= option found this problem to be infeasible.

NOTE: The IIS= option found an irreducible infeasible set with 1 variables and 3 constraints.

NOTE: The IIS solve time is 0.00 seconds.

There are two ways to display the rows and columns that are included in the IIS. One way is to use the PRINT statement to print the value of the .status suffix for each variable and constraint. The more straightforward approach is to use the EXPAND statement with the IIS option.
The “Solution Summary” table and the output of both approaches appear in Output 16.7.2.

**Output 16.7.2** Solution Summary and PRINT Statement Output

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Iterations</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iterations2</td>
<td>0</td>
</tr>
<tr>
<td>Presolve Time</td>
<td>0.00</td>
</tr>
<tr>
<td>Solution Time</td>
<td>0.00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[1] x.STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>c1.STATUS c2.STATUS c3.STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
</tr>
</tbody>
</table>

The SAS log output for the modified problem is shown in Output 16.7.3.

The “Solution Summary” table shows that the problem is infeasible. As you can see, the lower bound of variable \( x_3 \), the lower bound of constraint \( c_1 \), and the upper bounds of constraints \( c_2 \) and \( c_3 \) form an IIS.

Making any of the components in the preceding IIS nonbinding removes the infeasibility from the IIS. Because there could be multiple IISs, you would want to remove the infeasibility from the preceding IIS and call the NLP solver with the IIS= option enabled again to see whether there is any other IIS. The following statements show how to modify the original PROC OPTMODEL statements to set the upper bound of constraint \( c_3 \) to infinity, represented by CONSTANT(‘BIG’), and invoke the NLP IIS detection:

```latex
/* relax upper bound on constraint c3 */
c3.ub = constant('BIG');

solve with nlp / iis = true;

print x.status;
print c1.status c2.status c3.status;
```

The SAS log output for the modified problem is shown in **Output 16.7.3**.
Example 16.7: Finding an Irreducible Infeasible Set

Output 16.7.3 Finding an IIS: Modified Problem

NOTE: The OPTMODEL presolver is disabled when the IIS= option is enabled.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3 variables (0 free, 0 fixed).
NOTE: The problem has 3 linear constraints (1 LE, 0 EQ, 2 GE, 0 range).
NOTE: The problem has 6 linear constraint coefficients.
NOTE: The problem has 1 nonlinear constraints (1 LE, 0 EQ, 0 GE, 0 range).
WARNING: The nonlinear constraints are ignored because the IIS= option is enabled.
NOTE: The NLP solver is called.
NOTE: The LP solver is called.
NOTE: The IIS= option is enabled.

<table>
<thead>
<tr>
<th>Objective</th>
<th>Phase</th>
<th>Iteration</th>
<th>Value</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>P 1</td>
<td>1</td>
<td>1.400000E+01</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>P 1</td>
<td>2</td>
<td>0.000000E+00</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The IIS= option found this problem to be feasible.
NOTE: The IIS solve time is 0.00 seconds.

The “Solution Summary” table and the output of the PRINT statements appear in Output 16.7.4. As you can see, both the variable status and constraint status tables are empty. There is no other IIS, and the problem becomes feasible.

Output 16.7.4 Solution Summary and PRINT Statement Output

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
</tbody>
</table>

| Iterations | 2  |
| Iterations2 | 0  |
| Presolve Time | 0.00 |
| Solution Time  | 0.00 |

[1] x.STATUS
1
2
3

c1.STATUS c2.STATUS c3.STATUS
References


Overview: QP Solver

The OPTMODEL procedure provides a framework for specifying and solving quadratic programs.

Mathematically, a quadratic programming (QP) problem can be stated as follows:

\[
\begin{align*}
\min & \quad \frac{1}{2} x^T Q x + c^T x \\
\text{subject to} & \quad A x \{\geq, =, \leq\} b \\
& \quad 1 \leq x \leq u
\end{align*}
\]

where

- \( Q \in \mathbb{R}^{n \times n} \) is the quadratic (also known as Hessian) matrix
- \( A \in \mathbb{R}^{m \times n} \) is the constraints matrix
- \( x \in \mathbb{R}^n \) is the vector of decision variables
- \( c \in \mathbb{R}^n \) is the vector of linear objective function coefficients
- \( b \in \mathbb{R}^m \) is the vector of constraints right-hand sides (RHS)
- \( l \in \mathbb{R}^n \) is the vector of lower bounds on the decision variables
- \( u \in \mathbb{R}^n \) is the vector of upper bounds on the decision variables
The quadratic matrix \( Q \) is assumed to be symmetric; that is,

\[
q_{ij} = q_{ji}, \quad \forall i, j = 1, \ldots, n
\]

Indeed, it is easy to show that even if \( Q \neq Q^T \), the simple modification

\[
\tilde{Q} = \frac{1}{2}(Q + Q^T)
\]

produces an equivalent formulation \( x^TQx = x^T\tilde{Q}x \); hence symmetry is assumed. When you specify a quadratic matrix, it suffices to list only lower triangular coefficients.

In addition to being symmetric, \( Q \) is also required to be positive semidefinite for minimization type of models:

\[
x^TQx \geq 0, \quad \forall x \in \mathbb{R}^n
\]

\( Q \) is required to be negative semidefinite for maximization type of models. Convexity can come as a result of a matrix-matrix multiplication

\[
Q = LL^T
\]

or as a consequence of physical laws, and so on. See Figure 17.1 for examples of convex, concave, and nonconvex objective functions.

The order of constraints is insignificant. Some or all components of \( l \) or \( u \) (lower and upper bounds, respectively) can be omitted.

**Figure 17.1** Examples of Convex, Concave, and Nonconvex Objective Functions
Getting Started: QP Solver

Consider a small illustrative example. Suppose you want to minimize a two-variable quadratic function
\( f(x_1, x_2) \) on the nonnegative quadrant, subject to two constraints:

\[
\begin{align*}
\text{min} & \quad 2x_1 + 3x_2 + x_1^2 + 10x_2^2 + 2.5x_1x_2 \\
\text{subject to} & \quad x_1 - x_2 \leq 1 \\
& \quad x_1 + 2x_2 \geq 100 \\
& \quad x_1 \geq 0 \\
& \quad x_2 \geq 0
\end{align*}
\]

To use the OPTMODEL procedure, it is not necessary to fit this problem into the general QP formulation mentioned in the section “Overview: QP Solver” on page 825 and to compute the corresponding parameters. However, since these parameters are closely related to the data set that is used by the OPTQP procedure and has a quadratic programming system (QPS) format, you can compute these parameters as follows. The linear objective function coefficients, vector of right-hand sides, and lower and upper bounds are identified immediately as

\[
\begin{align*}
c &= \begin{bmatrix} 2 \\ 3 \end{bmatrix}, & b &= \begin{bmatrix} 1 \\ 100 \end{bmatrix}, & l &= \begin{bmatrix} 0 \\ 0 \end{bmatrix}, & u &= \begin{bmatrix} +\infty \\ +\infty \end{bmatrix}
\end{align*}
\]

Carefully construct the quadratic matrix \( Q \). Observe that you can use symmetry to separate the main-diagonal and off-diagonal elements:

\[
\frac{1}{2} x^T Q x = \frac{1}{2} \sum_{i,j=1}^{n} x_i q_{ij} x_j = \frac{1}{2} \sum_{i=1}^{n} q_{ii} x_i^2 + \sum_{i>j} x_i q_{ij} x_j
\]

The first expression

\[
\frac{1}{2} \sum_{i=1}^{n} q_{ii} x_i^2
\]

sums the main-diagonal elements. Thus, in this case you have

\[
q_{11} = 2, \quad q_{22} = 20
\]

Notice that the main-diagonal values are doubled in order to accommodate the \( 1/2 \) factor. Now the second term

\[
\sum_{i>j} x_i q_{ij} x_j
\]

sums the off-diagonal elements in the strict lower triangular part of the matrix. The only off-diagonal \((x_i x_j, i \neq j)\) term in the objective function is \(2.5x_1x_2\), so you have

\[
q_{21} = 2.5
\]

Notice that you do not need to specify the upper triangular part of the quadratic matrix.
Finally, the matrix of constraints is as follows:

\[
A = \begin{bmatrix}
1 & -1 \\
1 & 2
\end{bmatrix}
\]

The following OPTMODEL program formulates the preceding problem in a manner that is very close to the mathematical specification of the given problem:

```optmodel
/* getting started */
proc optmodel;
  var x1 >= 0; /* declare nonnegative variable x1 */
  var x2 >= 0; /* declare nonnegative variable x2 */

  /* objective: quadratic function f(x1, x2) */
  minimize f =
    /* the linear objective function coefficients */
    2 * x1 + 3 * x2 +
    /* quadratic <x, Qx> */
    x1 * x1 + 2.5 * x1 * x2 + 10 * x2 * x2;

  /* subject to the following constraints */
  con r1: x1 - x2 <= 1;
  con r2: x1 + 2 * x2 >= 100;

  /* specify iterative interior point algorithm (QP) */
  solve with qp;

  /* print the optimal solution */
  print x1 x2;
  save qps qpsdata;
quit;
```

The “with qp” clause in the SOLVE statement invokes the QP solver to solve the problem. The output is shown in Figure 17.2.
In this example, the SAVE QPS statement is used to save the QP problem in the QPS-format data set qpsdata, shown in Figure 17.3. The data set is consistent with the parameters of general quadratic programming previously computed. Also, the data set can be used as input to the OPTQP procedure.
Syntax: QP Solver

The following statement is available in the OPTMODEL procedure:

```plaintext
SOLVE WITH QP </options> ;
```

Functional Summary

Table 17.1 summarizes the list of options available for the SOLVE WITH QP statement, classified by function.

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solver Options</strong></td>
<td></td>
</tr>
<tr>
<td>Enables or disables IIS detection</td>
<td>IIS=</td>
</tr>
<tr>
<td><strong>Control Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the stopping criterion based on duality gap</td>
<td>DUALITYGAP=</td>
</tr>
<tr>
<td>Specifies the dual feasibility tolerance</td>
<td>DUALTOL=</td>
</tr>
<tr>
<td>Specifies how often to print the solution progress</td>
<td>LOGFREQ=</td>
</tr>
<tr>
<td>Specifies the maximum number of iterations</td>
<td>MAXITER=</td>
</tr>
<tr>
<td>Specifies the time limit for the optimization process</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Specifies the maximum number of threads</td>
<td>NTHREADS=</td>
</tr>
<tr>
<td>Specifies the type of presolve</td>
<td>PRESOLVER=</td>
</tr>
<tr>
<td>Specifies the primal feasibility tolerance</td>
<td>PRIMALTOL=</td>
</tr>
<tr>
<td>Specifies units of CPU time or real time</td>
<td>TIMETYPE=</td>
</tr>
</tbody>
</table>
QP Solver Options

This section describes the options recognized by the QP solver. These options can be specified after a forward slash (/) in the SOLVE statement, provided that the QP solver is explicitly specified using a WITH clause.

The QP solver does not provide an intermediate solution if the solver terminates before reaching optimality.

**DUALITYGAP=δ**

specifies the desired relative duality gap, $\delta \in [1E^{-9}, 1E^{-4}]$. This is the relative difference between the primal and dual objective function values and is the primary solution quality parameter. For more information, see the section “Interior Point Algorithm: Overview” on page 833. The default value is $1E^{-6}$.

**DUALTOL=β**

**OPTTOL=β**

specifies the maximum relative dual constraints violation, $\beta \in [1E^{-9}, 1E^{-4}]$. For more information, see the section “Interior Point Algorithm: Overview” on page 833. The default value is $1E^{-6}$.

**IIS=FALSE | TRUE**

specifies whether to attempt to identify a set of constraints and variables that form an irreducible infeasible set (IIS). You can specify the following values:

- **FALSE** — disables IIS detection.
- **TRUE** — enables IIS detection.

If an IIS is found, you can find information about the infeasibilities in the .status values of the constraints and variables. For more information about this option, see the section “Irreducible Infeasible Set” on page 836. For more information about the .status suffix, see the section “Suffixes” on page 380. By default, IIS=FALSE.

**LOGFREQ=k**

**PRINTFREQ=k**

prints the solution progress to the iteration log after every $k$ iterations, where $k$ is an integer between 0 and the largest four-byte signed integer, which is $2^{31} - 1$. The value $k = 0$ suppresses printing of the progress of the solution. By default, LOGFREQ=1.

**MAXITER=k**

specifies the maximum number of iterations, where $k$ can be any integer between 1 and the largest four-byte signed integer, which is $2^{31} - 1$. If you do not specify this option, the procedure does not stop based on the number of iterations performed.

**MAXTIME=t**

specifies an upper limit of $t$ units of time for the optimization process, including problem generation time and solution time. The value of the TIMETYPE= option determines the type of units used. If you do not specify the MAXTIME= option, the solver does not stop based on the amount of time elapsed. The value of $t$ can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.
**NTHREADS**\(=k\) specifies the number of threads that the QP solver can use, where \(k\) can be any integer between 1 and 256, inclusive. The default is the value of the OPTMODEL **NTHREADS** option.

Specifying \(k\) as a number greater than the actual number of available cores might result in reduced performance. Specifying a high value for \(k\) does not guarantee shorter solution time; the actual change in solution time depends on the computing hardware and the scalability of the underlying algorithms in the QP solver. In some circumstances, the QP solver might use fewer than \(k\) threads because the solver’s internal algorithms have determined that a smaller number is preferable.

**PRESOLVER**=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE specifies the presolve level. You can specify the following values:

- **AUTOMATIC** applies the presolver by using the default setting.
- **NONE** disables the presolver.
- **BASIC** applies the basic presolver.
- **MODERATE** applies the moderate presolver.
- **AGGRESSIVE** applies the aggressive presolver.

By default, **PRESOLVER**=AUTOMATIC.

**PRIMALTOL**=\(\alpha\)**

**FEASTOL**=\(\alpha\)** specifies the maximum relative bound and primal constraints violation, \(\alpha \in [1E^{-9}, 1E^{-4}]\). For more information, see the section “Interior Point Algorithm: Overview” on page 833. The default value is 1E–6.

**TIMETYPE**=CPU | REAL specifies the units of time used by the **MAXTIME** option and reported by the PRESOLVE_TIME and SOLUTION_TIME terms in the **_OROPTMODEL_** macro variable. You can specify the following values:

- **CPU** specifies that units are in CPU time.
- **REAL** specifies that units are in real time.

The “Optimization Statistics” table, an output of the OPTMODEL procedure if you specify PRINT-LEVEL=2 in the PROC OPTMODEL statement, also includes the same time units for Presolver Time, Solver Time, and other times (such as Problem Generation Time).

By default, **TIMETYPE**=REAL.
Details: QP Solver

Interior Point Algorithm: Overview

The QP solver implements an infeasible primal-dual predictor-corrector interior point algorithm. To illustrate the algorithm and the concepts of duality and dual infeasibility, consider the following QP formulation (the primal):

\[
\begin{align*}
\text{min} & \quad \frac{1}{2} x^T Q x + c^T x \\
\text{subject to} & \quad Ax \geq b \\
& \quad x \geq 0
\end{align*}
\]

The corresponding dual formulation is

\[
\begin{align*}
\text{max} & \quad -\frac{1}{2} x^T Q x + b^T y \\
\text{subject to} & \quad -Q x + A^T y + w = c \\
& \quad y \geq 0 \\
& \quad w \geq 0
\end{align*}
\]

where \( y \in \mathbb{R}^m \) refers to the vector of dual variables and \( w \in \mathbb{R}^n \) refers to the vector of dual slack variables.

The dual makes an important contribution to the certificate of optimality for the primal. The primal and dual constraints combined with complementarity conditions define the first-order optimality conditions, also known as KKT (Karush-Kuhn-Tucker) conditions, which can be stated as follows where \( e \equiv (1, \ldots, 1)^T \) of appropriate dimension and \( s \in \mathbb{R}^m \) is the vector of primal slack variables:

\[
\begin{align*}
Ax - s & = b \quad \text{(primal feasibility)} \\
-Q x + A^T y + w & = c \quad \text{(dual feasibility)} \\
WXe & = 0 \quad \text{(complementarity)} \\
SYe & = 0 \quad \text{(complementarity)} \\
x, y, w, s & \geq 0
\end{align*}
\]

**NOTE:** Slack variables (the \( s \) vector) are automatically introduced by the solver when necessary; it is therefore recommended that you not introduce any slack variables explicitly. This enables the solver to handle slack variables much more efficiently.

The letters \( X, Y, W, \) and \( S \) denote matrices with corresponding \( x, y, w, \) and \( s \) on the main diagonal and zero elsewhere, as in the following example:

\[
X = \begin{bmatrix}
x_1 & 0 & \cdots & 0 \\
0 & x_2 & \cdots & 0 \\
\vdots & \vdots & \ddots & \vdots \\
0 & 0 & \cdots & x_n
\end{bmatrix}
\]


If \((x^*, y^*, w^*, s^*)\) is a solution of the previously defined system of equations that represent the KKT conditions, then \(x^*\) is also an optimal solution to the original QP model.

At each iteration the interior point algorithm solves a large, sparse system of linear equations,

\[
\begin{bmatrix}
    Y^{-1}S & A \\
    A^T & -Q - X^{-1}W
\end{bmatrix}
\begin{bmatrix}
    \Delta y \\
    \Delta x
\end{bmatrix}
= \begin{bmatrix}
    \Xi \\
    \Theta
\end{bmatrix}
\]

where \(\Delta x\) and \(\Delta y\) denote the vector of search directions in the primal and dual spaces, respectively, and \(\Theta\) and \(\Xi\) constitute the vector of the right-hand sides.

The preceding system is known as the reduced KKT system. The QP solver uses a preconditioned quasi-minimum residual algorithm to solve this system of equations efficiently.

An important feature of the interior point algorithm is that it takes full advantage of the sparsity in the constraint and quadratic matrices, thereby enabling it to efficiently solve large-scale quadratic programs.

The interior point algorithm works simultaneously in the primal and dual spaces. It attains optimality when both primal and dual feasibility are achieved and when complementarity conditions hold. Therefore, it is of interest to observe the following four measures where \(\|v\|_2\) is the Euclidean norm of the vector \(v\):

- relative primal infeasibility measure \(\alpha\):
  \[
  \alpha = \frac{\|Ax - b - s\|_2}{\|b\|_2 + 1}
  \]

- relative dual infeasibility measure \(\beta\):
  \[
  \beta = \frac{\|Qx + c - A^Ty - w\|_2}{\|c\|_2 + 1}
  \]

- relative duality gap \(\delta\):
  \[
  \delta = \frac{|x^TQx + c^Tx - b^Ty|}{\frac{1}{2}x^TQx + c^Tx + 1}
  \]

- absolute complementarity \(\gamma\):
  \[
  \gamma = \sum_{i=1}^{n} x_i w_i + \sum_{i=1}^{m} y_i s_i
  \]

These measures are displayed in the iteration log.
Iteration Log

The following information is displayed in the iteration log:

<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iter</td>
<td>indicates the iteration number.</td>
</tr>
<tr>
<td>Complement</td>
<td>indicates the (absolute) complementarity.</td>
</tr>
<tr>
<td>Duality Gap</td>
<td>indicates the (relative) duality gap.</td>
</tr>
<tr>
<td>Primal Infeas</td>
<td>indicates the (relative) primal infeasibility measure.</td>
</tr>
<tr>
<td>Bound Infeas</td>
<td>indicates the (relative) bound infeasibility measure.</td>
</tr>
<tr>
<td>Dual Infeas</td>
<td>indicates the (relative) dual infeasibility measure.</td>
</tr>
<tr>
<td>Time</td>
<td>indicates the time elapsed (in seconds).</td>
</tr>
</tbody>
</table>

If the sequence of solutions converges to an optimal solution of the problem, you should see all columns in the iteration log converge to zero or very close to zero. Nonconvergence can be the result of insufficient iterations being performed to reach optimality. In this case, you might need to increase the value that you specify in the MAXITER= or MAXTIME= option. If the complementarity or the duality gap does not converge, the problem might be infeasible or unbounded. If the infeasibility columns do not converge, the problem might be infeasible.

Problem Statistics

Optimizers can encounter difficulty when solving poorly formulated models. Information about data magnitude provides a simple gauge to determine how well a model is formulated. For example, a model whose constraint matrix contains one very large entry (on the order of $10^9$) can cause difficulty when the remaining entries are single-digit numbers. The PRINTLEVEL=2 option in the OPTMODEL procedure causes the ODS table ProblemStatistics to be generated when the QP solver is called. This table provides basic data magnitude information that enables you to improve the formulation of your models.

The example output in Figure 17.4 demonstrates the contents of the ODS table ProblemStatistics.
Irreducible Infeasible Set

For a quadratic programming problem, an irreducible infeasible set (IIS) is an infeasible subset of constraints and variable bounds that becomes feasible if any single constraint or variable bound is removed. It is possible to have more than one IIS in an infeasible QP. Identifying an IIS can help isolate the structural infeasibility in a QP. The IIS=TRUE option directs the QP solver to search for an IIS in a specified QP.

Whether a quadratic programming problem is feasible or infeasible is determined by its constraints and variable bounds, which have nothing to do with its objective function. When you specify the IIS=TRUE option, the QP solver treats this problem as a linear programming problem by ignoring its objective function. Then finding IIS is the same as what the LP solver does with the IIS=TRUE option. For more information about the irreducible infeasible set, see the section “Irreducible Infeasible Set” on page 519 in Chapter 12, “The Linear Programming Solver.”
Macro Variable _OROPTMODEL_

The OPTMODEL procedure always creates and initializes a SAS macro called _OROPTMODEL_. This variable contains a character string. After each PROC OROPTMODEL run, you can examine this macro by specifying `%put &OROPTMODEL_;` and check the execution of the most recently invoked solver from the value of the macro variable. The various terms of the variable after the QP solver is called are interpreted as follows.

**STATUS**

indicates the solver status at termination. It can take one of the following values:

- **OK** The solver terminated normally.
- **SYNTAX_ERROR** Incorrect syntax was used.
- **DATA_ERROR** The input data were inconsistent.
- **OUT_OF_MEMORY** Insufficient memory was allocated to the procedure.
- **IO_ERROR** A problem occurred in reading or writing data.
- **SEMANTIC_ERROR** An evaluation error, such as an invalid operand type, occurred.
- **ERROR** The status cannot be classified into any of the preceding categories.

**ALGORITHM**

indicates the algorithm that produced the solution data in the macro variable. This term appears only when STATUS=OK. It can take the following value:

- **IP** The interior point algorithm produced the solution data.
- **IIS** The IIS functionality produced the solution data.

**SOLUTION_STATUS**

indicates the solution status at termination. It can take one of the following values:

- **OPTIMAL** The solution is optimal.
- **CONDITIONAL_OPTIMAL** The solution is optimal, but some infeasibilities (primal, dual or bound) exceed tolerances due to scaling or pre-processing.
- **INFEASIBLE** The problem is infeasible.
- **UNBOUNDED** The problem is unbounded.
- **INFEASIBLE_OR_UNBOUNDED** The problem is infeasible or unbounded.
- **BAD_PROBLEM_TYPE** The problem type is unsupported by the solver.
- **ITERATION_LIMIT_REACHED** The maximum allowable number of iterations was reached.
- **TIME_LIMIT_REACHED** The solver reached its execution time limit.
- **FUNCTION_CALL_LIMIT_REACHED** The solver reached its limit on function evaluations.
- **INTERRUPTED** The solver was interrupted externally.
- **FAILED** The solver failed to converge, possibly due to numerical issues.
**OBJECTIVE**
indicates the objective value obtained by the solver at termination.

**PRIMAL INFEASIBILITY**
indicates the (relative) infeasibility of the primal constraints at the solution. For more information, see the section “Interior Point Algorithm: Overview” on page 833.

**DUAL INFEASIBILITY**
indicates the (relative) infeasibility of the dual constraints at the solution. For more information, see the section “Interior Point Algorithm: Overview” on page 833.

**BOUND INFEASIBILITY**
indicates the (relative) violation by the solution of the lower or upper bounds (or both). For more information, see the section “Interior Point Algorithm: Overview” on page 833.

**DUALITY_GAP**
indicates the (relative) duality gap. For more information, see the section “Interior Point Algorithm: Overview” on page 833.

**COMPLEMENTARITY**
indicates the (absolute) complementarity at the solution. For more information, see the section “Interior Point Algorithm: Overview” on page 833.

**ITERATIONS**
indicates the number of iterations required to solve the problem.

**PRESOLVE_TIME**
indicates the time (in seconds) taken for preprocessing.

**SOLUTION_TIME**
indicates the time (in seconds) taken to solve the problem, including preprocessing time.

**Note:** The time that is reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time or real time. The type is determined by the TIMETYPE= option.

---

### Examples: QP Solver

This section presents examples that illustrate the use of the OPTMODEL procedure to solve quadratic programming problems. Example 17.1 illustrates how to model a linear least squares problem and solve it by using PROC OPTMODEL. Example 17.2 and Example 17.3 show in detail how to model the portfolio optimization and selection problems.
Example 17.1: Linear Least Squares Problem

The linear least squares problem arises in the context of determining a solution to an overdetermined set of linear equations. In practice, these equations could arise in data fitting and estimation problems. An overdetermined system of linear equations can be defined as

\[ Ax = b \]

where \( A \in \mathbb{R}^{m \times n} \), \( x \in \mathbb{R}^n \), \( b \in \mathbb{R}^m \), and \( m > n \). Since this system usually does not have a solution, you need to be satisfied with some sort of approximate solution. The most widely used approximation is the least squares solution, which minimizes \( \|Ax - b\|_2^2 \).

This problem is called a least squares problem for the following reason. Let \( Ax = b \) be defined as previously. Let \( k_i(x) \) be the \( k \)th component of the vector \( Ax - b \):

\[ k_i(x) = a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{in}x_n - b_i, \quad i = 1, 2, \ldots, m \]

By definition of the Euclidean norm, the objective function can be expressed as follows:

\[ \|Ax - b\|_2^2 = \sum_{i=1}^{m} k_i(x)^2 \]

Therefore, the function you minimize is the sum of squares of \( m \) terms \( k_i(x) \); hence the term least squares. The following example is an illustration of the linear least squares problem; that is, each of the terms \( k_i \) is a linear function of \( x \).

Consider the following least squares problem defined by

\[
A = \begin{bmatrix}
4 & 0 \\
-1 & 1 \\
3 & 2
\end{bmatrix}, \quad b = \begin{bmatrix}
1 \\
0 \\
1
\end{bmatrix}
\]

This translates to the following set of linear equations:

\[ 4x_1 = 1, \quad -x_1 + x_2 = 0, \quad 3x_1 + 2x_2 = 1 \]

The corresponding least squares problem is:

\[
\text{minimize} \quad (4x_1 - 1)^2 + (-x_1 + x_2)^2 + (3x_1 + 2x_2 - 1)^2
\]

The preceding objective function can be expanded to:

\[
\text{minimize} \quad 26x_1^2 + 5x_2^2 + 10x_1x_2 - 14x_1 - 4x_2 + 2
\]

In addition, you impose the following constraint so that the equation \( 3x_1 + 2x_2 = 1 \) is satisfied within a tolerance of 0.1:

\[ 0.9 \leq 3x_1 + 2x_2 \leq 1.1 \]
You can use the following SAS statements to solve the least squares problem:

```sas
/* example 1: linear least squares problem */
proc optmodel;
    /* declare free (no explicit bounds) variables x[1] and x[2] */
    var x {1..2};

    /* objective function: minimize the sum of squares */

    /* subject to the following constraint */
    con R: 0.9 <= 3*x[1] + 2*x[2] <= 1.1;

    /* call the QP solver */
    solve;

    /* print the optimal solution */
    print x;
quit;
```

The output is shown in Output 17.1.1.

**Output 17.1.1** Summaries and Optimal Solution

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
<tr>
<td>Number of Variables</td>
</tr>
<tr>
<td>Bounded Above</td>
</tr>
<tr>
<td>Bounded Below</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
</tr>
<tr>
<td>Free</td>
</tr>
<tr>
<td>Fixed</td>
</tr>
<tr>
<td>Number of Constraints</td>
</tr>
<tr>
<td>Linear LE (&lt;=)</td>
</tr>
<tr>
<td>Linear EQ (=)</td>
</tr>
<tr>
<td>Linear GE (&gt;=)</td>
</tr>
<tr>
<td>Linear Range</td>
</tr>
<tr>
<td>Constraint Coefficients</td>
</tr>
<tr>
<td>Hessian Diagonal Elements</td>
</tr>
<tr>
<td>Hessian Elements Below Diagonal</td>
</tr>
</tbody>
</table>
Example 17.2: Portfolio Optimization

Consider a portfolio optimization example. The two competing goals of investment are (1) long-term growth of capital and (2) low risk. A good portfolio grows steadily without wild fluctuations in value. The Markowitz model is an optimization model for balancing the return and risk of a portfolio. The decision variables are the amounts invested in each asset. The objective is to minimize the variance of the portfolio’s total return, subject to the constraints that (1) the expected growth of the portfolio reaches at least some target level and (2) you do not invest more capital than you have.

Let $x_1, \ldots, x_n$ be the amount invested in each asset, $B$ be the amount of capital you have, $R$ be the random vector of asset returns over some period, and $\mathbf{r}$ be the expected value of $R$. Let $G$ be the minimum growth you hope to obtain, and $\mathbf{C}$ be the covariance matrix of $R$. The objective function is $\text{Var}\left(\sum_{i=1}^{n} x_i R_i\right)$, which can be equivalently denoted as $\mathbf{x}^T \mathbf{C} \mathbf{x}$.

Assume, for example, $n = 4$. Let $B = 10,000$, $G = 1,000$, $\mathbf{r} = [0.05, -0.2, 0.15, 0.30]$, and

$$
\mathbf{C} = \begin{bmatrix}
0.08 & -0.05 & -0.05 & -0.05 \\
-0.05 & 0.16 & -0.02 & -0.02 \\
-0.05 & -0.02 & 0.35 & 0.06 \\
-0.05 & -0.02 & 0.06 & 0.35 
\end{bmatrix}
$$
The QP formulation can be written as:

\[
\begin{align*}
\text{min} & \quad 0.08x_1^2 - 0.1x_1x_2 - 0.1x_1x_3 - 0.1x_1x_4 + 0.16x_2^2 \\
& \quad -0.04x_2x_3 - 0.04x_2x_4 + 0.35x_3^2 + 0.12x_3x_4 + 0.35x_4^2 \\
\text{subject to} & \\
\text{(budget)} & \quad x_1 + x_2 + x_3 + x_4 \leq 10000 \\
\text{(growth)} & \quad 0.05x_1 - 0.2x_2 + 0.15x_3 + 0.30x_4 \geq 1000 \\
& \quad x_1, x_2, x_3, x_4 \geq 0
\end{align*}
\]

Use the following SAS statements to solve the problem:

```sas
/* example 2: portfolio optimization */
proc optmodel;
    /* let x1, x2, x3, x4 be the amount invested in each asset */
    var x{1..4} >= 0;
    num coeff{1..4, 1..4} = 
        [0.08 -.05 -.05 -.05 \
        -.05 0.16 -.02 -.02 \
        -.05 -.02 0.35 0.06 \
        -.05 -.02 0.06 0.35];
    num r{1..4} = [0.05 -.20 0.15 0.30];
    /* minimize the variance of the portfolio's total return */
    minimize f = sum{i in 1..4, j in 1..4} coeff[i,j]*x[i]*x[j];
    /* subject to the following constraints */
    con BUDGET: sum{i in 1..4} x[i] <= 10000;
    con GROWTH: sum{i in 1..4} r[i]*x[i] >= 1000;
    solve with qp;
    /* print the optimal solution */
    print x;
```

The summaries and the optimal solution are shown in Output 17.2.1.
### Output 17.2.1 Portfolio Optimization

#### The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
<tr>
<td>Number of Variables</td>
</tr>
<tr>
<td>Bounded Above</td>
</tr>
<tr>
<td>Bounded Below</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
</tr>
<tr>
<td>Free</td>
</tr>
<tr>
<td>Fixed</td>
</tr>
<tr>
<td>Number of Constraints</td>
</tr>
<tr>
<td>Linear LE (&lt;=)</td>
</tr>
<tr>
<td>Linear EQ (=)</td>
</tr>
<tr>
<td>Linear GE (&gt;=)</td>
</tr>
<tr>
<td>Linear Range</td>
</tr>
<tr>
<td>Constraint Coefficients</td>
</tr>
<tr>
<td>Hessian Diagonal Elements</td>
</tr>
<tr>
<td>Hessian Elements Below Diagonal</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
</tr>
<tr>
<td>Dual Infeasibility</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
</tr>
<tr>
<td>Duality Gap</td>
</tr>
<tr>
<td>Complementarity</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[1]x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 3452.9</td>
</tr>
<tr>
<td>2 0.0</td>
</tr>
<tr>
<td>3 1068.8</td>
</tr>
<tr>
<td>4 2223.5</td>
</tr>
</tbody>
</table>
Thus, the minimum variance portfolio that earns an expected return of at least 10% is \(x_1 = 3.452.9, x_2 = 0,\)
\(x_3 = 1.068.8, x_4 = 2.223.5\). Asset 2 gets nothing because its expected return is \(-20\%\) and its covariance with the other assets is not sufficiently negative for it to bring any diversification benefits. What if you drop
the nonnegativity assumption?

Financially, that means you are allowed to short-sell—that is, sell low-mean-return assets and use the proceeds to invest in high-mean-return assets. In other words, you put a negative portfolio weight in low-mean assets and “more than 100%” in high-mean assets.

To solve the portfolio optimization problem with the short-sale option, continue to submit the following SAS statements:

```sas
/* example 2: portfolio optimization with short-sale option */
/* dropping nonnegativity assumption */
for {i in 1..4} x[i].lb=-x[i].ub;
solve with qp;

/* print the optimal solution */
print x;
quit;
```

You can see in the optimal solution displayed in Output 17.2.2 that the decision variable \(x_2\), denoting Asset 2, is equal to \(-1,563.61\), which means short sale of that asset.

### Output 17.2.2 Portfolio Optimization with Short-Sale Option

<table>
<thead>
<tr>
<th>The OPTMODEL Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution Summary</td>
</tr>
<tr>
<td>Solver: QP</td>
</tr>
<tr>
<td>Algorithm: Interior Point</td>
</tr>
<tr>
<td>Objective Function: f</td>
</tr>
<tr>
<td>Solution Status: Optimal</td>
</tr>
<tr>
<td>Objective Value: 1907122.2254</td>
</tr>
</tbody>
</table>

| Primal Infeasibility: 4.988212E-14 |
| Dual Infeasibility: 4.4944064E-8 |
| Bound Infeasibility: 0 |
| Duality Gap: 3.886226E-11 |
| Complementarity: 0 |

| Iterations: 5 |
| Presolve Time: 0.00 |
| Solution Time: 0.01 |

<table>
<thead>
<tr>
<th>[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
Example 17.3: Portfolio Selection with Transactions

Consider a portfolio selection problem with a slight modification. You are now required to take into account the current position and transaction costs associated with buying and selling assets. The objective is to find the minimum variance portfolio. In order to understand the scenario better, consider the following data.

You are given three assets. The current holding of the three assets is denoted by the vector \( c = [200, 300, 500] \), the amount of asset bought and sold is denoted by \( b_i \) and \( s_i \), respectively, and the net investment in each asset is denoted by \( x_i \) and is defined by the following relation:

\[
x_i - b_i + s_i = c_i, \quad i = 1, 2, 3
\]

Suppose that you pay a transaction fee of 0.01 every time you buy or sell. Let the covariance matrix \( C \) be defined as

\[
C = \begin{bmatrix}
0.027489 & -0.00874 & -0.00015 \\
-0.00874 & 0.109449 & -0.00012 \\
-0.00015 & -0.00012 & 0.000766
\end{bmatrix}
\]

Assume that you hope to obtain at least 12% growth. Let \( r = [1.109048, 1.169048, 1.074286] \) be the vector of expected return on the three assets, and let \( B=1000 \) be the available funds. Mathematically, this problem can be written in the following manner:

\[
\begin{align*}
\min & \quad 0.027489x_1^2 - 0.01748x_1x_2 - 0.0003x_1x_3 + 0.109449x_2^2 \\
& -0.00024x_2x_3 + 0.000766x_3^2 \\
\text{subject to} & \\
\text{(return)} & \quad \sum_{i=1}^{3} r_i x_i \geq 1.12B \\
\text{(budget)} & \quad \sum_{i=1}^{3} x_i + \sum_{i=1}^{3} 0.01(b_i + s_i) = B \\
\text{(balance)} & \quad x_i - b_i + s_i = c_i, \quad i = 1, 2, 3 \\
& \quad x_i, b_i, s_i \geq 0, \quad i = 1, 2, 3
\end{align*}
\]
The problem can be solved by the following SAS statements:

```sas
/* example 3: portfolio selection with transactions */
proc optmodel;
/* let x1, x2, x3 be the amount invested in each asset */
var x{1..3} >= 0;
/* let b1, b2, b3 be the amount of asset bought */
var b{1..3} >= 0;
/* let s1, s2, s3 be the amount of asset sold */
var s{1..3} >= 0;

/* current holdings */
num c{1..3}=[ 200 300 500];

/* covariance matrix */
num coeff{1..3, 1..3} = 
  [0.027489 -.008740 -.000150
   -.008740 0.109449 -.000120
   -.000150 -.000120 0.000766];

/* returns */
num r{1..3}=[1.109048 1.169048 1.074286];

/* minimize the variance of the portfolio's total return */
minimize f = sum{i in 1..3, j in 1..3}coeff[i,j]*x[i]*x[j];

/* subject to the following constraints */
con BUDGET: sum{i in 1..3}(x[i]+.01*b[i]+.01*s[i]) <= 1000;
con RETURN: sum{i in 1..3}r[i]*x[i] >= 1120;
con BALANC{i in 1..3}: x[i]-b[i]+s[i]=c[i];

solve with qp;

/* print the optimal solution */
print x b s;
quit;
```

The output is displayed in Output 17.3.1.
Output 17.3.1 Portfolio Selection with Transactions

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
<tr>
<td>Number of Variables</td>
</tr>
<tr>
<td>Bounded Above</td>
</tr>
<tr>
<td>Bounded Below</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
</tr>
<tr>
<td>Free</td>
</tr>
<tr>
<td>Fixed</td>
</tr>
<tr>
<td>Number of Constraints</td>
</tr>
<tr>
<td>Linear LE ($\leq$)</td>
</tr>
<tr>
<td>Linear EQ ($=$)</td>
</tr>
<tr>
<td>Linear GE ($\geq$)</td>
</tr>
<tr>
<td>Linear Range</td>
</tr>
<tr>
<td>Constraint Coefficients</td>
</tr>
<tr>
<td>Hessian Diagonal Elements</td>
</tr>
<tr>
<td>Hessian Elements Below Diagonal</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
</tr>
<tr>
<td>Dual Infeasibility</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
</tr>
<tr>
<td>Duality Gap</td>
</tr>
<tr>
<td>Complementarity</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$[1]$</th>
<th>$x$</th>
<th>$b$</th>
<th>$s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>397.58</td>
<td>197.58</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>406.12</td>
<td>106.12</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td>190.17</td>
<td>0.00</td>
<td>309.83</td>
</tr>
</tbody>
</table>
References


Chapter 18
The Decomposition Algorithm

Contents

Overview: Decomposition Algorithm ................................................. 850
Getting Started: Decomposition Algorithm ....................................... 852
  Solving a MILP with DECOMP and PROC OPTMODEL ...................... 852
  Solving a MILP with DECOMP and PROC OPTMILP ......................... 854
Syntax: Decomposition Algorithm .................................................. 855
  Decomposition Algorithm Options in the PROC OPTLP Statement or the SOLVE WITH LP Statement in PROC OPTMODEL ..................... 856
  Decomposition Algorithm Options in the PROC OPTMILP Statement or the SOLVE WITH MILP Statement in PROC OPTMODEL ............. 857
  DECOMP Statement ..................................................................... 860
  DECOMPMASTER Statement ....................................................... 863
  DECOMPMASTERIP Statement .................................................... 865
  DECOMPSUBPROB Statement ..................................................... 867
Details: Decomposition Algorithm ................................................... 871
  Data Input .............................................................................. 871
  Decomposition Algorithm ......................................................... 872
  Parallel Processing ..................................................................... 873
  Special Case: Identical Blocks and Ryan-Foster Branching .............. 873
  Log for the Decomposition Algorithm .......................................... 877
Examples: Decomposition Algorithm ................................................. 879
  Example 18.1: Multicommodity Flow Problem and METHOD=NETWORK . 879
  Example 18.2: Generalized Assignment Problem .............................. 885
  Example 18.3: Block-Diagonal Structure and METHOD=CONCOMP .......... 892
  Example 18.4: Block-Angular Structure and METHOD=AUTO in Single-Machine Mode .............................................................. 896
  Example 18.5: Block-Angular Structure and METHOD=AUTO in Distributed Mode ................................................................. 900
  Example 18.6: Bin Packing Problem .............................................. 902
  Example 18.7: Resource Allocation Problem .................................... 907
  Example 18.8: Vehicle Routing Problem ......................................... 920
  Example 18.9: ATM Cash Management in Single-Machine Mode ........ 926
  Example 18.10: ATM Cash Management in Distributed Mode ............ 937
  Example 18.11: Kidney Donor Exchange and METHOD=SET ................ 939
References ................................................................................. 946
Overview: Decomposition Algorithm

The SAS/OR decomposition algorithm (DECOMP) provides an alternative method of solving linear programs (LPs) and mixed integer linear programs (MILPs) by exploiting the ability to efficiently solve a relaxation of the original problem. The algorithm is available as an option in the OPTMODEL, OPTLP, and OPTMILP procedures and is based on the methodology described in Galati (2009).

A standard linear or mixed integer linear program has the formulation

\[
\begin{align*}
\text{minimize} & \quad c^T x + f^T y \\
\text{subject to} & \quad D x + B y \{\geq, =, \leq\} d \quad \text{(master)} \\
& \quad A x \{\geq, =, \leq\} b \quad \text{(subproblem)} \\
& \quad \underline{x} \leq x \leq \bar{x} \\
& \quad \underline{y} \leq y \leq \bar{y} \\
& \quad x_i \in \mathbb{Z} \quad i \in S_x \\
& \quad y_i \in \mathbb{Z} \quad i \in S_y
\end{align*}
\]

where

- \( x \in \mathbb{R}^n \) is the vector of structural variables
- \( y \in \mathbb{R}^s \) is the vector of master-only structural variables
- \( c \in \mathbb{R}^n \) is the vector of objective function coefficients that are associated with variables \( x \)
- \( f \in \mathbb{R}^s \) is the vector of objective function coefficients that are associated with variables \( y \)
- \( D \in \mathbb{R}^{t \times n} \) is the matrix of master constraint coefficients that are associated with variables \( x \)
- \( B \in \mathbb{R}^{t \times s} \) is the matrix of master constraint coefficients that are associated with variables \( y \)
- \( A \in \mathbb{R}^{m \times n} \) is the matrix of subproblem constraint coefficients
- \( d \in \mathbb{R}^t \) is the vector of master constraints’ right-hand sides
- \( b \in \mathbb{R}^m \) is the vector of subproblem constraints’ right-hand sides
- \( \underline{x} \in \mathbb{R}^n \) is the vector of lower bounds on variables \( x \)
- \( \bar{x} \in \mathbb{R}^n \) is the vector of upper bounds on variables \( x \)
- \( \underline{y} \in \mathbb{R}^s \) is the vector of lower bounds on variables \( y \)
- \( \bar{y} \in \mathbb{R}^s \) is the vector of upper bounds on variables \( y \)
- \( S_x \) is a subset of the set \( \{1, \ldots, n\} \) of indices on variables \( x \)
- \( S_y \) is a subset of the set \( \{1, \ldots, s\} \) of indices on variables \( y \)

You can form a relaxation of the preceding mathematical program by removing the master constraints, which are defined by the matrices \( D \) and \( B \). The resulting constraint system, defined by the matrix \( A \), forms the subproblem, which can often be solved much more efficiently than the entire original problem. This is one of the key motivators for using the decomposition algorithm.

The decomposition algorithm works by finding convex combinations of extreme points of the subproblem polyhedron that satisfy the constraints defined in the master. For MILP subproblems, the strength of the relaxation is another important motivator for using this method. If the subproblem polyhedron defines feasible solutions that are close to the original feasible space, the chance of success for the algorithm increases.
The region that defines the subproblem space is often separable. That is, the formulation of the preceding mathematical program can be written in block-angular form as

\[
\begin{align*}
\text{minimize} & \quad c^1 x^1 + c^2 x^2 + \cdots + c^\kappa x^\kappa + f^\top y \\
\text{subject to} & \quad D^1 x^1 + D^2 x^2 + \cdots + D^\kappa x^\kappa + B y \quad \{\geq, =, \leq\} \quad d \\
& \quad \{\geq, =, \leq\} \quad b^1 \\
& \quad \{\geq, =, \leq\} \quad b^2 \\
& \quad \{\geq, =, \leq\} \quad \vdots \\
& \quad \{\geq, =, \leq\} \quad b^\kappa \\
& \quad \mathbf{x} \leq \mathbf{x} \leq \mathbf{\bar{x}} \\
& \quad \mathbf{y} \leq \mathbf{y} \leq \mathbf{\bar{y}} \\
& \quad x_i \in \mathbb{Z} \\
& \quad y_i \in \mathbb{Z} \\
& \quad i \in S_x \\
& \quad i \in S_y
\end{align*}
\]

where \( K = \{1, \ldots, \kappa\} \) defines a partition of the constraints (and variables) into independent subproblems (blocks) such that \( A = [A^1 \ldots A^\kappa], D = [D^1 \ldots D^\kappa], c = [c^1 \ldots c^\kappa], b = [b^1 \ldots b^\kappa], \mathbf{x} = [\mathbf{x}^1 \ldots \mathbf{x}^\kappa], \mathbf{\bar{x}} = [\mathbf{\bar{x}}^1 \ldots \mathbf{\bar{x}}^\kappa], \) and \( \mathbf{\bar{x}} = [\mathbf{x}^1 \ldots x^\kappa] \). This type of structure is relatively common in modeling mathematical programs. For example, consider a model that defines a workplace that has separate departmental restrictions (defined as the subproblem constraints), which are coupled together by a company-wide budget across departments (defined as the master constraint). By relaxing the budget (master) constraint, the decomposition algorithm can take advantage of the fact that the decoupled subproblems are separable, and it can process them in parallel. A special case of block-angular form, called block-diagonal form, occurs when the set of master constraints is empty. In this special case, the subproblem matrices define the entire original problem.

An important indicator of a problem that is well suited for decomposition is the amount by which the subproblems cover the original problem with respect to both variables and constraints in the original presolved model. This value, which is expressed as a percentage of the original model, is known as the coverage. For LPs, the decomposition algorithm usually performs better than standard approaches only if the subproblems cover a significant amount of the original problem. For MILPs, the correlation between performance and coverage is more difficult to determine, because the strength of the subproblem with respect to integrality is not always proportional to the size of the system. Regardless, it is unlikely that the decomposition algorithm will outperform more standard methods (such as branch-and-cut) in problems that have small coverage.

The primary input and output for the decomposition algorithm are identical to those that are needed and produced by the OPTLP, OPTMILP, and OPTMODEL procedures. For more information, see the following sections:

- “Data Input and Output” on page 119 in Chapter 6, “The OPTLP Procedure”
- “Data Input and Output” on page 177 in Chapter 7, “The OPTMILP Procedure”
- “Details: LP Solver” on page 510 in Chapter 12, “The Linear Programming Solver”
- “Details: MILP Solver” on page 621 in Chapter 14, “The Mixed Integer Linear Programming Solver”

The only additional input that can be provided for the decomposition algorithm is an explicit definition of the partition of the subproblem constraints. The following section gives a simple example of providing this input for both PROC OPTMILP and PROC OPTMODEL.


### Getting Started: Decomposition Algorithm

This example illustrates how you can use the decomposition algorithm to solve a simple mixed integer linear program. Suppose you want to solve the following problem:

\[
\begin{align*}
\text{maximize} & \quad x_{11} + 2x_{21} + x_{31} + x_{12} + x_{22} + x_{32} \\
\text{subject to} & \quad 5x_{11} + 7x_{21} + 4x_{31} \geq 1 \quad (m) \\
& \quad 5x_{11} + 7x_{21} + 4x_{31} \leq 11 \quad (s1) \\
& \quad x_{12} + 2x_{22} + x_{32} \leq 2 \quad (s2) \\
& \quad x_{ij} \in \{0, 1\} \quad i \in \{1, 2, 3\}, j \in \{1, 2\}
\end{align*}
\]

It is obvious from the structure of the problem that if constraint \(m\) is removed, then the remaining constraints \(s1\) and \(s2\) decompose into two independent subproblems. The next two sections describe how to solve this MILP by using the decomposition algorithm in the OPTMODEL procedure and OPTMILP procedure, respectively.

### Solving a MILP with DECOMP and PROC OPTMODEL

The following statements use the OPTMODEL procedure and the decomposition algorithm to solve the MILP:

```plaintext
proc optmodel sessref="mysess";
    var x{i in 1..3, j in 1..2} binary;
    max f = x[1,1] + 2*x[2,1] + x[3,1] \\
            + x[2,2] + x[3,2];
    con m : x[1,1] + x[1,2] >= 1;
    con s1: 5*x[1,1] + 7*x[2,1] + 4*x[3,1] <= 11;
    con s2: x[1,2] + 2*x[2,2] + x[3,2] <= 2;
    s1.block = 0;
    s2.block = 1;
    solve with milp / presolver=none decomp=(logfreq=1);
    print x;
quit;
```

Here, the PRESOLVER=NONE option is used, because otherwise the presolver solves this small instance without invoking any solver. The solution summary and optimal solution are displayed in Figure 18.1.
The iteration log, which displays the problem statistics, the progress of the solution, and the optimal objective value, is shown in Figure 18.2.
Solving a MILP with DECOMP and PROC OPTMILP

Alternatively, to solve the MILP with the OPTMILP procedure, create a corresponding data table that uses the mathematical programming system (MPS) format as follows:

```plaintext
data mpsdata;
  input field1 $ field2 $ field3 $ field4 field5 $ field6;
datalines;
NAME .    mpsdata . . .
ROWS . . . . . .
MAX f . . . . . .
G m . . . . . .
L s1 . . . . . .
L s2 . . . . . .
```

Figure 18.2 Log

NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6 variables (0 free, 0 fixed).
NOTE: The problem has 6 binary and 0 integer variables.
NOTE: The problem has 3 linear constraints (2 LE, 0 EQ, 1 GE, 0 range).
NOTE: The problem has 8 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The SOLVE statement is executing in the distributed computing environment in single-machine mode.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The number of block threads has been reduced to 2 threads.
NOTE: The problem has a decomposable structure with 2 blocks. The largest block covers 33.33% of the constraints in the problem.
NOTE: The decomposition subproblems cover 6 (100%) variables and 2 (66.67%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Iter</th>
<th>Best Bound</th>
<th>Master Objective</th>
<th>Best Integer</th>
<th>Gap</th>
<th>Gap Time</th>
<th>Time</th>
<th>CPU Real Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4.0000</td>
<td>.</td>
<td>4.0000</td>
<td>.</td>
<td>0.00%</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Node Active Sols Best Integer Bound Gap CPU Real Time Time
0 0 3 4.0000 4.0000 0.00% 0 0

NOTE: The Decomposition algorithm used 32 threads.
NOTE: The Decomposition algorithm time is 0.10 seconds.
NOTE: Optimal.
NOTE: Objective = 4.
Next, use the following data table to define the subproblem blocks:

```
data mycas.blocks;
  input _row_ $ _block_;
datalines;
s1 0
s2 1
;
```

Now, you can use the following PROC OPTMILP statements to solve this MILP:

```
proc optmip
  data       = mycas.mpsdata
  presolver  = none;
  decomp
    logfreq = 1
    blocks  = mycas.blocks;
run;
```

**Syntax: Decomposition Algorithm**

You can specify the decomposition algorithm either by using options in a SOLVE statement in the OPTMODEL procedure or by using statements in the OPTLP and OPTMILP procedures. Except for the fact
that you use SOLVE statement options in PROC OPTMODEL or you use statements in PROC OPTLP and PROC OPTMILP, the syntax is identical.

You can specify the following decomposition algorithm options in the SOLVE statement in the OPTMODEL procedure:

```plaintext
SOLVE WITH LP / <options>
   <DECOMP=(decomp-options)>
   <DECOMPMASTER=(<master-options>)>
   <DECOMPSUBPROB=(<subprob-options>)> ;

SOLVE WITH MILP / <options>
   <DECOMP=(decomp-options)>
   <DECOMPMASTER=(<master-options>)>
   <DECOMPMASTERIP=(<masterip-options>)>
   <DECOMPSUBPROB=(<subprob-options>)> ;
```

You can specify the following statements in the OPTLP procedure:

```plaintext
PROC OPTLP <options> ;
   DECOMP <decomp-options> ;
   DECOMPMASTER <master-options> ;
   DECOMPSUBPROB <subprob-options> ;
```

You can specify the following statements in the OPTMILP procedure:

```plaintext
PROC OPTMILP <options> ;
   DECOMP <decomp-options> ;
   DECOMPMASTER <master-options> ;
   DECOMPMASTERIP <masterip-options> ;
   DECOMPSUBPROB <subprob-options> ;
```

Decomposition Algorithm Options in the PROC OPTLP Statement or the SOLVE WITH LP Statement in PROC OPTMODEL

To solve a linear program, you can specify the decomposition algorithm in a SOLVE WITH LP statement in the OPTMODEL procedure or in a PROC OPTLP statement in the OPTLP procedure. To control the overall decomposition algorithm, you can specify one or more of the LP solver options shown in Table 18.1. (As the table indicates, you can specify some options only in the PROC OPTLP statement.)

The options in Table 18.1 control the overall process flow for solving a linear program; they are equivalent to the options that are used in PROC OPTLP and PROC OPTMODEL with standard methods. These options are called main solver options in this chapter. They are described in detail in the section “Syntax: LP Solver” on page 504 in Chapter 12, “The Linear Programming Solver,” and the section “Syntax: OPTLP Procedure” on page 111 in Chapter 6, “The OPTLP Procedure.” The DUALIZE= option has a different default when you use the decomposition algorithm, as shown in Table 18.1.
Table 18.1 LP Options in the PROC OPTLP Statement or SOLVE WITH LP Statement

<table>
<thead>
<tr>
<th>Description</th>
<th>option</th>
<th>Different Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Table Options (OPTLP procedure only)</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the input data table</td>
<td>DATA=</td>
<td></td>
</tr>
<tr>
<td>Specifies the dual solution output data table</td>
<td>DUALOUT=</td>
<td></td>
</tr>
<tr>
<td>Specifies whether the model is a maximization or minimization problem</td>
<td>OBJSENSE=</td>
<td></td>
</tr>
<tr>
<td>Specifies the primal solution output data table</td>
<td>PRIMALOUT=</td>
<td></td>
</tr>
<tr>
<td><strong>Presolve Options</strong></td>
<td>DUALIZE=</td>
<td>OFF</td>
</tr>
<tr>
<td>Specifies the type of presolve</td>
<td>PRESOLVER=</td>
<td></td>
</tr>
<tr>
<td><strong>Control Options</strong></td>
<td>FEASTOL=</td>
<td></td>
</tr>
<tr>
<td>Specifies the feasibility tolerance</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies how frequently to print the solution progress</td>
<td>LOGFREQ=</td>
<td></td>
</tr>
<tr>
<td>Specifies the level of detail of solution progress to print in the log</td>
<td>LOGLEVEL=</td>
<td></td>
</tr>
<tr>
<td>Specifies the maximum number of iterations</td>
<td>MAXITER=</td>
<td></td>
</tr>
<tr>
<td>Specifies the time limit for the optimization process</td>
<td>MAXTIME=</td>
<td></td>
</tr>
<tr>
<td>Specifies the optimality tolerance</td>
<td>OPTTOL=</td>
<td></td>
</tr>
<tr>
<td>Enables or disables printing summary (OPTLP procedure only)</td>
<td>PRINTLEVEL=</td>
<td></td>
</tr>
<tr>
<td>Specifies whether time units are CPU time or real time</td>
<td>TIMETYPE=</td>
<td></td>
</tr>
<tr>
<td><strong>Algorithm Options</strong></td>
<td>SCALE=</td>
<td></td>
</tr>
<tr>
<td>Enables or disables scaling of the problem</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Decomposition Algorithm Options in the PROC OPTMILP Statement or the SOLVE WITH MILP Statement in PROC OPTMODEL

To solve a mixed integer linear program, you can specify the decomposition algorithm in a SOLVE WITH MILP statement in the OPTMODEL procedure or in a PROC OPTMILP statement in the OPTMILP procedure. To control the overall decomposition algorithm, you can specify one or more of the MILP solver options shown in Table 18.2. (As the table indicates, you can specify some options only in the PROC OPTMILP statement.)

The options in Table 18.2 control the overall process flow for solving a mixed integer linear program; they are equivalent to the options that are used in the OPTMILP and OPTMODEL procedures with standard methods. These options are called main solver options in this chapter. They are described in detail in the section “Syntax: MILP Solver” on page 609 and the section “Syntax: OPTMILP Procedure” on page 165.

Table 18.2 MILP Options in the PROC OPTMILP Statement or SOLVE WITH MILP Statement

<table>
<thead>
<tr>
<th>Description</th>
<th>option</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Table Options (OPTMILP procedure only)</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the input data table</td>
<td>DATA=</td>
</tr>
<tr>
<td>Specifies the constraint activities output data table</td>
<td>DUALOUT=</td>
</tr>
<tr>
<td>Description</td>
<td>option</td>
</tr>
<tr>
<td>----------------------------------------------------------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>Specifies whether the model is a maximization or minimization problem</td>
<td>OBJSENSE=</td>
</tr>
<tr>
<td>Specifies the primal solution input data table (warm start)</td>
<td>PRIMALIN=</td>
</tr>
<tr>
<td>Specifies the primal solution output data table</td>
<td>PRIMALOUT=</td>
</tr>
<tr>
<td><strong>Presolve Option</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the type of presolve</td>
<td>PRESOLVER=</td>
</tr>
<tr>
<td><strong>Control Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the stopping criterion based on an absolute objective gap</td>
<td>ABSOBJGAP=</td>
</tr>
<tr>
<td>Specifies the cutoff value for node removal</td>
<td>CUTOFF=</td>
</tr>
<tr>
<td>Emphasizes feasibility or optimality</td>
<td>EMPHASIS=</td>
</tr>
<tr>
<td>Specifies the maximum violation of variables and constraints</td>
<td>FEASTOL=</td>
</tr>
<tr>
<td>Specifies the maximum difference allowed between an integer variable’s value and an integer</td>
<td>INTTOL=</td>
</tr>
<tr>
<td>Specifies how frequently to print the node log</td>
<td>LOGFREQ=</td>
</tr>
<tr>
<td>Specifies the level of detail of solution progress to print in the log</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the maximum number of nodes to process</td>
<td>MAXNODES=</td>
</tr>
<tr>
<td>Specifies the maximum number of solutions to find</td>
<td>MAXSOLS=</td>
</tr>
<tr>
<td>Specifies the time limit for the optimization process</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Specifies the overall capacity for the number of active threads (per machine)</td>
<td>NTHREADS=</td>
</tr>
<tr>
<td>Specifies the tolerance to use in determining the optimality of nodes in the branch-and-bound tree</td>
<td>OPTTOL=</td>
</tr>
<tr>
<td>Uses the input primal solution (warm start) (OPTMODEL procedure only)</td>
<td>PRIMALIN</td>
</tr>
<tr>
<td>Enables or disables printing summary (OPTMILP procedure only)</td>
<td>PRINTLEVEL=</td>
</tr>
<tr>
<td>Specifies the probing level</td>
<td>PROBE=</td>
</tr>
<tr>
<td>Specifies the stopping criterion based on a relative objective gap</td>
<td>RELOBJGAP=</td>
</tr>
<tr>
<td>Specifies the scale of the problem matrix</td>
<td>SCALE=</td>
</tr>
<tr>
<td>Specifies the initial seed for the random number generator</td>
<td>SEED=</td>
</tr>
<tr>
<td>Specifies the stopping criterion based on a target objective value</td>
<td>TARGET=</td>
</tr>
<tr>
<td>Specifies whether time units are CPU time or real time</td>
<td>TIMETYPE=</td>
</tr>
<tr>
<td><strong>Control Options (OPTMODEL procedure only)</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the maximum number of solutions to return from the pool</td>
<td>MAXPOOLSOLS=</td>
</tr>
<tr>
<td><strong>Heuristics Option</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the primal heuristics level</td>
<td>HEURISTICS=</td>
</tr>
<tr>
<td><strong>Search Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the restarting strategy</td>
<td>RESTARTS=</td>
</tr>
<tr>
<td>Specifies the number of iterations to perform on each variable for strong branching</td>
<td>STRONGITER=</td>
</tr>
<tr>
<td>Specifies the number of candidates for strong branching</td>
<td>STRONGLEN=</td>
</tr>
<tr>
<td>Specifies the level of symmetry detection</td>
<td>SYMMETRY=</td>
</tr>
<tr>
<td>Specifies the rule for selecting the branching variable</td>
<td>VARSEL=</td>
</tr>
<tr>
<td><strong>Cut Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the cut level for all cuts</td>
<td>ALLCUTS=</td>
</tr>
<tr>
<td>Specifies the clique cut level</td>
<td>CUTCLIQUE=</td>
</tr>
<tr>
<td>Specifies the flow cover cut level</td>
<td>CUTCLIQUE=</td>
</tr>
<tr>
<td>Specifies the flow path cut level</td>
<td>CUTFLOWPATH=</td>
</tr>
<tr>
<td>Specifies the Gomory cut level</td>
<td>CUTGOMORY=</td>
</tr>
</tbody>
</table>
Table 18.2  (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies the generalized upper bound (GUB) cover cut level</td>
<td>CUTGUB=</td>
</tr>
<tr>
<td>Specifies the implied bounds cut level</td>
<td>CUTFIMPLIED=</td>
</tr>
<tr>
<td>Specifies the knapsack cover cut level</td>
<td>CUTKNAPSACK=</td>
</tr>
<tr>
<td>Specifies the lift-and-project cut level</td>
<td>CUTLAP=</td>
</tr>
<tr>
<td>Specifies the mixed lifted 0-1 cut level</td>
<td>CUTFMILIFTED=</td>
</tr>
<tr>
<td>Specifies the mixed integer rounding (MIR) cut level</td>
<td>CUTMIR=</td>
</tr>
<tr>
<td>Specifies the multicommodity network flow cut level</td>
<td>CUTMULTICOMMODITY=</td>
</tr>
<tr>
<td>Specifies the row multiplier factor for cuts</td>
<td>CUTSFACOR=</td>
</tr>
<tr>
<td>Specifies the overall cut aggressiveness</td>
<td>CUTSTRATEGY=</td>
</tr>
<tr>
<td>Specifies the zero-half cut level</td>
<td>CUTZEROHALF=</td>
</tr>
</tbody>
</table>

The HYBRID= option in the DECOMP statement indicates the processing mode for the root node of the branch-and-bound search tree. When HYBRID=TRUE, the root node is first processed using standard MILP techniques, as described in the section “Details: MILP Solver” on page 621. When HYBRID=FALSE, the root processing is done solely by the decomposition algorithm, and the following direct MILP options are ignored: CUTOFF=, EMPHASIS=, RESTARTS=, SEED=, TARGET=, and all the cut options.

The following search options, listed in Table 18.2, have a different interpretation or a different set of options from what is described in the MILP solver sections:

**LOGFREQ=number**

**PRINTFREQ=number**

specifies how often to print information in the node log. The value of *number* can be any 32-bit integer greater than or equal to 0. The default value is 10. If *number* is 0, then the node log is disabled. If *number* is positive, then an entry is made in the node log at the first node, at the last node, and at intervals dictated by the value of *number*. An entry is also made in the node log each time the solver finds a better integer solution or improved bound.

**STRONGITER=number | AUTOMATIC**

specifies the number of pricing iterations that are performed for each variable in the candidate list when you use the strong branching variable selection strategy. The value of *number* can be any positive 32-bit integer. If you specify the keyword AUTOMATIC, the MILP solver uses the default value, which is calculated automatically.

**VARSEL=AUTOMATIC | MAXINFEAS | PSEUDO | RYANFOSTER | STRONG**

specifies the rule for selecting the branching variable. You can specify the following values:

- **AUTOMATIC** uses automatic branching variable selection.
- **MAXINFEAS** selects the variable in the original compact formulation with maximum infeasibility.
- **PSEUDO** selects the variable in the original compact formulation that maximizes the weighted up and down pseudocosts.
- **RYANFOSTER** when appropriate, uses a specialized branching rule known as *Ryan-Foster branching*.
- **STRONG** selects the variable in the original compact formulation that maximizes the estimated improvement in the objective value based on strong branching.
By default, VARSEL=AUTOMATIC. For more information about variable selection, see the sections “Variable Selection” on page 181 and “Special Case: Identical Blocks and Ryan-Foster Branching” on page 873.

**DECOMP Statement**

```
DECOMP < decomp-options > ;
DECOMPOSITION < decomp-options > ;
```

The DECOMP statement controls the overall decomposition algorithm.

Table 18.3 summarizes the `decomp-options` available in the DECOMP statement. These options control the overall decomposition algorithm process flow during the solution of an LP or a MILP. (As the table indicates, you can specify the data table options only in the OPTLP or OPTMILP procedure, and you can specify some control options only for a MILP.)

**Table 18.3 Options in the DECOMP Statement**

<table>
<thead>
<tr>
<th>Description</th>
<th>decomp-option</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data Table Options (OPTLP and OPTMILP procedures only)</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the blocks input data table</td>
<td>BLOCKS=</td>
</tr>
<tr>
<td><strong>Control Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the stopping criterion based on an absolute objective gap</td>
<td>ABSOBJGAP=</td>
</tr>
<tr>
<td>Specifies the level of detail of solution progress to print in the log</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the maximum number of blocks to allow</td>
<td>MAXBLOCKS=</td>
</tr>
<tr>
<td>Specifies the block-specification method</td>
<td>METHOD=</td>
</tr>
<tr>
<td>Specifies the number of blocks to search for by using METHOD=AUTO</td>
<td>NBLOCKS=</td>
</tr>
<tr>
<td>Specifies the number of block threads to use in the decomposition algorithm</td>
<td>NTHREADS=</td>
</tr>
<tr>
<td>Specifies the stopping criterion based on a relative objective gap</td>
<td>RELOBJGAP=</td>
</tr>
<tr>
<td><strong>Control Options (MILP only)</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies whether to process the root node by using standard MILP techniques</td>
<td>HYBRID=</td>
</tr>
<tr>
<td>Specifies how frequently to print the continuous iteration log</td>
<td>LOGFREQ=</td>
</tr>
<tr>
<td>Specifies the maximum number of outer iterations for the decomposition algorithm</td>
<td>MAXITER=</td>
</tr>
</tbody>
</table>

You can specify the following `decomp-options`:

- **ABSOBJGAP=** *number*
- **ABSOLUTEOBJECTIVEGAP=** *number*
  specifies a stopping criterion for the continuous bound of the decomposition. When the absolute difference between the master objective and the best dual bound falls below *number*, the decomposition algorithm stops adding columns. The value of *number* can be any nonnegative number. The default value is the value of the OPTTOL= main solver option.

- **BLOCKS=** *CAS-libref.data-table*
  specifies (for OPTLP and OPTMILP procedures only) the input data table that contains block definitions to use in the decomposition algorithm if METHOD=USER. For more information, see the section “The BLOCKS= Data Table in PROC OPTMILP and PROC OPTLP” on page 871. To specify blocks in
PROC OPTMODEL, use the .block constraint suffix instead (see the section “The .block Constraint Suffix in PROC OPTMODEL” on page 872). CAS-libref.data-table is a two-level name, where CAS-libref refers to the caslib and session identifier, and data-table specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 19 in Chapter 4, “Shared Concepts.”

**HYBRID=FALSE | TRUE**

specifies whether to first process the root node by using standard MILP techniques, as described in the section “Details: MILP Solver” on page 621. You can specify the following values:

- **FALSE**  
  disables root processing by using standard MILP techniques.

- **TRUE**  
  enables root processing by using standard MILP techniques.

By default, HYBRID=TRUE if METHOD=AUTO, and HYBRID=FALSE otherwise.

**LOGFREQ=number**

specifies (for MILP problems only) how often to print information in the continuous iteration log. The value of number can be any nonnegative 32-bit integer. The default value of number is 10. If number is 0, then the iteration log is disabled. If number is positive, then an entry is made in the log at the first iteration, at the last iteration, and at intervals that are dictated by the value of number. An entry is also made each time a better integer solution or improved bound is found.

**LOGLEVEL=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE**

controls the amount of information that the decomposition algorithm displays in the SAS log.

You can specify the following values for an LP:

- **AUTOMATIC**  
  prints the continuous iteration log at the interval dictated by the LOGFREQ= main solver option.

- **NONE**  
  turns off printing of all the decomposition algorithm messages to the SAS log.

- **BASIC**  
  prints the continuous iteration log at the interval dictated by the LOGFREQ= main solver option.

- **MODERATE**  
  prints the continuous iteration log and summary information for each iteration at the interval dictated by the LOGFREQ= main solver option.

- **AGGRESSIVE**  
  prints the continuous iteration log and detailed information for each iteration at the interval dictated by the LOGFREQ= main solver option.

You can specify the following values for a MILP:

- **AUTOMATIC**  
  prints the continuous iteration log for the root node at the interval dictated by the LOGFREQ= option in the DECOMP statement. Prints the branch-and-bound node log at the interval dictated by the LOGFREQ= main solver option.

- **NONE**  
  turns off printing of all the decomposition algorithm messages to the SAS log.

- **BASIC**  
  prints the continuous iteration log for each branch-and-bound node at the interval dictated by the LOGFREQ= option in the DECOMP statement.

- **MODERATE**  
  prints the continuous iteration log and summary information for each iteration of each branch-and-bound node at the interval dictated by the LOGFREQ= option in the DECOMP statement.
AGGRESSIVE  prints the continuous iteration log and detailed information for each iteration of each branch-and-bound node at the interval dictated by the LOGFREQ= option in the DECOMP statement.

By default, LOGLEVEL=AUTOMATIC for both LPs and MILPs.

MAXBLOCKS=number  specifies the maximum number of blocks to allow. If the defined number of blocks exceeds number, the algorithm creates superblocks by using a very simple round-robin scheme. The value of number can be any positive 32-bit integer. The default is the largest number that can be represented by a 32-bit integer.

MAXITER=number  specifies (for MILP problems only) the maximum number of outer iterations for the decomposition algorithm. The value of number can be any positive 32-bit integer. The default is the largest number that can be represented by a 32-bit integer.

METHOD=AUTO | CONCOMP | NETWORK | SET | USER  specifies the block-specification method. You can specify the following values:

AUTO  attempts to find a block-angular structure in the constraint matrix by using matrix-stretching techniques similar to what is described in Grcar (1990) and Aykanat, Pinar, and Çatalyürek (2004). The NBLOCKS= option specifies the number of blocks into which the algorithm attempts to decompose the constraint matrix. If the algorithm fails to find a decomposition, the MILP solver is called directly.

CONCOMP  attempts to find a block-diagonal (not block-angular) structure in the constraint matrix. Unless your problem separates into completely independent problems with no linking constraints, this method finds only one block and hence is equivalent to calling the MILP solver directly.

NETWORK  attempts to find an embedded network similar to what is described in the section “The Network Simplex Algorithm” on page 511. The weakly connected components of this network are used as the blocks.

SET  attempts to find a set partitioning or set covering structure in the constraint matrix and defines this as the master (linking) constraints. The weakly connected components of the remaining constraints are used as the blocks.

USER  uses a user-defined method to specify which rows belong to which blocks (subproblems). In PROC OPTMODEL, use the .block constraint suffix. In PROC OPTLP and PROC OPTMILP, use the BLOCKS= data table instead.

By default, METHOD=USER if blocks are defined, and METHOD=AUTO otherwise.

NBLOCKS=number  NUMBLOCKS=number  specifies the initial number of blocks to search for when you specify METHOD=AUTO. If the algorithm is unable to find a block-angular structure that contains this number of blocks, it repeatedly attempts to find an appropriate structure that contains half the previously attempted number of blocks. If the algorithm fails to find a decomposition that contains at least two blocks, then the standard MILP solver is called directly. The value of number can be any positive number less than or equal to the number
of rows in the presolved model; the default value is the number of block threads that are used for processing. In single-machine mode, this is equivalent to the value of the NTHREADS= option in the DECOMP statement. In distributed mode, this is equivalent to the number of workers in your session times the value of the NTHREADS= option in the DECOMP statement (that is, the number of block threads that you specify for each compute node). For more information about parallel execution, see the section “Parallel Processing” on page 873.

**NTHREADS=number**

**NUMTHREADS=number**

specifies the number of block threads to use in the decomposition algorithm. The value of the NTHREADS= option in the main solver statement serves as the overall capacity for the number of active threads that can run at one time. By default, the number of block threads is \( t = \min(p, d, b) \), where \( p \) is the value of the NTHREADS= option in the main solver statement, \( d \) is the value of the NTHREADS= option in the DECOMP statement, and \( b \) is the number of blocks that the decomposition algorithm sets or finds.

**RELOBJGAP=number**

specifies the relative objective gap as a stopping criterion. The relative objective gap is based on the master objective (MasterObjective) and the best dual bound (BestBound); it is equal to

\[
\frac{|\text{MasterObjective} - \text{BestBound}|}{10^{-10} + |\text{BestBound}|}
\]

When this value becomes smaller than the specified gap size **number**, the decomposition algorithm stops adding columns. The value of **number** can be any nonnegative number. For LP, the default value is 0; for MILP, the default value is 1E–4.

---

### DECOMPMASTER Statement

**DECOMPMASTER**

**DECOMPOSITIONMASTER**

**MASTER**

The DECOMPMASTER statement controls the master problem.

Table 18.4 summarizes the options available in the DECOMPMASTER statement. These options control the master LP solver in the decomposition algorithm during the solution of an LP or a MILP. (As the table indicates, you can specify the PRINTLEVEL= option only in the OPTLP procedure.) For descriptions of these options, see the section “LP Solver Options” on page 505 in Chapter 12, “The Linear Programming Solver,” and the section “PROC OPTLP Statement” on page 112 in Chapter 6, “The OPTLP Procedure.” Some options have different defaults when you use the decomposition algorithm, as indicated in Table 18.4.
### Table 18.4 Options in the DECOMPMASTER Statement

<table>
<thead>
<tr>
<th>Description</th>
<th>master-option</th>
<th>Different Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm Option</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the master algorithm</td>
<td>ALGORITHM=</td>
<td>PS†</td>
</tr>
<tr>
<td><strong>Presolve Option</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Controls the dualization of the problem</td>
<td>DUALIZE=</td>
<td>OFF</td>
</tr>
<tr>
<td>Specifies, for the first master solve only, the type of presolve</td>
<td>INITPRESOLVER=</td>
<td></td>
</tr>
<tr>
<td>Specifies the type of presolve</td>
<td>PRESOLVER=</td>
<td>NONE†</td>
</tr>
<tr>
<td><strong>Control Options</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the feasibility tolerance</td>
<td>FEASTOL=</td>
<td>1E–7</td>
</tr>
<tr>
<td>Specifies how frequently to print the solution progress</td>
<td>LOGFREQ=</td>
<td></td>
</tr>
<tr>
<td>Specifies the level of detail of solution progress to print in the log</td>
<td>LOGLEVEL=</td>
<td>NONE</td>
</tr>
<tr>
<td>Specifies the time limit for the optimization process</td>
<td>MAXTIME=</td>
<td></td>
</tr>
<tr>
<td>Specifies the number of threads to use in the master solver</td>
<td>NTHREADS=</td>
<td></td>
</tr>
<tr>
<td>Specifies the optimality tolerance</td>
<td>OPTTOL=</td>
<td>1E–7</td>
</tr>
<tr>
<td>Enables or disables printing summary (OPTLP procedure only)</td>
<td>PRINTLEVEL=</td>
<td></td>
</tr>
<tr>
<td>Specifies whether time units are CPU time or real time</td>
<td>TIMETYPE=</td>
<td>WARMSTART†</td>
</tr>
<tr>
<td>Specifies the type of initial basis</td>
<td>BASIS=</td>
<td></td>
</tr>
<tr>
<td>Specifies the type of pricing strategy</td>
<td>PRICETYPE=</td>
<td></td>
</tr>
<tr>
<td>Specifies the queue size for determining the entering variable</td>
<td>QUEUESIZE=</td>
<td></td>
</tr>
<tr>
<td>Enables or disables scaling of the problem</td>
<td>SCALE=</td>
<td></td>
</tr>
<tr>
<td>Specifies the initial seed for the random number generator</td>
<td>SEED=</td>
<td></td>
</tr>
<tr>
<td><strong>Interior Point Algorithm Options</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Enables or disables interior crossover</td>
<td>CROSSOVER=</td>
<td></td>
</tr>
<tr>
<td>Specifies the stopping criterion based on a duality gap</td>
<td>DUALITYGAP=</td>
<td></td>
</tr>
</tbody>
</table>

† The reason for the different defaults (ALGORITHM=PS, PRESOLVER=NONE, and BASIS=WARMSTART) is that primal feasibility of the master problem is preserved when columns are added, so a warm start from the previous optimal basis tends to be more efficient than solving the master from scratch at each iteration.

The following options, listed in Table 18.4, are specific to the DECOMPMASTER statement and are not described in the LP solver sections:

**INITPRESOLVER**=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE

**INITPRESOL**=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE

specifies, for the first master solve only, the presolve level. You can specify the following values:

- **AUTOMATIC** applies the default level of presolve processing.
- **NONE** disables the presolver.
- **BASIC** performs minimal presolve processing.
- **MODERATE** applies a higher level of presolve processing.
- **AGGRESSIVE** applies the highest level of presolve processing.

By default, INITPRESOLVER=AUTOMATIC.
**DECOMPMASTERIP Statement**

**DECOMPMASTERIP** `<masterip-options> ;`  
**DECOMPOSITIONMASTERIP** `<masterip-options> ;`  
**MASTERIP** `<masterip-options> ;`

For mixed integer linear programming problems, the DECOMPMASTERIP statement controls the (restricted) master problem, which is solved as a MILP with the current set of columns in an effort to obtain an integer feasible solution.

Table 18.5 summarizes the options available in the DECOMPMASTERIP statement. These options control the MILP solver that is used to solve the integer version of the master problem. For descriptions of these options, see the section “MILP Solver Options” on page 611 in Chapter 14, “The Mixed Integer Linear Programming Solver,” and the section “PROC OPTMILP Statement” on page 166 in Chapter 7, “The OPTMILP Procedure.” Some options have different defaults when you use the decomposition algorithm, as shown in Table 18.5.

<table>
<thead>
<tr>
<th>Description</th>
<th>masterip-option</th>
<th>Different Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Presolve Option</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the type of presolve</td>
<td>PRESOLVER=</td>
<td></td>
</tr>
<tr>
<td><strong>Control Options</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the stopping criterion based on an absolute objective gap</td>
<td>ABSOBJGAP=</td>
<td></td>
</tr>
<tr>
<td>Specifies the cutoff value for node removal</td>
<td>CUTOFF=</td>
<td></td>
</tr>
<tr>
<td>Emphasizes feasibility or optimality</td>
<td>EMPHASIS=</td>
<td></td>
</tr>
<tr>
<td>Specifies the maximum violation on variables and constraints</td>
<td>FEASTOL=</td>
<td>1E–7</td>
</tr>
<tr>
<td>Specifies the maximum difference allowed between an integer variable’s value and an integer</td>
<td>INTTOL=</td>
<td></td>
</tr>
<tr>
<td>Specifies how frequently to print the node log</td>
<td>LOGFREQ=</td>
<td></td>
</tr>
<tr>
<td>Specifies the level of detail of solution progress to print in the log</td>
<td>LOGLEVEL=</td>
<td>NONE</td>
</tr>
<tr>
<td>Specifies the maximum number of nodes to process</td>
<td>MAXNODES=†</td>
<td></td>
</tr>
<tr>
<td>Specifies the maximum number of solutions to find</td>
<td>MAXSOLS=</td>
<td></td>
</tr>
<tr>
<td>Specifies the time limit for the optimization process</td>
<td>MAXTIME=</td>
<td></td>
</tr>
<tr>
<td>Specifies the number of threads to use in the master integer solver</td>
<td>NTHREADS=</td>
<td></td>
</tr>
<tr>
<td>Specifies the tolerance to use in determining the optimality of nodes in the branch-and-bound tree</td>
<td>OPTTOL=</td>
<td>1E–7</td>
</tr>
<tr>
<td>Specifies whether to use the previous best primal solution as a warm start</td>
<td>PRIMALIN=</td>
<td></td>
</tr>
</tbody>
</table>
### Table 18.5 (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>masterip-option</th>
<th>Different Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies the probing level</td>
<td>PROBE=</td>
<td></td>
</tr>
<tr>
<td>Specifies the stopping criterion based on a relative objective gap</td>
<td>RELOBJGAP= 0.01</td>
<td></td>
</tr>
<tr>
<td>Specifies the scale of the problem matrix</td>
<td>SCALE=</td>
<td></td>
</tr>
<tr>
<td>Specifies the initial seed for the random number generator</td>
<td>SEED=</td>
<td></td>
</tr>
<tr>
<td>Specifies the stopping criterion based on a target objective value</td>
<td>TARGET=</td>
<td></td>
</tr>
<tr>
<td>Specifies whether time units are CPU time or real time</td>
<td>TIMETYPE=</td>
<td></td>
</tr>
<tr>
<td><strong>Heuristics Option</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the primal heuristics level</td>
<td>HEURISTICS=</td>
<td></td>
</tr>
<tr>
<td><strong>Search Options</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the level of conflict search</td>
<td>CONFLICTSEARCH=</td>
<td></td>
</tr>
<tr>
<td>Specifies the node selection strategy</td>
<td>NODESEL=</td>
<td></td>
</tr>
<tr>
<td>Specifies the restarting strategy</td>
<td>RESTARTS=</td>
<td></td>
</tr>
<tr>
<td>Specifies the number of iterations to perform on each variable for</td>
<td>STRONGITER=</td>
<td></td>
</tr>
<tr>
<td>strong branching strategy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the number of candidates for strong branching</td>
<td>STRONGLEN=</td>
<td></td>
</tr>
<tr>
<td>Specifies the level of symmetry detection</td>
<td>SYMMETRY=</td>
<td></td>
</tr>
<tr>
<td>Specifies the rule for selecting branching variable</td>
<td>VARSEL=</td>
<td></td>
</tr>
<tr>
<td><strong>Cut Options</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the cut level for all cuts</td>
<td>ALLCUTS=</td>
<td></td>
</tr>
<tr>
<td>Specifies the clique cut level</td>
<td>CUTCLIQUE=</td>
<td></td>
</tr>
<tr>
<td>Specifies the flow cover cut level</td>
<td>CUTFLOWCOVER=</td>
<td></td>
</tr>
<tr>
<td>Specifies the flow path cut level</td>
<td>CUTFLOWPATH=</td>
<td></td>
</tr>
<tr>
<td>Specifies the Gomory cut level</td>
<td>CUTGOMORY=</td>
<td></td>
</tr>
<tr>
<td>Specifies the generalized upper bound (GUB) cover cut level</td>
<td>CUTGUB=</td>
<td></td>
</tr>
<tr>
<td>Specifies the implied bounds cut level</td>
<td>CUTIMPLIED=</td>
<td></td>
</tr>
<tr>
<td>Specifies the knapsack cover cut level</td>
<td>CUTKNAPSACK=</td>
<td></td>
</tr>
<tr>
<td>Specifies the lift-and-project cut level</td>
<td>CUTLAP=</td>
<td></td>
</tr>
<tr>
<td>Specifies the mixed lifted 0-1 cut level</td>
<td>CUTMLIFTED=</td>
<td></td>
</tr>
<tr>
<td>Specifies the mixed integer rounding (MIR) cut level</td>
<td>CUTFIR=</td>
<td></td>
</tr>
<tr>
<td>Specifies the multicommodity network flow cut level</td>
<td>CUTMULTICOMMODITY=</td>
<td></td>
</tr>
<tr>
<td>Specifies the row multiplier factor for cuts</td>
<td>CUTSFACOR=</td>
<td></td>
</tr>
<tr>
<td>Specifies the overall cut aggressiveness</td>
<td>CUTSTRATEGY=</td>
<td></td>
</tr>
<tr>
<td>Specifies the zero-half cut level</td>
<td>CUTZEROHALF=</td>
<td></td>
</tr>
</tbody>
</table>

† MAXNODES=100000 in the root node, and MAXNODES=10000 in nodes that are not the root.

The following options are listed in Table 18.5 but are not described in the MILP solver sections. These options are specific to the DECOMPMASTERIP statement.

**NTHREADS=number**

**NUMTHREADS=number**

specifies the number of threads to use in the master integer solver (if the selected solver method supports multithreading). The value of the NTHREADS= option in the main solver statement serves as
the overall capacity for the number of active threads that can run at one time. By default, the number of master integer threads is the value of the NTHREADS= option in the main solver statement.

**PRIMALIN=FALSE | TRUE**

**PIN=FALSE | TRUE**
specifies whether the MILP solver is to use the previous best solution’s variables values as a starting solution (warm start). If the MILP solver finds that the input solution is feasible, then the input solution provides an incumbent solution and a bound for the branch-and-bound algorithm. If the solution is not feasible, the MILP solver tries to repair it. When it is difficult to find a good integer feasible solution for a problem, a warm start can reduce solution time significantly. You can specify the following values:

**FALSE**
ignores the previous solution.

**TRUE**
starts from the previous solution.

By default, PRIMALIN=TRUE.

---

**DECOMPSUBPROB Statement**

**DECOMPSUBPROB**<sub>subprob-options</sub>;  
**DECOMPOSITIONSUBPROB**<sub>subprob-options</sub>;  
**SUBPROB**<sub>subprob-options</sub>

The DECOMPSUBPROB statement controls the subproblem.

Table 18.6 summarizes the options available for the decomposition algorithm in the DECOMPSUBPROB statement when the chosen subproblem algorithm is an LP algorithm. (As the table indicates, you can specify the PRINTLEVEL= option only in the OPTLP procedure.) For descriptions of these options, see the section “LP Solver Options” on page 505 in Chapter 12, “The Linear Programming Solver,” and the section “PROC OPTLP Statement” on page 112 in Chapter 6, “The OPTLP Procedure.” Some options have different defaults when you use the decomposition algorithm, as shown in Table 18.6.

<table>
<thead>
<tr>
<th><strong>Description</strong></th>
<th><strong>subprob-option</strong></th>
<th><strong>Different Default</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm Option</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the subproblem algorithm</td>
<td>ALGORITHM=</td>
<td>+</td>
</tr>
<tr>
<td><strong>Presolve Option</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Controls the dualization of the problem</td>
<td>DUALIZE=</td>
<td>OFF</td>
</tr>
<tr>
<td>Specifies, for the first subproblem solve only, the type of presolve</td>
<td>INITPRESOLVER=</td>
<td></td>
</tr>
<tr>
<td>Specifies the type of presolve</td>
<td>PRESOLVER=</td>
<td>NONE†</td>
</tr>
<tr>
<td><strong>Control Options</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the feasibility tolerance</td>
<td>FEASTOL=</td>
<td>1E-7</td>
</tr>
<tr>
<td>Specifies how frequently to print the solution progress</td>
<td>LOGFREQ=</td>
<td></td>
</tr>
<tr>
<td>Specifies the level of detail of solution progress to print in the log</td>
<td>LOGLEVEL=</td>
<td>NONE</td>
</tr>
</tbody>
</table>
Table 18.6  (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>subprob-option</th>
<th>Different Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies the time limit for the optimization process</td>
<td>MAXTIME=</td>
<td></td>
</tr>
<tr>
<td>Specifies the number of threads to use in the subproblem solver</td>
<td>NTHREADS=</td>
<td></td>
</tr>
<tr>
<td>Specifies the optimality tolerance</td>
<td>OPTTOL=</td>
<td>1E–7</td>
</tr>
<tr>
<td>Enables or disables printing summary (OPTLP procedure only)</td>
<td>PRINTLEVEL=</td>
<td></td>
</tr>
<tr>
<td>Specifies the initial seed for the random number generator</td>
<td>SEED=</td>
<td></td>
</tr>
<tr>
<td>Specifies whether time units are CPU time or real time</td>
<td>TIMETYPE=</td>
<td></td>
</tr>
</tbody>
</table>

**Simplex Algorithm Options**

- Specifies the type of initial basis                                      | BASIS=              | WARMSTART†       |
- Specifies the type of pricing strategy                                    | PRICETYPE=          |                   |
- Specifies the queue size for determining entering variable               | QUEUESIZE=          |                   |
- Enables or disables scaling of the problem                                | SCALE=              |                   |

**Interior Point Algorithm Options**

- Enables or disables interior crossover                                    | CROSSOVER=          |                   |
- Specifies the stopping criterion based on a duality gap                   | DUALITYGAP=         |                   |

† When you specify METHOD=USER or METHOD=AUTO in the DECOMP statement, ALGORITHM=PS, PRESOLVER=NONE, and BASIS=WARMSTART by default. The reason for these defaults is that primal feasibility of the subproblem is preserved when the objective is changed, so a warm start from the previous optimal basis tends to be more efficient than solving the subproblem from scratch at each iteration. When METHOD=NETWORK, ALGORITHM=NETWORKPURE by default because each subproblem is a pure network, causing the specialized pure network solver to usually be the most efficient choice. When METHOD=CONCOMP, ALGORITHM=DS by default because dual simplex generally has the best performance on linear programs and, in this case, one outer iteration is sufficient (that is, warm starts are not required).

Table 18.7 summarizes the options available in the DECOMPSUBPROB statement when the chosen subproblem algorithm is a MILP algorithm. When the subproblem consists of multiple blocks (a block-diagonal structure), these settings apply to all subproblems. For descriptions of these options, see the section “MILP Solver Options” on page 611 in Chapter 14, “The Mixed Integer Linear Programming Solver,” and the section “PROC OPTMILP Statement” on page 166 in Chapter 7, “The OPTMILP Procedure.”

Table 18.7  Options in the DECOMPSUBPROB Statement Used with a MILP Algorithm

<table>
<thead>
<tr>
<th>Description</th>
<th>subprob-option</th>
<th>Different Default</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm Option</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the subproblem algorithm</td>
<td>ALGORITHM=</td>
<td></td>
</tr>
<tr>
<td><strong>Presolve Option</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies, for the first subproblem solve only, the type of presolve</td>
<td>INITPRESOLVER=</td>
<td></td>
</tr>
<tr>
<td>Specifies the type of presolve</td>
<td>PRESOLVER=</td>
<td></td>
</tr>
<tr>
<td><strong>Control Options</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the stopping criterion based on an absolute objective gap</td>
<td>ABSOBJGAP=</td>
<td></td>
</tr>
<tr>
<td>Emphasizes feasibility or optimality</td>
<td>EMPHASIS=</td>
<td></td>
</tr>
<tr>
<td>Specifies the maximum violation on variables and constraints</td>
<td>FEASTOL=</td>
<td>1E–7</td>
</tr>
<tr>
<td>Specifies the maximum difference allowed between an integer variable’s value and an integer</td>
<td>INTTOL=</td>
<td></td>
</tr>
</tbody>
</table>
### Table 18.7  (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>subprob-option</th>
<th>Different Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies how frequently to print the node log</td>
<td>LOGFREQ=</td>
<td></td>
</tr>
<tr>
<td>Specifies the level of detail of solution progress to print in the log</td>
<td>LOGLEVEL=</td>
<td></td>
</tr>
<tr>
<td>Specifies the maximum number of nodes to process</td>
<td>MAXNODES=</td>
<td></td>
</tr>
<tr>
<td>Specifies the maximum number of solutions to find</td>
<td>MAXSOLS=</td>
<td></td>
</tr>
<tr>
<td>Specifies the time limit for the optimization process</td>
<td>MAXTIME=</td>
<td></td>
</tr>
<tr>
<td>Specifies the number of threads to use in the subproblem solver</td>
<td>NTHREADS=</td>
<td></td>
</tr>
<tr>
<td>Specifies the tolerance to use in determining the optimality of nodes in the branch-and-bound tree</td>
<td>OPTTOL= 1E–7</td>
<td></td>
</tr>
<tr>
<td>Specifies whether to use the previous best primal solution as a warm start</td>
<td>PRIMALIN=</td>
<td></td>
</tr>
<tr>
<td>Specifies the probing level</td>
<td>PROBE=</td>
<td></td>
</tr>
<tr>
<td>Specifies the stopping criterion based on a relative objective gap</td>
<td>RELOBJGAP=</td>
<td></td>
</tr>
<tr>
<td>Specifies the scale of the problem matrix</td>
<td>SCALE=</td>
<td></td>
</tr>
<tr>
<td>Specifies the initial seed for the random number generator</td>
<td>SEED=</td>
<td></td>
</tr>
<tr>
<td>Specifies the stopping criterion based on a target objective value</td>
<td>TARGET=</td>
<td></td>
</tr>
<tr>
<td>Specifies whether time units are CPU time or real time</td>
<td>TIMETYPE=</td>
<td></td>
</tr>
</tbody>
</table>

#### Heuristics Option

- Specifies the primal heuristics level
  - HEURISTICS=          

#### Search Options

- Specifies the level of conflict search
  - CONFLICTSEARCH=     
- Specifies the node selection strategy
  - NODESEL=            
- Specifies the restarting strategy
  - RESTARTS=           
- Specifies the number of iterations to perform for strong branching
  - STRONGITER=         
- Specifies the number of candidates for strong branching
  - STRONGLEN=          
- Specifies the level of symmetry detection
  - SYMMETRY=           
- Specifies the rule for selecting branching variable
  - VARSEL=             

#### Cut Options

- Specifies the cut level for all cuts
  - ALLCUTS=            
- Specifies the clique cut level
  - CUTCLIQUE=          
- Specifies the flow cover cut level
  - CUTFLOWCOVER=       
- Specifies the flow path cut level
  - CUTFLOWPATH=        
- Specifies the Gomory cut level
  - CUTGOMORY=          
- Specifies the generalized upper bound (GUB) cover cut level
  - CUTGUB=             
- Specifies the implied bounds cut level
  - CUTIMPLIED=         
- Specifies the knapsack cover cut level
  - CUTKNAPSACK=        
- Specifies the lift-and-project cut level
  - CTULAP=             
- Specifies the mixed lifted 0-1 cut level
  - CUTMLILIFTED=       
- Specifies the mixed integer rounding (MIR) cut level
  - CUTFMIR=            
- Specifies the multicommodity network flow cut level
  - CUTMULTICOMMODITY=  
- Specifies the row multiplier factor for cuts
  - CUTSFACOR=          
- Specifies the overall cut aggressiveness
  - CUTSTRATEGY=        
- Specifies the zero-half cut level
  - CUTFZEROHALF=       |
Chapter 18: The Decomposition Algorithm

The following options, listed in Table 18.6 and Table 18.7, are specific to the DECOMPSUBPROB statement and are not described in the LP or MILP solver sections:

**ALGORITHM**=string
**SOLVER**=string
**SOL**=string

specifies the algorithm to use for the subproblem solves. You can specify the following values (the valid abbreviated value for each string follows the vertical bar):

- **PRIMAL** | **PS** uses the primal simplex algorithm.
- **DUAL** | **DS** uses the dual simplex algorithm.
- **NETWORK** | **NS** uses the network simplex algorithm.
- **NETWORKPURE** | **NSPURE** uses the network simplex algorithm for pure networks.
- **INTERIORPOINT** | **IP** uses the interior point algorithm.
- **MILP** uses the mixed integer linear solver.

By default, ALGORITHM=NETWORKPURE if METHOD=NETWORK, ALGORITHM=MILP for mixed integer linear programming subproblems, and ALGORITHM=PS for linear programming subproblems.

**INITPRESOLVER**=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE
**INITPRESOL**=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE

specifies, for the first subproblem solve only, the presolve level. You can specify the following values:

- **AUTOMATIC** applies the default level of presolve processing.
- **NONE** disables the presolver.
- **BASIC** performs minimal presolve processing.
- **MODERATE** applies a higher level of presolve processing.
- **AGGRESSIVE** applies the highest level of presolve processing.

By default, INITPRESOLVER=AUTOMATIC.

**NTHREADS**=number
**NUMTHREADS**=number

specifies the number of threads to use in the subproblem solver (if the selected solver method supports multithreading). The value of the NTHREADS= option in the main solver statement serves as the overall capacity for the number of active threads that can run at one time. By default, the number of subproblem threads is \( t = \max(1, \lfloor p/n \rfloor) \), where \( p \) is the value of the NTHREADS= option in the main solver statement; \( n = \min(p, \lfloor d/m \rfloor) \), which is the number of blocks being processed simultaneously; \( d \) is the number of block threads; and \( m \) is the number of compute nodes (which can be more than one, when you run the decomposition algorithm in distributed mode).
Details: Decomposition Algorithm

Data Input

This subsection describes the format for describing the partition of the constraint system that defines the subproblem blocks. In the OPTLP and OPTMILP procedures, partitioning is done by using a data table specified in the BLOCKS= data option in the DECOMP statement. In PROC OPTMODEL, partitioning is done by using the .block suffix on constraints.

The blocks must be disjoint with respect to variables. If two blocks contain a nonzero coefficient for the same variable, the decomposition algorithm produces an error that contains information about where the blocks overlap.

The BLOCKS= Data Table in PROC OPTMILP and PROC OPTLP

The BLOCKS= data table has two required variables:

_ROW_
specifies the constraint (row) names of the problem. The values should be a subset of the row names in the DATA= data table for the current problem.

_BLOCK_
specifies the numeric block identifier for each constraint in the problem. A missing observation or missing value indicates a master (linking) constraint that does not appear in any block. Listing the linking constraints is optional. The block identifiers must start from 0 and be consecutive.

See the section “Solving a MILP with DECOMP and PROC OPTMILP” on page 854 for an example of using this BLOCKS= data table with PROC OPTMILP.
The .block Constraint Suffix in PROC OPTMODEL

The .block constraint suffix specifies the numeric block identifier for each constraint in the problem. The block identifiers do not need to start from 0, nor do they need to be consecutive. Master (linking) constraints can be identified by using a missing value. Listing the linking constraints is optional.

See the section “Solving a MILP with DECOMP and PROC OPTMODEL” on page 852 for an example of using the .block constraint suffix with PROC OPTMODEL.

Decomposition Algorithm

The decomposition algorithm for LPs is based on the original Dantzig-Wolfe method (Dantzig and Wolfe 1960). Embedding this method in the context of a branch-and-bound algorithm for MILPs is described in Barnhart et al. (1998) and is often referred to as branch-and-price. The design of a framework that allows for building a generic branch-and-price solver that requires only the original (compact) formulation and the constraint partition was first proposed independently by Ralphs and Galati (2006) and Vanderbeck and Savelsbergh (2006). This method is also commonly referred to as column generation, although the algorithm implemented here is only one specific variant of this wider class of algorithms.

The algorithm setup starts by forming various components that are used iteratively during the solver process. These components include the master problem (controlled by options in the DECOMPMASTER statement), one subproblem for each block (controlled by options in the DECOMPSUBPROB statement) and, for MILPs, the integer version of the master problem (controlled by options in the DECOMPMASTERIP statement).

The master problem is a linear program that is defined over a potentially large number of variables that represent the weights of a convex combination. The points in the convex combination satisfy the constraints that are defined in the subproblem. The master constraints of the original problem are enforced in this reformulated space. In this sense, the decomposition algorithm takes the intersection of two polyhedra: one defined by original master constraints and one defined by the subproblem constraints. Since the set of variables needed to define the intersection of the polyhedra can be large, the algorithm works on a restricted subset and generates only those variables (columns) that have good potential with respect to feasibility and optimality. This generation is done by using the dual information that is obtained by solving the master problem to price out new variables. These new variables are generated by solving the subproblems over the appropriate cost vector (the reduced cost in the original space). This generation is similar to the revised simplex method, except that the variable space is exponentially large and therefore is generated implicitly by solving an optimization problem. This idea of generating variables as needed is the reason why this method is often referred to as column generation.

Similar to the two-phase simplex algorithm, the algorithm first introduces slack variables and solves a phase I problem to find a feasible solution. After the algorithm finds a feasible solution, it switches to a phase II problem to search for an optimal solution. The process of solving the master to generate pricing information and then solving one or more subproblems to generate candidate variables is repeated until there are no longer any improving variables and the method has converged.

For MILPs, this process is then used as a bounding method in a branch-and-bound algorithm, as described in the section “Branch-and-Bound Algorithm” on page 179. The strength of the subproblem polyhedron is one of the key reasons why decomposition can often solve problems that the standard branch-and-cut algorithm cannot solve in a reasonable amount of time. Since the points used in the convex combination are solutions (extreme points) of the subproblem (typically a MILP itself), then the bounds obtained can often be much stronger than the bounds obtained from standard branch-and-bound methods that are based on the LP
relaxation. The subproblem polyhedron intersected with the continuous master polyhedron can provide a very good approximation of the true convex hull of the original integer program.

For more information about the algorithm process flow and the framework design, see Galati (2009).

---

**Parallel Processing**

At each iteration of the decomposition method, the subproblem is solved to minimize the reduced cost that is derived from the dual information that solving the master problem provides. As discussed in the section “Overview: Decomposition Algorithm” on page 850, the subproblem often has a block-angular structure that enables the solver to process each block independently.

When solving MILPs, you can run the decomposition algorithm in either single-machine or distributed mode. If the `DISTRIBUTED` option in the main solver statement is set to TRUE (and your configured CAS session has two or more workers), then the computation is executed in distributed mode (using multiple machines simultaneously). The decomposition algorithm supports only the deterministic mode (`DETERMINISTIC=TRUE` in the main solver statement). The default mode of operation is single-machine mode.

You can control the number of threads that are used (by each machine) by specifying the `NTHREADS` option in the main solver statement. In distributed mode, if the allowable number of active CPUs differs among machines, the decomposition algorithm will use the minimum number allowed as the limit applied to the `NTHREADS` option in the main solver statement. In addition, you can use the `NTHREADS` option in each subcomponent statement to specify the number of threads to use for that solver. This is discussed in detail in the following sections:

- `NTHREADS` option in the DECOMP statement on page 863
- `NTHREADS` option in the DECOMPMASTER statement on page 865
- `NTHREADS` option in the DECOMPMASTERIP statement on page 866
- `NTHREADS` option in the DECOMPSUBPROB statement on page 870

Because of communication costs, increasing the number of machines does not guarantee faster execution, especially when you have subproblems that are unbalanced with respect to computational cost. For all the documentation examples, unless otherwise noted, the CAS session is configured for four worker nodes, each having 32 cores. For general information about CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*.

---

**Special Case: Identical Blocks and Ryan-Foster Branching**

In the special case of a set partitioning master problem and identical blocks, the underlying algorithm is automatically adjusted to reduce symmetry and improve overall performance. Identical blocks are subproblems (see the section “Overview: Decomposition Algorithm” on page 850) that have equivalent
feasible regions (and optima) when they are projected. Algebraically, this means that

\[
\begin{align*}
A^1 &= A^2 = \cdots = A^x \\
D^1 &= D^2 = \cdots = D^x \\
c^1 &= c^2 = \cdots = c^x \\
b^1 &= b^2 = \cdots = b^x \\
\bar{x}^1 &= \bar{x}^2 = \cdots = \bar{x}^x \\
\end{align*}
\]

A set partitioning problem is a specific type of integer programming model in which each constraint represents choosing exactly one member of a set. These constraints are often referred to as assignment constraints. The linear relaxation of a set partitioning problem enables an algorithm to choose fractional parts of several members of some set such that they sum to 1. Algebraically, this means \(Ax = 1\), where all the coefficients in \(A\) are 0 or 1.

The performance of algorithms that use a branch-and-bound method can suffer when the formulation contains substructures that are symmetric. In this context, symmetric means that an assignment of solutions can be arbitrarily permuted for some component without affecting the optimality of that solution. For example, if

\[
x_{11} = 1 \quad x_{12} = 0 \quad x_{21} = 0 \quad x_{22} = 1
\]

and

\[
x_{11} = 0 \quad x_{12} = 1 \quad x_{21} = 1 \quad x_{22} = 0
\]

are both optimal, then these solutions, \(x_{ij}\), are considered symmetric on index \(j\). That is, you can interchange \(j = 1\) and \(j = 2\) without affecting the optimality of the solution. The presence of identical blocks in a mathematical program is an obvious case in which symmetry can hurt performance. In order to overcome this handicap, the decomposition algorithm aggregates the identical blocks into one block when it forms the Dantzig-Wolfe master problem. If the Dantzig-Wolfe master problem is a set partitioning model, the algorithm uses a specialized branching rule known as Ryan-Foster branching. If the original master model (after aggregation) is equivalent to the identity matrix, this guarantees that the Dantzig-Wolfe master problem is of the appropriate form. For more information about the aggregate formulation and Ryan-Foster branching, see Barnhart et al. (1998).

Suppose you want to solve the following problem:

\[
\begin{align*}
\text{maximize} & \quad x_{11} + 2x_{21} + x_{31} + x_{12} + 2x_{22} + x_{32} \\
\text{subject to} & \quad x_{11} + x_{12} = 1 \quad (m1) \\
& \quad 5x_{11} + 7x_{21} + 4x_{31} + x_{22} = 1 \quad (m2) \\
& \quad 5x_{11} + 7x_{22} + 4x_{32} \leq 11 \quad (s1) \\
& \quad 5x_{12} + 7x_{22} + 4x_{32} \leq 11 \quad (s2) \\
& \quad x_{ij} \in \{0, 1\} \quad i \in \{1, 2, 3\}, \quad j \in \{1, 2\}
\end{align*}
\]

If constraints \(m1\) and \(m2\) are removed, then the remaining constraints \(s1\) and \(s2\) decompose into two independent and identical subproblems. In addition, constraints \(m1\) and \(m2\) form a set partitioning master problem.

The following statements use the OPTMODEL procedure and the decomposition algorithm to solve the preceding problem:
proc optmodel sessref="mysess";
  var x{i in 1..3, j in 1..2} binary;

  max f = x[1,1] + 2*x[2,1] + x[3,1] + x[1,2] + 2*x[2,2] + x[3,2];

  con m1: x[1,1] + x[1,2] = 1;
  con m2: x[2,1] + x[2,2] = 1;
  con s1: 5*x[1,1] + 7*x[2,1] + 4*x[3,1] <= 11;
  con s2: 5*x[1,2] + 7*x[2,2] + 4*x[3,2] <= 11;

  s1.block = 0;
  s2.block = 1;

  solve with milp / presolver=none decomp=(logfreq=1);
  print x;
  quit;

Here, the PRESOLVER=NONE option is used again, because otherwise the presolver solves this small instance without invoking any solver. The solution summary and optimal solution are displayed in Figure 18.3.

**Figure 18.3** Solution Summary and Optimal Solution

### The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
</tbody>
</table>

| Relative Gap | 0 |
| Absolute Gap | 0 |
| Primal Infeasibility | 0 |
| Bound Infeasibility | 0 |
| Integer Infeasibility | 0 |
| Best Bound | 5 |
| Nodes | 1 |
| Solutions Found | 2 |
| Iterations | 2 |
| Presolve Time | 0.03 |
| Solution Time | 0.08 |

<table>
<thead>
<tr>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
</tr>
<tr>
<td>1 1 0</td>
</tr>
<tr>
<td>2 0 1</td>
</tr>
<tr>
<td>3 1 1</td>
</tr>
</tbody>
</table>
Chapter 18: The Decomposition Algorithm

The iteration log, which displays the problem statistics, the progress of the solution, and the optimal objective value, is shown in Figure 18.4.

Figure 18.4 Log

<table>
<thead>
<tr>
<th>Iter</th>
<th>Master Bound</th>
<th>Objective</th>
<th>Best Integer</th>
<th>LP Gap</th>
<th>IP Gap</th>
<th>Time CPU Real Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>6.0000</td>
<td>5.0000</td>
<td>5.0000</td>
<td>16.67%</td>
<td>16.67%</td>
<td>0 0</td>
</tr>
<tr>
<td>1</td>
<td>6.0000</td>
<td>5.0000</td>
<td>5.0000</td>
<td>16.67%</td>
<td>16.67%</td>
<td>0 0</td>
</tr>
<tr>
<td>2</td>
<td>5.0000</td>
<td>5.0000</td>
<td>5.0000</td>
<td>0.00%</td>
<td>0.00%</td>
<td>0 0</td>
</tr>
</tbody>
</table>

The decomposition solver recognizes that the original master model is of the appropriate form and that each block is identical. It formulates the aggregate master and uses Ryan-Foster branching to solve the model.

In the presence of identical blocks, under certain circumstances, the aggregate formulation can also be used with a set covering master formulation. A set covering problem is an integer programming model in which each constraint represents choosing at least one member of a set. Algebraically, this means $Ax \geq 1$, where all the coefficients in $A$ are 0 or 1. Aggregate formulation and Ryan-Foster branching can be used if there exists an optimal solution, $x^*$, that is binding at equality ($Ax^* = 1$). If you can guarantee such a condition, you can greatly improve performance by explicitly using VARSEL=RYANFOSTER as a MILP main solver option. The decomposition algorithm usually performs better when it uses a set covering formulation as opposed to a
set partitioning formulation, because it is usually easier to find integer feasible solutions. If the models are equivalent, using the set covering formulation is recommended. For two examples, see Example 18.6, which shows the bin packing problem, and Example 18.8, which shows the vehicle routing problem.

Similarly, a set packing problem is an integer programming model in which each constraint represents choosing at most one member of a set. Algebraically, this means $Ax \leq 1$, where all the coefficients in $A$ are 0 or 1. Aggregate formulation and Ryan-Foster branching can be used if there exists an optimal solution, $x^*$, that is binding at equality ($Ax^* = 1$). In this case, using VARSEL=RYANFOSTER can improve performance. Alternatively, you can transform any set packing model into a set partitioning model by introducing a zero-cost slack variable for each packing constraint. See Example 18.11, which shows an application that optimizes a kidney donor exchange.

The decomposition algorithm automatically searches for identical blocks and the appropriate set partitioning master formulation. If it finds this structure, the algorithm automatically generates the aggregate formulation and uses Ryan-Foster branching. The aggregate model needs to process only one block at each subproblem iteration. Therefore, parallel processing (in which multiple blocks are processed simultaneously), as described in the section “Parallel Processing” on page 873, cannot improve performance. For this reason, when the decomposition algorithm runs in distributed mode, it does not create the aggregate formulation, nor does it use Ryan-Foster branching, even if the blocks are found to be identical.

---

**Log for the Decomposition Algorithm**

The following subsections describe what to expect in the SAS log when you run the decomposition algorithm.

**Setup Information in the SAS Log**

In the setup phase of the algorithm, information about the method you choose and the structure of the model is written to the SAS log. One of the most important pieces of information displayed in the log is the number of disjoint blocks and the coverage of those blocks with respect to both variables and constraints in the original presolved model. As explained in the section “Overview: Decomposition Algorithm” on page 850, the decomposition algorithm usually performs better than standard approaches only if the subproblems cover a significant amount of the original problem. However, this is not always a straightforward indicator for MILPs, because the strength of the subproblem with respect to integrality is not always proportional to the size of the system.

After the structural information is written to the log, the algorithm begins and the iteration log is displayed.

**Iteration Log for LPs**

When the decomposition algorithm solves LPs, the iteration log shows the progress of convergence in finding the appropriate set of columns in the reformulated space.

The following information is written to the iteration log:

- **Iter**: indicates the iteration number.
- **Best Bound**: indicates the best dual bound found so far.
- **Master Objective**: indicates the current amount of infeasibility in phase I and the primal objective value of the current solution in phase II.
Chapter 18: The Decomposition Algorithm

Gap indicates the relative difference between the master objective and the best known dual bound. This indicates how close the algorithm is to convergence. If the relative gap is greater than 1000%, then the absolute gap is written.

CPU Time indicates the CPU time elapsed (in seconds).

Real Time indicates the real time elapsed (in seconds).

Entries are made in the log at a frequency that is specified in the LOGFREQ= option. If LOGFREQ=0, then the iteration log is disabled. If the LOGFREQ= value is positive, then an entry is made in the log at the first iteration, at the last iteration, and at intervals that are specified by the LOGFREQ= value. An entry is also made each time an improved bound is found.

The behavior of objective values in the iteration log depends on both the current phase and on which solver you choose. In phase I, the master formulation has an artificial objective value that decreases to 0 when a feasible solution is found. In phase II, the decomposition algorithm maintains a primal feasible solution, so a minimization problem has decreasing objective values in the iteration log.

When you specify LOGLEVEL=MODERATE or LOGLEVEL=AGGRESSIVE in the DECOMP statement, information about the subproblem solves is written before each iteration line.

Iteration Log for MILPs

When the decomposition algorithm solves MILPs, the iteration log shows the progress of convergence in finding the appropriate set of columns in the reformulated space, in addition to the global convergence of the branch-and-bound algorithm for finding an optimal integer solution.

You can control the amount of information at each node by using the LOGLEVEL= option in the DECOMP statement. By default, the continuous iteration log for the root node is written at the interval specified in the LOGFREQ= option in the DECOMP statement. Then the branch-and-bound node log is written at the interval specified in the LOGFREQ= main solver option.

When the algorithm solves MILPs, the continuous iteration log is similar to the iteration log described in the section “Iteration Log for LPs” on page 877 except that information about integer feasible solutions is also displayed. The following information is printed in the continuous iteration log when the algorithm solves MILPs:

Iter indicates the iteration number.

Best Bound indicates the best dual bound found so far.

Master Objective indicates the current amount of infeasibility in phase I and the primal objective value of the current solution in phase II.

Best Integer indicates the objective value of the best integer feasible solution found so far.

LP Gap indicates the relative difference between the master objective and the best known dual bound. This indicates how close the algorithm for this particular node is to convergence. If the relative gap is greater than 1000%, then the absolute gap is displayed.

IP Gap indicates the relative difference between the best integer and the best known dual bound. This indicates how close the branch-and-bound algorithm is to convergence. If the relative gap is greater than 1000%, then the absolute gap is displayed.

CPU Time indicates the CPU time elapsed (in seconds).
Real Time indicates the real time elapsed (in seconds).

After the root node is complete, the algorithm then moves into the branch-and-bound phase. By default, it displays the branch-and-bound node log and suppresses the continuous iteration log.

The following information is printed in the branch-and-bound node log when the algorithm solves MILPs:

- **Node** indicates the sequence number of the current node in the search tree.
- **Active** indicates the current number of active nodes in the branch-and-bound tree.
- **Sols** indicates the number of feasible solutions found so far.
- **Best Integer** indicates the objective value of the best integer feasible solution found so far.
- **Best Bound** indicates the best dual bound found so far.
- **Gap** indicates the relative difference between the best integer and the best known dual bound. This indicates how close the branch-and-bound algorithm is to convergence. If the relative gap is greater than 1000%, then the absolute gap is displayed.
- **CPU Time** indicates the CPU time elapsed (in seconds).
- **Real Time** indicates the real time elapsed (in seconds).

If the LOGLEVEL= option in the DECOMP statement is set to BASIC, MODERATE or AGGRESSIVE, then the continuous iteration log is displayed for each branch-and-bound node at the interval specified in the LOGFREQ= option in the DECOMP statement.

Additional information can be displayed to the log by specifying the LOGLEVEL= option in each of the algorithmic component statements (DECOMPMASTER, DECOMPMASTERIP, and DECOMPSUBPROB). By default, the individual component log levels are all disabled.

### Examples: Decomposition Algorithm

#### Example 18.1: Multicommodity Flow Problem and METHOD=NETWORK

This example demonstrates how to use the decomposition algorithm to find a minimum-cost multicommodity flow (MMCF) in a directed network. This type of problem was motivation for the development of the original Dantzig-Wolfe decomposition method (Dantzig and Wolfe 1960).

Let $G = (N, A)$ be a directed graph, and let $K$ be a set of commodities. For each link $(i, j) \in A$ and each commodity $k$, associate a cost per unit of flow, designated by $c_{ij}^k$. The demand (or supply) at each node $i \in N$ for commodity $k$ is designated as $b_i^k$, where $b_i^k \geq 0$ denotes a supply node and $b_i^k < 0$ denotes a demand node. Define decision variables $x_{ij}^k$ that denote the amount of commodity $k$ sent from node $i$ and node $j$. The amount of total flow, across all commodities, that can be sent across each link is bounded above by $u_{ij}$. 

The problem can be modeled as a linear programming problem as follows:

\[
\text{minimize} \quad \sum_{(i,j) \in A} \sum_{k \in K} c_{ij}^k x_{ij}^k
\]

subject to

\[
\sum_{k \in K} x_{ij}^k \leq u_{ij} \quad (i, j) \in A \quad \text{(Capacity)}
\]

\[
\sum_{k \in K} x_{ij}^k - \sum_{(j,i) \in A} x_{ji}^k = b_i^k \quad i \in N, k \in K \quad \text{(Balance)}
\]

\[
x_{ij}^k \geq 0 \quad (i, j) \in A, k \in K
\]

In this formulation, the Capacity constraints limit the total flow across all commodities on each arc. The Balance constraints ensure that the flow of commodities leaving each supply node and entering each demand node are balanced.

Consider the directed graph in Figure 18.5 which appears in Ahuja, Magnanti, and Orlin (1993).

**Figure 18.5** Example Network with Two Commodities

![Figure 18.5](image.png)

The goal in this example is to minimize the total cost of sending two commodities across the network while satisfying all supplies and demands and respecting arc capacities. If there were no arc capacities linking the two commodities, you could solve a separate minimum-cost network flow problem for each commodity one at a time.

The following data set `arc_comm_data` provides the cost \( c_{ij}^k \) of sending a unit of commodity \( k \) along arc \((i, j)\):

```plaintext
data arc_comm_data;
   input k i j cost;
datalines;
1 1 2 1
1 1 3 5
1 5 3 1
1 5 6 5
1 3 4 1
```

Example 18.1: Multicommodity Flow Problem and METHOD=NETWORK

```
1 4 2 5
1 4 6 1
2 1 2 1
2 1 3 5
2 5 3 1
2 5 6 5
2 3 4 1
2 4 2 5
2 4 6 1
;
```

Next, the data set arc_data provides the capacity $u_{ij}$ for each arc:

```
data arc_data;
  input i j capacity;
datalines;
  1 2 5
  1 3 30
  5 3 30
  5 6 30
  3 4 10
  4 2 30
  4 6 30
;
```

```
data supply_data;
  input k i supply;
datalines;
  1 1 10
  1 2 -10
  2 5 20
  2 6 -20
;
```

The following PROC OPTMODEL statements find the minimum-cost multicommodity flow:

```
proc optmodel sessref="mysess";
  set <num,num,num> ARC_COMM;
  num cost {ARC_COMM};
  read data arc_comm_data into ARC_COMM=[i j k] cost;

  set ARCS = setof {<i,j,k> in ARC_COMM} <i,j>;
  set COMMODITIES = setof {<i,j,k> in ARC_COMM} k;
  set NODES = union {<i,j> in ARCS} {i,j};

  num arcCapacity (ARCS);
  read data arc_data into [i j] arcCapacity=capacity;

  num supply (NODES, COMMODITIES) init 0;
  read data supply_data into [i k] supply;

  var Flow {<i,j,k> in ARC_COMM} >= 0;
  min TotalCost =
    sum {<i,j,k> in ARC_COMM} cost[i,j,k] * Flow[i,j,k];
  con Balance {i in NODES, k in COMMODITIES}:
    supply[i,k] + sum {j in ARCS} Flow[i,j,k] =
    sum {j in ARCS} Flow[j,i,k] + sum {j in ARCS} Flow[i,j,k];
```

Because each Balance constraint involves variables for only one commodity, a decomposition by commodity is a natural choice. In both the OPTLP and OPTMILP procedures, the block identifiers must be consecutive integers starting from 0. In PROC OPTMODEL, the block identifiers only need to be numeric. The following FOR loop populates the .block constraint suffix with block identifier k for commodity k:

```plaintext
for{i in NODES, k in COMMODITIES}
  Balance[i,k].block = k;
```

The .block constraint suffix for the linking Capacity constraints is left missing, so these constraints become part of the master problem.

The following SOLVE statement uses the DECOMP option to invoke the decomposition algorithm:

```plaintext
solve with LP / presolver=none decomp subprob=(algorithm=nspure);
  print Flow;
quit;
```

Here, the PRESOLVER=NONE option is used, because otherwise the presolver solves this small instance without invoking any solver. Because each subproblem is a pure network flow problem, you can use the ALGORITHM=NSPURE option in the SUBPROB= option to request that a network simplex algorithm for pure networks be used instead of the default algorithm, which for linear programming subproblems is primal simplex.

It turns out for this example that if you specify METHOD=NETWORK (instead of the default METHOD=USER) in the DECOMP option, the network extractor finds the same blocks, one per commodity. To invoke the METHOD=NETWORK option, simply change the SOLVE statement as follows:

```plaintext
solve with LP / presolver=none decomp=(method=network);
```

In this case, the default subproblem solver is NSPURE.

The optimal solution and solution summary are displayed in Output 18.1.1.
### Output 18.1.1 Solution Summary and Optimal Solution

**The OPTMODEL Procedure**

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>LP</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Decomposition</td>
</tr>
<tr>
<td>Objective Function</td>
<td>TotalCost</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Optimal</td>
</tr>
<tr>
<td>Objective Value</td>
<td>150</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Dual Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Iterations</td>
<td>3</td>
</tr>
<tr>
<td>Presolve Time</td>
<td>0.00</td>
</tr>
<tr>
<td>Solution Time</td>
<td>0.01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>2</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

The optimal solution is shown on the network in Figure 18.6.
The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 18.1.2.
Example 18.2: Generalized Assignment Problem

The generalized assignment problem (GAP) is that of finding a maximum profit assignment from \( n \) tasks to \( m \) machines such that each task is assigned to precisely one machine subject to capacity restrictions on the machines. With each possible assignment, associate a binary variable \( x_{ij} \), which, if set to 1, indicates that machine \( i \) is assigned to task \( j \). For ease of notation, define two index sets \( M = \{1, \ldots, m\} \) and \( J = \{1, \ldots, n\} \).
N = \{1, \ldots, n\}. A GAP can be formulated as a MILP as follows:

\[
\begin{align*}
\text{maximize} & \quad \sum_{i \in M} \sum_{j \in N} p_{ij} x_{ij} \\
\text{subject to} & \quad \sum_{i \in M} x_{ij} = 1 \quad j \in N \quad \text{(Assignment)} \\
& \quad \sum_{j \in N} w_{ij} x_{ij} \leq b_i \quad i \in M \quad \text{(Knapsack)} \\
& \quad x_{ij} \in \{0, 1\} \quad i \in M, j \in N
\end{align*}
\]

In this formulation, Assignment constraints ensure that each task is assigned to exactly one machine. Knapsack constraints ensure that for each machine, the capacity restrictions are met.

Consider the following example taken from Koch et al. (2011) with n = 24 tasks to be assigned to m = 8 machines. The data set profit_data provides the profit for assigning a particular task to a particular machine:

```r
%let NumTasks = 24;
%let NumMachines = 8;

data profit_data;
  input p1-p&NumTasks;
datalines;
20 18 23 23 23 17 19 16 24 24 17 23 19 22 23 25 23 18 19 24 20 17 23 23
16 16 15 23 15 15 25 22 17 20 19 16 17 17 20 17 17 18 16 18 15 25 22 17
17 23 21 20 24 22 25 17 22 20 16 22 21 23 24 15 22 25 18 19 19 17 22 23
24 21 23 17 21 19 19 17 18 24 15 15 17 18 15 24 19 21 23 24 17 20 16 21
18 21 22 23 22 15 18 15 21 22 15 23 21 25 25 23 20 16 25 17 15 15 18 16
19 24 18 17 21 18 24 25 18 23 21 15 24 23 18 23 23 16 20 20 19 25 21
;
```

The data set weight_data provides the amount of resources used by a particular task when assigned to a particular machine:

```r
data weight_data;
  input w1-w&NumTasks;
datalines;
8 18 22 5 11 11 22 11 17 22 11 20 13 13 7 22 15 22 24 8 8 24 18 8
24 14 11 15 24 8 10 15 19 25 6 13 10 25 19 24 13 12 5 18 10 24 8 5
22 22 21 22 13 16 21 5 25 13 12 9 24 6 22 24 11 21 11 14 12 10 20 6
13 8 19 12 19 18 10 21 5 9 11 9 22 8 12 13 9 25 19 24 22 6 19 14
25 16 13 5 11 8 7 8 25 20 24 20 11 6 10 10 6 22 10 10 13 21 5 19
19 19 5 11 22 24 18 11 6 13 24 24 22 6 22 5 14 6 16 11 6 8 18 10
24 10 9 10 6 15 7 13 20 8 7 9 24 9 21 9 11 19 10 5 23 20 5 21
6 9 9 5 12 10 16 15 19 18 20 18 16 21 11 12 22 16 21 25 7 14 16 10
;
```

Finally, the data set capacity_data provides the resource capacity for each machine:
Example 18.2: Generalized Assignment Problem

```plaintext
data capacity_data;
   input b @@;
datalines;
36 35 38 34 32 34 31 34 ;
```

The following PROC OPTMODEL statements read in the data and define the necessary sets and parameters:

```plaintext
proc optmodel sessref="mysess";
   /* declare index sets */
   set TASKS = 1..&NumTasks;
   set MACHINES = 1..&NumMachines;

   /* declare parameters */
   num profit {MACHINES, TASKS};
   num weight {MACHINES, TASKS};
   num capacity {MACHINES};

   /* read data sets to populate data */
   read data profit_data into [i=_n_] {j in TASKS} <profit[i,j]=col('p'||j)>;
   read data weight_data into [i=_n_] {j in TASKS} <weight[i,j]=col('w'||j)>;
   read data capacity_data into [_n_] capacity=b;
```

The following statements declare the optimization model:

```plaintext
   /* declare decision variables */
   var Assign {MACHINES, TASKS} binary;

   /* declare objective */
   max TotalProfit =
      sum {i in MACHINES, j in TASKS} profit[i,j] * Assign[i,j];

   /* declare constraints */
   con Assignment {j in TASKS}:
      sum {i in MACHINES} Assign[i,j] = 1;

   con Knapsack {i in MACHINES}:
      sum {j in TASKS} weight[i,j] * Assign[i,j] <= capacity[i];
```

The following statements use two different decompositions to solve the problem. The first decomposition defines each Assignment constraint as a block and uses the pure network simplex solver for the subproblem. The second decomposition defines each Knapsack constraint as a block and uses the MILP solver for the subproblem.

```plaintext
   /* each Assignment constraint defines a block */
   for{j in TASKS}
      Assignment[j].block = j;

   solve with milp / logfreq=1000
decomp
decompsubprob=(algorithm=nspure);

   /* each Knapsack constraint defines a block */
   for{j in TASKS}
```
Assignment[j].block = .;
for{i in MACHINES}
    Knapsack[i].block = i;
solve with milp / decomp;
quit;

The solution summaries are displayed in Output 18.2.1.

Output 18.2.1 Solution Summaries

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>MILP</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Decomposition</td>
</tr>
<tr>
<td>Objective Function</td>
<td>TotalProfit</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Optimal</td>
</tr>
<tr>
<td>Objective Value</td>
<td>563</td>
</tr>
</tbody>
</table>

Relative Gap       0
Absolute Gap       0
Primal Infeasibility 0
Bound Infeasibility 0
Integer Infeasibility 0

Best Bound 563
Nodes 80
Solutions Found 11
Iterations 570
Presolve Time 0.02
Solution Time 1.31

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>MILP</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Decomposition</td>
</tr>
<tr>
<td>Objective Function</td>
<td>TotalProfit</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Optimal</td>
</tr>
<tr>
<td>Objective Value</td>
<td>563</td>
</tr>
</tbody>
</table>

Relative Gap       0
Absolute Gap       0
Primal Infeasibility 4.440892E-16
Bound Infeasibility 4.440892E-16
Integer Infeasibility 4.440892E-16

Best Bound 563
Nodes 3
Solutions Found 14
Iterations 60
Presolve Time 0.06
Solution Time 0.38
The iteration log for both decompositions is shown in Output 18.2.2. This example is interesting because it shows the trade-off between the strength of the relaxation and the difficulty of its resolution. In the first decomposition, the subproblems are totally unimodular and can be solved trivially. Consequently, each iteration of the decomposition algorithm is very fast. However, the bound obtained is as weak as the bound found in direct methods (the LP bound). The weaker bound leads to the need to enumerate more nodes overall. Alternatively, in the second decomposition, the subproblem is the knapsack problem, which is solved using MILP. In this case, the bound is much tighter and the problem solves in very few nodes. The trade-off, of course, is that each iteration takes longer because solving the knapsack problem is not trivial. Another interesting aspect of this problem is that the subproblem coverage in the second decomposition is much smaller than that of the first decomposition. However, when dealing with MILP, it is not always the size of the coverage that determines the overall effectiveness of a particular choice of decomposition.
Chapter 18: The Decomposition Algorithm

Output 18.2.2 Log

NOTE: There were 8 observations read from the data set WORK.PROFIT_DATA.
NOTE: There were 8 observations read from the data set WORK.WEIGHT_DATA.
NOTE: There were 8 observations read from the data set WORK.CAPACITY_DATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 192 variables (0 free, 0 fixed).
NOTE: The problem has 192 binary and 0 integer variables.
NOTE: The problem has 32 linear constraints (8 LE, 24 EQ, 0 GE, 0 range).
NOTE: The problem has 384 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The SOLVE statement is executing in the distributed computing environment in single-machine mode.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 5 constraint coefficients.
NOTE: The presolved problem has 192 variables, 32 constraints, and 384 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The number of block threads has been reduced to 24 threads.
WARNING: The subproblem solver chosen is an LP solver but at least one block has integer variables.
NOTE: The problem has a decomposable structure with 24 blocks. The largest block covers 3.125% of the constraints in the problem.
NOTE: The decomposition subproblems cover 192 (100%) variables and 24 (75%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Iter</th>
<th>Best Bound</th>
<th>Best Objective</th>
<th>Master Integer</th>
<th>Best LP Gap</th>
<th>IP Gap</th>
<th>CPU Time</th>
<th>Real Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>568.6281</td>
<td>568.1556</td>
<td>563.0000</td>
<td>0.02%</td>
<td>0.99%</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The Decomposition algorithm stopped on the continuous RELOBJGAP= option.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>Best Integer</th>
<th>Best Bound</th>
<th>Gap</th>
<th>CPU Time</th>
<th>Real Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>79</td>
<td>0</td>
<td>10</td>
<td>563.0000</td>
<td>568.6281</td>
<td>0.99%</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

NOTE: The Decomposition algorithm used 32 threads.
NOTE: The Decomposition algorithm time is 1.28 seconds.
NOTE: Optimal.
NOTE: Objective = 563.
NOTE: The SOLVE statement is executing in the distributed computing environment in single-machine mode.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
### Output 18.2.2 continued

NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 5 constraint coefficients.
NOTE: The presolved problem has 192 variables, 32 constraints, and 384 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The number of block threads has been reduced to 8 threads.
NOTE: The problem has a decomposable structure with 8 blocks. The largest block covers 3.125% of the constraints in the problem.
NOTE: The decomposition subproblems cover 192 (100%) variables and 8 (25%) constraints.
NOTE: The deterministic parallel mode is enabled.

<table>
<thead>
<tr>
<th>Iter</th>
<th>Best Bound</th>
<th>Objective</th>
<th>Master Integer</th>
<th>Best Integer</th>
<th>LP Gap</th>
<th>IP Gap</th>
<th>CPU Time</th>
<th>Real Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>568.6281</td>
<td>534.0000</td>
<td>534.0000</td>
<td>560.0000</td>
<td>6.09%</td>
<td>6.09%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>568.6281</td>
<td>560.0000</td>
<td>560.0000</td>
<td>560.0000</td>
<td>1.52%</td>
<td>1.52%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>568.6281</td>
<td>563.0000</td>
<td>563.0000</td>
<td>563.0000</td>
<td>0.99%</td>
<td>0.99%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>566.5000</td>
<td>563.0000</td>
<td>563.0000</td>
<td>563.0000</td>
<td>0.62%</td>
<td>0.62%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>21</td>
<td>566.1111</td>
<td>563.3333</td>
<td>563.3333</td>
<td>563.3333</td>
<td>0.49%</td>
<td>0.55%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>22</td>
<td>565.0000</td>
<td>564.0000</td>
<td>563.0000</td>
<td>563.0000</td>
<td>0.18%</td>
<td>0.35%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>564.0000</td>
<td>564.0000</td>
<td>563.0000</td>
<td>563.0000</td>
<td>0.00%</td>
<td>0.18%</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: Starting branch and bound.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>Best Integer</th>
<th>Best Bound</th>
<th>Gap</th>
<th>CPU Time</th>
<th>Real Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>13</td>
<td>563.0000</td>
<td>564.0000</td>
<td>0.18%</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>13</td>
<td>563.0000</td>
<td>563.0000</td>
<td>0.00%</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The Decomposition algorithm used 32 threads.

NOTE: The Decomposition algorithm time is 0.32 seconds.

NOTE: Optimal.

NOTE: Objective = 563.
Example 18.3: Block-Diagonal Structure and METHOD=CONCOMP

This example demonstrates how you can use the METHOD=CONCOMP option in the DECOMP statement to run the decomposition algorithm.

Consider a mixed integer linear program that is defined by the MPS data table mycas.mpsdata. In this case, the structure of the model is unknown and only the MPS data table is provided to you.

The following PROC OPTMILP statements solve the problem by using standard methods. The DATA= option assumes that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```plaintext
proc optmilp
  nthreads = 4
  data     = mycas.mpsdata;
run;
```

The solution summary is shown in Output 18.3.1.

**Output 18.3.1 Solution Summary**

<table>
<thead>
<tr>
<th>The OPTMILP Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution Summary</td>
</tr>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Relative Gap</td>
</tr>
<tr>
<td>Absolute Gap</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
</tr>
<tr>
<td>Integer Infeasibility</td>
</tr>
<tr>
<td>Best Bound</td>
</tr>
<tr>
<td>Nodes</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

The iteration log, which contains the problem statistics and the progress of the solution, is shown in Output 18.3.2.
A note in the log suggests that you can use the decomposition algorithm because of the structure of the problem. The following PROC OPTMILP statements use the METHOD=CONCOMP option in the DECOMP statement. The DATA= option assumes that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```plaintext
proc optmilp
  nthreads = 4
  data      = mycas.mpsdata;
  decomp
    loglevel = moderate
    method  = concomp;
run;
```

The solution summary is displayed in Output 18.3.3.
Output 18.3.3 Solution Summary

The OPTMILP Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
</tbody>
</table>

Relative Gap 0
Absolute Gap 0
Primal Infeasibility 1.376677E-13
Bound Infeasibility 6.883383E-14
Integer Infeasibility 2.220446E-16

Best Bound 120
Nodes 1
Iterations 1

Presolve Time 0.01
Solution Time 0.72

The iteration log, which contains the problem statistics and the progress of the solution, is shown in Output 18.3.4. When you specify NTHREADS=4 in the PROC OPTMILP statement, the blocks are processed simultaneously on four threads.
Example 18.3: Block-Diagonal Structure and METHOD=CONCOMP

Output 18.3.4 Log

NOTE: The problem MPSDATA has 388 variables (36 binary, 0 integer, 1 free, 0 fixed).
NOTE: The problem has 1297 constraints (630 LE, 37 EQ, 630 GE, 0 range).
NOTE: The problem has 4204 constraint coefficients.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 37 variables and 37 constraints.
NOTE: The MILP presolver removed 424 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 351 variables, 1260 constraints, and 3780 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in the distributed computing environment in single-machine mode.
NOTE: The DECOMP method value CONCOMP is applied.
NOTE: The problem has a decomposable structure with 4 blocks. The largest block covers 25.08% of the constraints in the problem.
NOTE: The decomposition subproblems cover 351 (100%) variables and 1260 (100%) constraints.
NOTE: Block 1 has 88 (25.07%) variables and 316 (25.08%) constraints.
NOTE: Block 2 has 88 (25.07%) variables and 316 (25.08%) constraints.
NOTE: Block 3 has 88 (25.07%) variables and 316 (25.08%) constraints.
NOTE: Block 4 has 87 (24.79%) variables and 312 (24.76%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.
NOTE: Starting to process node 0.
NOTE: Using a starting solution with objective value 161 to provide initial columns.
NOTE: Using a starting solution with objective value 231 to provide initial columns.
NOTE: Using a starting solution with objective value 231 to provide initial columns.
NOTE: The initial column pool using the starting solution contains 8 columns.
NOTE: The subproblem solver for 4 blocks at iteration 0 is starting.
NOTE: The subproblem solver for 4 blocks used 0.68 (cpu: 1.92, max: 0.68) seconds.
NOTE: The initial column pool after generating initial variables contains 12 columns.

<table>
<thead>
<tr>
<th>Iter</th>
<th>Bound</th>
<th>Master</th>
<th>Best</th>
<th>LP</th>
<th>IP</th>
<th>CPU Real</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Objective</td>
<td>Integer</td>
<td>Gap</td>
<td>Gap</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Time</td>
<td>Time</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: The master solver at iteration 1 is starting.
NOTE: The master solver used 0.00 (cpu: 0.00) seconds and 0 iterations.

| 1 | 120.0000 | 120.0000 | 120.0000 | 0.00% | 0.00% | 1 | 0 |

NOTE: The number of active nodes is 0.
NOTE: The objective value of the best integer feasible solution is 120.0000 and the best bound is 120.0000.
NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 0.71 seconds.
NOTE: Optimal.
NOTE: Objective = 120.
NOTE: The Cloud Analytic Services server processed the request in 0.771055 seconds.
In this case, the solver finds that after the presolve, the constraint matrix decomposes into block-diagonal form. That is, all the constraints are covered by subproblem blocks, leaving the set of master constraints empty. Because there are no coupling constraints, the problem decomposes into four completely independent problems. If you specify LOGLEVEL=MODERATE in the DECOMP statement, the log displays the size of each block. The blocks in this case are nicely balanced, allowing parallel execution to be efficient.

Example 18.4: Block-Angular Structure and METHOD=AUTO in Single-Machine Mode

This example demonstrates how you can use the METHOD=AUTO option in the DECOMP statement to run the decomposition algorithm in single-machine mode.

As in Example 18.3, consider a mixed integer linear program that is defined by the MPS data table mycas.mpsdata. In this case, the structure of the model is unknown and only the MPS data table is provided to you.

The following PROC OPTMILP statements attempt to solve the problem by using standard methods and a 60-second time limit. The DATA= option assumes that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```plaintext
proc optmilp
   nthreads = 4
   maxtime = 60
   data    = mycas.mpsdata;
run;
```

The solution summary is shown in Output 18.4.1.

Output 18.4.1 Solution Summary

<table>
<thead>
<tr>
<th>The OPTMILP Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution Summary</td>
</tr>
<tr>
<td>------------------------</td>
</tr>
<tr>
<td>Solver:</td>
</tr>
<tr>
<td>MILP</td>
</tr>
<tr>
<td>Algorithm:</td>
</tr>
<tr>
<td>Branch and Cut</td>
</tr>
<tr>
<td>Objective Function:</td>
</tr>
<tr>
<td>Total_Profit</td>
</tr>
<tr>
<td>Solution Status:</td>
</tr>
<tr>
<td>Time Limit Reached</td>
</tr>
<tr>
<td>Objective Value:</td>
</tr>
<tr>
<td>6151.1464478</td>
</tr>
<tr>
<td>Relative Gap:</td>
</tr>
<tr>
<td>0.1509088646</td>
</tr>
<tr>
<td>Absolute Gap:</td>
</tr>
<tr>
<td>1093.2425124</td>
</tr>
<tr>
<td>Primal Infeasibility:</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>Bound Infeasibility:</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>Integer Infeasibility:</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>Best Bound:</td>
</tr>
<tr>
<td>7244.3889602</td>
</tr>
<tr>
<td>Nodes:</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>Iterations:</td>
</tr>
<tr>
<td>49628</td>
</tr>
<tr>
<td>Presolve Time:</td>
</tr>
<tr>
<td>1.04</td>
</tr>
<tr>
<td>Solution Time:</td>
</tr>
<tr>
<td>59.74</td>
</tr>
</tbody>
</table>
The iteration log, which contains the problem statistics and the progress of the solution, is shown in Output 18.4.2.

### Output 18.4.2 Log

NOTE: The problem MPSDATA has 52638 variables (16038 binary, 0 integer, 0 free, 0 fixed).
NOTE: The problem has 3949 constraints (3339 LE, 0 EQ, 610 GE, 0 range).
NOTE: The default decomposition method (METHOD=AUTO) attempts to find a block-angular structure by using the matrix-stretching techniques that are described in Grcar (1990) and Aykanat, Pinar, and Çatalyürek (2004).

```plaintext
proc optmilp
    nthreads = 4
    data = mycas.mpsdata;
    decomp
        hybrid = false
        method = auto;
run;
```

The solution summary is displayed in Output 18.4.3.
Output 18.4.3  Solution Summary

The OPTMILP Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>MILP</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Decomposition</td>
</tr>
<tr>
<td>Objective Function</td>
<td>Total_Profit</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Optimal within Relative Gap</td>
</tr>
<tr>
<td>Objective Value</td>
<td>6972.3309351</td>
</tr>
<tr>
<td>Relative Gap</td>
<td>3.5550173E-9</td>
</tr>
<tr>
<td>Absolute Gap</td>
<td>0.0000247868</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
<td>4.596323E-14</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
<td>7.4113868E-8</td>
</tr>
<tr>
<td>Integer Infeasibility</td>
<td>9.325873E-15</td>
</tr>
<tr>
<td>Best Bound</td>
<td>6972.3309599</td>
</tr>
<tr>
<td>Nodes</td>
<td>1</td>
</tr>
<tr>
<td>Iterations</td>
<td>7</td>
</tr>
<tr>
<td>Presolve Time</td>
<td>1.03</td>
</tr>
<tr>
<td>Solution Time</td>
<td>36.57</td>
</tr>
</tbody>
</table>

The iteration log, which contains the problem statistics and the progress of the solution, is shown in Output 18.4.4.
As stated in the log, the automated method attempts to find a balanced block-angular form that contains four blocks (the default is the same as the value of the NTHREADS= option). The algorithm successfully finds such a decomposition and then further decomposes each block into its weakly connected components, resulting in 610 blocks and more than 99% subproblem coverage.
**Example 18.5: Block-Angular Structure and METHOD=AUTO in Distributed Mode**

This example demonstrates how you can use the METHOD=AUTO option in the DECOMP statement to run the decomposition algorithm in distributed mode.

Consider again the mixed integer linear program that is defined by the MPS data table mycas.mpsdata. The following PROC OPTMILP statements run the decomposition algorithm in distributed mode. The DATA= option assumes that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```plaintext
proc optmilp
  distributed = true
  nthreads = 4
  data = mycas.mpsdata;
  decomp
    hybrid = false
    method = auto;
run;
```

The solution summary is displayed in **Output 18.5.1**.

**Output 18.5.1 Solution Summary**

The OPTMILP Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>MILP</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Decomposition</td>
</tr>
<tr>
<td>Objective Function</td>
<td>Total_Profit</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Optimal within Relative Gap</td>
</tr>
<tr>
<td>Objective Value</td>
<td>6972.3309354</td>
</tr>
</tbody>
</table>

| Relative Gap             | 3.5121033E-9 |
| Absolute Gap             | 0.0000244875 |
| Primal Infeasibility     | 4.596323E-14 |
| Bound Infeasibility      | 8.5714041E-8 |
| Integer Infeasibility    | 9.325873E-15 |

| Best Bound               | 6972.3309599 |
| Nodes                    | 1           |
| Iterations               | 12          |

| Presolve Time            | 1.13        |
| Solution Time            | 23.85       |

The iteration log, which contains the problem statistics and the progress of the solution, is shown in **Output 18.5.2**. When you specify DISTRIBUTED=TRUE and NTHREADS=4 in the OPTMILP statement, the blocks are processed using all the machines in your configured session (four machines in this case). Each machine uses four threads.
In this case, the automated method attempts to find a balanced block-angular form that contains 16 blocks. The default for the NBLOCKS= option in the DECOMP statement is the number of machines in your configured session times the value of the NTHREADS= option in the DECOMP statement. The algorithm successfully finds such a decomposition and then further decomposes each block into the same 610 weakly connected components as in Example 18.4.
Example 18.6: Bin Packing Problem

The bin packing problem (BPP) finds the minimum number of capacitated bins that are needed to store a set of products of varying size. Define a set \( P \) of products, their sizes \( s_p \), and a set \( B = \{1, \ldots, |P| \} \) of candidate bins, each having capacity \( C \). Let \( x_{pb} \) be a binary variable that, if set to 1, indicates that product \( p \) is assigned to bin \( b \). In addition, let \( y_b \) be a binary variable that, if set to 1, indicates that bin \( b \) is used.

A BPP can be formulated as a MILP as follows:

\[
\begin{align*}
\text{minimize} & \quad \sum_{b \in B} y_b \\
\text{subject to} & \quad \sum_{b \in B} x_{pb} = 1, \quad p \in P \quad \text{(Assignment)} \\
& \quad \sum_{p \in P} s_p x_{pb} \leq C y_b, \quad b \in B \quad \text{(Capacity)} \\
& \quad x_{pb} \in \{0, 1\}, \quad p \in P, b \in B \\
& \quad y_b \in \{0, 1\}, \quad b \in B
\end{align*}
\]

In this formulation, the Assignment constraints ensure that each product is assigned to exactly one bin. The Capacity constraints ensure that the capacity restrictions are met for each bin. In addition, these constraints enforce the condition that if any product is assigned to bin \( b \), then \( y_b \) must be positive.

In this formulation, the bin identifier is arbitrary. For example, in any solution, the assignments to bin 1 can be swapped with the assignments to bin 2 without affecting feasibility or the objective value. Consider a decomposition by bin, where the Assignment constraints form the master problem and the Capacity constraints form identical subproblems. As described in the section “Special Case: Identical Blocks and Ryan-Foster Branching” on page 873, this is a situation in which an aggregate formulation and Ryan-Foster branching can greatly improve performance by reducing symmetry.

Consider a series of University of North Carolina basketball games that are recorded on a DVR. The following data set, \( \text{dvr} \), provides the name of each game in the column \( \text{opponent} \) and the size of that game in gigabytes (GB) as it resides on the DVR in the column size:

```/* game, size (in GBs) */
data dvr;
   input opponent $ size;
datalines;
Clemson 1.36
Clemson2 1.97
Duke 2.76
Duke2 2.52
FSU 2.56
FSU2 2.34
GT 1.49
GT2 1.12
IN 1.45
KY 1.42
Loyola 1.42
MD 1.33
MD2 2.71```

Example 18.6: Bin Packing Problem

The goal is to use the fewest DVDs on which to store the games for safekeeping. Each DVD can hold 4.38GB recorded data. The problem can be formulated as a bin packing problem and solved by using PROC OPTMODEL and the decomposition algorithm. The following PROC OPTMODEL statements read in the data, declare the optimization model, and use the decomposition algorithm to solve it:

```plaintext
proc optmodel sessref="mysess";
    /* read the product and size data */
    set <str> PRODUCTS;
    num size {PRODUCTS};
    read data dvr into PRODUCTS=[opponent] size;
    /* 4.38 GBs per DVD */
    num binsize = 4.38;
    /* the number of products is a trivial upper bound on the
     number of bins needed */
    num upperbound init card(PRODUCTS);
    set BINS = 1..upperbound;
    /* Assign[p,b] = 1, if product p is assigned to bin b */
    var Assign {PRODUCTS, BINS} binary;
    /* UseBin[b] = 1, if bin b is used */
    var UseBin {BINS} binary;
    /* minimize number of bins used */
    min Objective = sum {b in BINS} UseBin[b];
    /* assign each product to exactly one bin */
    con Assignment {p in PRODUCTS}:
        sum {b in BINS} Assign[p,b] = 1;
    /* Capacity constraint on each bin (and definition of UseBin) */
    con Capacity {b in BINS}:
        sum {p in PRODUCTS} size[p] * Assign[p,b] <= binsize * UseBin[b];
    /* decompose by bin (subproblem is a knapsack problem) */
    for {b in BINS} Capacity[b].block = b;
    /* solve using decomp (aggregate formulation) */
    solve with milp / decomp;
```

Miami 1.22
NCSU 2.52
NCSU2 2.54
UConn 1.25
VA 2.33
VA2 2.48
VT 1.41
Vermont 1.28
WM 1.25
WM2 1.23
Wake 1.61

;
Chapter 18: The Decomposition Algorithm

The following PROC OPTMODEL statements create a sequential numbering of the bins and then output to the data set dvd the optimal assignments of games to bins:

```plaintext
/* create a map from arbitrary bin number to sequential bin number */
num binId init 1;
num binMap {BINS};
for {b in BINS: UseBin[b].sol > 0.5} do;
    binMap[b] = binId;
    binId = binId + 1;
end;

/* create map of product to bin from solution */
num bin {PRODUCTS};
for {p in PRODUCTS} do;
    for {b in BINS: Assign[p,b].sol > 0.5} do;
        bin[p] = binMap[b];
        leave;
    end;
end;

/* create solution data */
create data dvd from [product] bin size;
quit;
```

The solution summary is displayed in Output 18.6.1.

**Output 18.6.1  Solution Summary**

<table>
<thead>
<tr>
<th>The OPTMODEL Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Solution Summary</strong></td>
</tr>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
</tbody>
</table>

| Relative Gap | 0  |
| Absolute Gap | 0  |
| Primal Infeasibility | 0  |
| Bound Infeasibility  | 0  |
| Integer Infeasibility | 0  |

| Best Bound | 11 |
| Nodes      | 1  |
| Solutions Found | 4  |
| Iterations | 2  |
| Presolve Time | 0.01 |
| Solution Time | 0.11 |

The iteration log is displayed in Output 18.6.2.
Output 18.6.2 Log

NOTE: There were 24 observations read from the data set WORK.DVR.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 600 variables (0 free, 0 fixed).
NOTE: The problem has 600 binary and 0 integer variables.
NOTE: The problem has 48 linear constraints (24 LE, 24 EQ, 0 GE, 0 range).
NOTE: The problem has 1176 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The SOLVE statement is executing in the distributed computing environment in single-machine mode.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 0 constraints.
NOTE: The MILP presolver removed 0 constraint coefficients.
NOTE: The MILP presolver modified 384 constraint coefficients.
NOTE: The presolved problem has 600 variables, 48 constraints, and 1176 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: All blocks are identical and the master model is set partitioning.
NOTE: The Decomposition algorithm is using an aggregate formulation and Ryan-Foster branching.
NOTE: The number of block threads has been reduced to 1 threads.
NOTE: The problem has a decomposable structure with 24 blocks. The largest block covers 2.083% of the constraints in the problem.
NOTE: The decomposition subproblems cover 600 (100%) variables and 24 (50%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Iter</th>
<th>Best Bound</th>
<th>Master Bound</th>
<th>Best Objective</th>
<th>LP Objective</th>
<th>IP Gap</th>
<th>CPU Time</th>
<th>Real Time</th>
</tr>
</thead>
</table>

NOTE: The continuous bound was improved to 11 due to objective granularity.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active Sols</th>
<th>Best Integer</th>
<th>Best Bound</th>
<th>Gap</th>
<th>CPU Time</th>
<th>Real Time</th>
</tr>
</thead>
</table>

NOTE: The Decomposition algorithm used 32 threads.
NOTE: The Decomposition algorithm time is 0.10 seconds.
NOTE: Optimal.
NOTE: Objective = 11.
NOTE: The data set WORK.DVD has 24 observations and 3 variables.
The following call to PROC SORT sorts the assignments by bin:

``` SAS
proc sort data=dvd;
    by bin;
run;
```

The optimal assignments from the output data set dvd are displayed in Figure 18.7.

**Figure 18.7 Optimal Assignment of Games to DVDs**

<table>
<thead>
<tr>
<th>bin=1</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
</tr>
<tr>
<td>size</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>Clemson2</td>
</tr>
<tr>
<td>GT2</td>
</tr>
<tr>
<td>WM</td>
</tr>
<tr>
<td><strong>bin</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bin=2</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
</tr>
<tr>
<td>size</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>Duke</td>
</tr>
<tr>
<td>GT</td>
</tr>
<tr>
<td><strong>bin</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bin=3</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
</tr>
<tr>
<td>size</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>Clemson</td>
</tr>
<tr>
<td>Duke2</td>
</tr>
<tr>
<td><strong>bin</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bin=4</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
</tr>
<tr>
<td>size</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>FSU</td>
</tr>
<tr>
<td>Loyola</td>
</tr>
<tr>
<td><strong>bin</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bin=5</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
</tr>
<tr>
<td>size</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>FSU2</td>
</tr>
<tr>
<td>KY</td>
</tr>
<tr>
<td><strong>bin</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>bin=6</th>
</tr>
</thead>
<tbody>
<tr>
<td>product</td>
</tr>
<tr>
<td>size</td>
</tr>
<tr>
<td>------------</td>
</tr>
<tr>
<td>NCSU2</td>
</tr>
<tr>
<td>Vermont</td>
</tr>
<tr>
<td><strong>bin</strong></td>
</tr>
</tbody>
</table>
In this example, the objective function ensures that there exists an optimal solution that never assigns a product to more than one bin. Therefore, you could instead model the Assignment constraint as an inequality rather than an equality. In this case, the best performance would come from forcing the use of an aggregate formulation and Ryan-Foster branching by specifying the option VARSEL=RYANFOSTER. An example of doing this is shown in Example 18.8.

**Example 18.7: Resource Allocation Problem**

This example describes a model for selecting tasks to be run on a shared resource (Gamrath 2010). Consider a set $I$ of tasks and a resource capacity $C$. Each item $i \in I$ has a profit $p_i$, a resource utilization level $w_i$, a starting period $s_i$, and an ending period $e_i$. The time horizon that is considered is from the earliest starting time to the latest ending time of all tasks. With each task, associate a binary variable $x_i$, which, if set to 1, indicates that the task is running from its start time until just before its end time. A task consumes capacity if it is running. The goal is to select which tasks to run in order to maximize profit while not

<table>
<thead>
<tr>
<th>Bin</th>
<th>Product</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>MD</td>
<td>1.33</td>
</tr>
<tr>
<td></td>
<td>UConn</td>
<td>1.25</td>
</tr>
<tr>
<td></td>
<td>WM2</td>
<td>1.23</td>
</tr>
<tr>
<td></td>
<td>Bin 3.81</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>NCSU</td>
<td>2.52</td>
</tr>
<tr>
<td></td>
<td>VT</td>
<td>1.41</td>
</tr>
<tr>
<td></td>
<td>Bin 3.93</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>IN</td>
<td>1.45</td>
</tr>
<tr>
<td></td>
<td>VA</td>
<td>2.33</td>
</tr>
<tr>
<td></td>
<td>Bin 3.78</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>VA2</td>
<td>2.48</td>
</tr>
<tr>
<td></td>
<td>Wake</td>
<td>1.61</td>
</tr>
<tr>
<td></td>
<td>Bin 4.09</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>MD2</td>
<td>2.71</td>
</tr>
<tr>
<td></td>
<td>Miami</td>
<td>1.22</td>
</tr>
<tr>
<td></td>
<td>Bin 3.93</td>
<td></td>
</tr>
<tr>
<td></td>
<td>43.57</td>
<td></td>
</tr>
</tbody>
</table>
exceeding the shared resource capacity. Let \( S = \{ s_i \mid i \in I \} \) define the set of start times for all tasks, and let \( L_s = \{ i \in I \mid s_i \leq s < e_i \} \) define the set of tasks that are running at each start time \( s \in S \). You can model the problem as a mixed integer linear programming problem as follows:

\[
\begin{align*}
\text{maximize} & \quad \sum_{i \in I} p_i x_i \\
\text{subject to} & \quad \sum_{i \in L_s} w_i x_i \leq C & s \in S \\
& \quad x_i \in \{0, 1\} & i \in I
\end{align*}
\]

In this formulation, CapacityCon constraints ensure that the running tasks do not exceed the resource capacity. To illustrate, consider the following five-task example with data: \( p_i = (6, 8, 5, 9, 8) \), \( w_i = (8, 5, 3, 4, 3) \), \( s_i = (1, 3, 5, 7, 8) \), \( e_i = (5, 8, 9, 17, 10) \), and \( C = 10 \). The formulation leads to a constraint matrix that has a \textit{staircase structure} that is determined by tasks coming online and offline:

\[
\begin{align*}
\text{maximize} & \quad 6x_1 + 8x_2 + 5x_3 + 9x_4 + 8x_5 \\
\text{subject to} & \quad 8x_1 + 5x_2 \leq 10 \\
& \quad 5x_2 + 3x_3 \leq 10 \\
& \quad 5x_2 + 3x_3 + 4x_4 \leq 10 \\
& \quad 3x_3 + 4x_4 + 3x_5 \leq 10 \\
& \quad x_i \in \{0, 1\} & i \in I
\end{align*}
\]

**Lagrangian Decomposition**

This formulation clearly has no decomposable structure. However, you can use a common modeling technique known as \textit{Lagrangian decomposition} to bring the model into block-angular form. Lagrangian decomposition works by first partitioning the constraints into blocks. Then, each original variable is split into multiple copies of itself, one copy for each block in which the variable has a nonzero coefficient in the constraint matrix. Constraints are added to enforce the equality of each copy of the original variable. Then, you can write the original constraints in block-angular form by using the duplicate variables.

To apply Lagrangian decomposition to the resource allocation problem, define a set \( B \) of blocks and let \( S_b \) define the set of start times for a given block \( b \), such that \( S = \bigcup_{b \in B} S_b \). Given this partition of start times, let \( B_i \) define the set of blocks in which task \( i \in I \) is scheduled to be running. Now, for each task \( i \in I \), define duplicate variables \( x_i^b \) for each \( b \in B_i \). Let \( m_i \) define the minimum block index for each class of variable that represents task \( i \). You can now model the problem in block-angular form as follows:

\[
\begin{align*}
\text{maximize} & \quad \sum_{i \in I} p_i x_i^{m_i} \\
\text{subject to} & \quad x_i^b = x_i^{m_i} & i \in I, \ b \in B_i \setminus \{m_i\} \\
& \quad \sum_{i \in L_s} w_i x_i^b \leq C & b \in B, \ s \in S_b \\
& \quad x_i^b \in \{0, 1\} & i \in I, \ b \in B_i
\end{align*}
\]
In this formulation, the LinkDupVarsCon constraints ensure that the duplicate variables are equal to the original variables. Now, the five-task example has been transformed from a staircase structure to a block-angular structure:

\[
\begin{align*}
\text{maximize} & \quad 6x_1^1 + 8x_2^1 + 5x_3^2 + 9x_4^2 + 8x_5^3 \\
\text{subject to} & \quad x_2^1 - x_2^2 + x_3^2 - x_3^3 - x_4^3 = 0 \\
& \quad 8x_1^1 - x_4^2 = 0 \\
& \quad 8x_1^1 + 5x_2^1 \leq 10 \\
& \quad 5x_2^2 + 3x_3^2 \leq 10 \\
& \quad 5x_2^2 + 3x_3^2 + 4x_4^2 \leq 10 \\
& \quad 3x_3^3 + 4x_4^3 + 3x_5^3 \leq 10 \\
& \quad x_i^b \in \{0, 1\}, i \in I, b \in B_i
\end{align*}
\]

To see how to apply Lagrangian decomposition in PROC OPTMODEL, consider the data set TaskData from Caprara, Furini, and Malaguti (2010), which consists of $|I| = 2,916$ tasks:

```plaintext
data TaskData;
   input profit weight start end;
datalines;
99 92 1 9
56 30 1 3
39 73 1 20
86 76 1 9
...
24 94 768 769
95 40 768 769
66 17 768 769
18 48 768 769
97 23 768 769
;
```

**Using the MILP Solver Directly in PROC OPTMODEL**

The following PROC OPTMODEL statements read in the data and solve the original staircase formulation by calling the MILP solver directly:

```plaintext
%macro SetupData(task_data=, capacity=);
   set TASKS;
   num capacity=&capacity;
   num profit{TASKS}, weight{TASKS}, start{TASKS}, end{TASKS};
   read data &task_data into TASKS=[_n_] profit weight start end;
   /* the set of start times */
   set STARTS = setof{i in TASKS} start[i];
   /* the set of tasks i that are active at a given start time s */
   set TASKS_START{s in STARTS} = {i in TASKS: start[i] <= s < end[i]};
%mend SetupData;
```
Chapter 18: The Decomposition Algorithm

%macro ResourceAllocation_Direct(task_data=, capacity=);
   proc optmodel sessref="mysess";
      %SetupData(task_data=&task_data,capacity=&capacity);

      /* select task i to come online from period [start to end] */
      var x{TASKS} binary;

      /* maximize the total profit of running tasks */
      max TotalProfit = sum{i in TASKS} profit[i] * x[i];

      /* enforce that the shared resource capacity is not exceeded */
      con CapacityCon{s in STARTS}:
         sum{i in TASKS_START[s]} weight[i] * x[i] <= capacity;

      solve with milp / maxtime=200 logfreq=10000;
      quit;
   %mend ResourceAllocation_Direct;

%ResourceAllocation_Direct(task_data=TaskData, capacity=100);

The problem summary and solution summary are displayed in Output 18.7.1.

Output 18.7.1  Problem Summary and Solution Summary

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Variables</th>
<th>2916</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bounded Above</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below</td>
<td>0</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
<td>2916</td>
</tr>
<tr>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>Fixed</td>
<td>0</td>
</tr>
<tr>
<td>Binary</td>
<td>2916</td>
</tr>
<tr>
<td>Integer</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Constraints</th>
<th>768</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear LE (&lt;=)</td>
<td>768</td>
</tr>
<tr>
<td>Linear EQ (=)</td>
<td>0</td>
</tr>
<tr>
<td>Linear GE (&gt;=)</td>
<td>0</td>
</tr>
<tr>
<td>Linear Range</td>
<td>0</td>
</tr>
</tbody>
</table>

| Constraint Coefficients | 23236  |
### Output 18.7.1 continued

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>MILP</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Branch and Cut</td>
</tr>
<tr>
<td>Objective Function</td>
<td>TotalProfit</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Optimal within Relative Gap</td>
</tr>
<tr>
<td>Objective Value</td>
<td>40982</td>
</tr>
<tr>
<td>Relative Gap</td>
<td>0.0000998259</td>
</tr>
<tr>
<td>Absolute Gap</td>
<td>4.0914747699</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Integer Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Best Bound</td>
<td>40986.091475</td>
</tr>
<tr>
<td>Nodes</td>
<td>88638</td>
</tr>
<tr>
<td>Solutions Found</td>
<td>22</td>
</tr>
<tr>
<td>Iterations</td>
<td>2388556</td>
</tr>
<tr>
<td>Presolve Time</td>
<td>0.15</td>
</tr>
<tr>
<td>Solution Time</td>
<td>42.99</td>
</tr>
</tbody>
</table>

The iteration log, which contains the problem statistics, the progress of the solution, and the best integer feasible solution found, is shown in Output 18.7.2.
Output 18.7.2 Log

NOTE: There were 2916 observations read from the data set WORK.TASKDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 2916 variables (0 free, 0 fixed).
NOTE: The problem has 2916 binary and 0 integer variables.
NOTE: The problem has 768 linear constraints (768 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 23236 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The SOLVE statement is executing in the distributed computing environment in single-machine mode.
NOTE: The remaining solution time after problem generation and solver initialization is 199.32 seconds.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 1021 variables and 126 constraints.
NOTE: The MILP presolver removed 12544 constraint coefficients.
NOTE: The MILP presolver modified 987 constraint coefficients.
NOTE: The presolved problem has 1895 variables, 642 constraints, and 10692 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Solns</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>33939.0000000</td>
<td>109181</td>
<td>68.91%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>33939.0000000</td>
<td>45862.9249030</td>
<td>26.00%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
<td>39381.0000000</td>
<td>43471.6068311</td>
<td>9.41%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
<td>39381.0000000</td>
<td>42707.3723942</td>
<td>7.79%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
<td>39381.0000000</td>
<td>42241.1002514</td>
<td>6.77%</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
<td>39381.0000000</td>
<td>41955.1816775</td>
<td>6.14%</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
<td>39381.0000000</td>
<td>41734.1601960</td>
<td>5.64%</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
<td>39381.0000000</td>
<td>41607.5699470</td>
<td>5.35%</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
<td>39381.0000000</td>
<td>41505.9202811</td>
<td>5.12%</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
<td>39381.0000000</td>
<td>41425.0313262</td>
<td>4.93%</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>8</td>
<td>39381.0000000</td>
<td>41362.4491601</td>
<td>4.79%</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>40449.0000000</td>
<td>41305.7252510</td>
<td>2.07%</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>40449.0000000</td>
<td>41274.6068425</td>
<td>2.00%</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>40449.0000000</td>
<td>41234.5169431</td>
<td>1.90%</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>40449.0000000</td>
<td>41215.4383549</td>
<td>1.86%</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>40449.0000000</td>
<td>41190.2974715</td>
<td>1.80%</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>40449.0000000</td>
<td>41171.5824493</td>
<td>1.76%</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>40449.0000000</td>
<td>41160.1367199</td>
<td>1.73%</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>40449.0000000</td>
<td>41149.4384559</td>
<td>1.70%</td>
<td>4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>40449.0000000</td>
<td>41139.0749379</td>
<td>1.68%</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>40449.0000000</td>
<td>41127.2553311</td>
<td>1.65%</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>40449.0000000</td>
<td>41113.7119945</td>
<td>1.62%</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>40449.0000000</td>
<td>41103.9721930</td>
<td>1.59%</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>40449.0000000</td>
<td>41097.6440386</td>
<td>1.58%</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>11</td>
<td>40449.0000000</td>
<td>41092.5748715</td>
<td>1.57%</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>14</td>
<td>40874.0000000</td>
<td>41088.3184301</td>
<td>0.52%</td>
<td>6</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>14</td>
<td>40874.0000000</td>
<td>41084.3901110</td>
<td>0.51%</td>
<td>6</td>
</tr>
</tbody>
</table>
Example 18.7: Resource Allocation Problem

Using the Decomposition Algorithm in PROC OPTMODEL

To transform this data into block-angular form, first sort the task data to help reduce the number of duplicate variables that are needed in the reformulation as follows:

```sas
proc sort data=TaskData;
  by start end;
run;
```

Then, create the partition of constraints into blocks of size `block_size` as follows:

```sas
%macro ResourceAllocation_Decomp(task_data=, capacity=, block_size=);
  proc optmodel sessref="mysess";
    %SetupData(task_data=&task_data,capacity=&capacity);
    /* partition into blocks of size block_size */
    num block_size = &block_size;
    num num_blocks = ceil( card(TASKS) / block_size );
    set BLOCKS = 1..num_blocks;
    /* the set of starts s for which task i is active */
    set STARTS_TASK{i in TASKS} = {s in STARTS: start[i] <= s < end[i]};
    /* partition the start times into blocks of size block_size */
    set STARTS_BLOCK(BLOCKS) init {};
    num block_id init 1;
    num block_count init 0;
    for{s in STARTS} do;
      STARTS_BLOCK[block_id] = STARTS_BLOCK[block_id] union {s};
      block_count = block_count + 1;
      if(mod(block_count, block_size) = 0) then
        block_id = block_id + 1;
    end;
  %end;
```

Output 18.7.2 continued

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>14</td>
<td>40874.000000</td>
<td>41078.2336982</td>
<td>0.50%</td>
<td>6</td>
</tr>
</tbody>
</table>

NOTE: The MIP solver added 845 cuts with 11367 cut coefficients at the root.

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3213</td>
<td>1995</td>
<td>15</td>
<td>40904.000000</td>
<td>41041.0561324</td>
<td>0.33%</td>
<td>8</td>
</tr>
<tr>
<td>4990</td>
<td>3263</td>
<td>16</td>
<td>40954.000000</td>
<td>41036.6565997</td>
<td>0.20%</td>
<td>9</td>
</tr>
<tr>
<td>4991</td>
<td>3235</td>
<td>17</td>
<td>40963.000000</td>
<td>41036.6565997</td>
<td>0.18%</td>
<td>9</td>
</tr>
<tr>
<td>6175</td>
<td>2742</td>
<td>18</td>
<td>40971.000000</td>
<td>41033.2414590</td>
<td>0.15%</td>
<td>9</td>
</tr>
<tr>
<td>13956</td>
<td>7181</td>
<td>19</td>
<td>40973.000000</td>
<td>41021.3838979</td>
<td>0.12%</td>
<td>11</td>
</tr>
<tr>
<td>17561</td>
<td>8937</td>
<td>20</td>
<td>40975.000000</td>
<td>41018.8825706</td>
<td>0.11%</td>
<td>12</td>
</tr>
<tr>
<td>43746</td>
<td>17769</td>
<td>21</td>
<td>40982.000000</td>
<td>41007.3125222</td>
<td>0.06%</td>
<td>23</td>
</tr>
<tr>
<td>88637</td>
<td>1914</td>
<td>21</td>
<td>40982.000000</td>
<td>40986.0914748</td>
<td>0.01%</td>
<td>42</td>
</tr>
</tbody>
</table>

NOTE: Optimal within relative gap.

NOTE: Objective = 40982.
/* blocks in which task i is online */
set BLOCKS_TASK{i in TASKS} =
    {b in BLOCKS: card(STARTS_BLOCK[b] inter STARTS_TASK[i]) > 0};

/* minimum block id in which task i is online */
num min_block{i in TASKS} = min{b in BLOCKS_TASK[i]} b;

/* select task i to come online from period [start to end) in each block */
var x{i in TASKS, b in BLOCKS_TASK[i]} binary;

/* maximize the total profit of running tasks */
max TotalProfit = sum{i in TASKS} profit[i] * x[i,min_block[i]];

/* enforce that task selection is consistent across blocks */
con LinkDupVarsCon{i in TASKS, b in BLOCKS_TASK[i] diff {min_block[i]}}:
    x[i,b] = x[i,min_block[i]];

/* enforce that the shared resource capacity is not exceeded */
con CapacityCon{b in BLOCKS, s in STARTS_BLOCK[b]}:
    sum{i in TASKS_START[s]} weight[i] * x[i,b] <= capacity;

/* define blocks for decomposition algorithm */
for{b in BLOCKS, s in STARTS_BLOCK[b]} CapacityCon[b,s].block = b;

solve with milp / presolver=basic decomp;
quit;
%mend ResourceAllocation_Decomp;

%ResourceAllocation_Decomp(task_data=TaskData, capacity=100, block_size=20);

The problem summary and solution summary are displayed in Output 18.7.3. Compared to the original formulation, the numbers of variables and constraints are increased by the number of duplicate variables.
Output 18.7.3  Problem Summary and Solution Summary

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Sense</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Objective Type</td>
</tr>
<tr>
<td>Number of Variables</td>
</tr>
<tr>
<td>Bounded Above</td>
</tr>
<tr>
<td>Bounded Below</td>
</tr>
<tr>
<td>Bounded Below and Above</td>
</tr>
<tr>
<td>Free</td>
</tr>
<tr>
<td>Fixed</td>
</tr>
<tr>
<td>Binary</td>
</tr>
<tr>
<td>Integer</td>
</tr>
<tr>
<td>Number of Constraints</td>
</tr>
<tr>
<td>Linear LE (&lt;=)</td>
</tr>
<tr>
<td>Linear EQ (=)</td>
</tr>
<tr>
<td>Linear GE (&gt;=)</td>
</tr>
<tr>
<td>Linear Range</td>
</tr>
<tr>
<td>Constraint Coefficients</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Relative Gap</td>
</tr>
<tr>
<td>Absolute Gap</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
</tr>
<tr>
<td>Integer Infeasibility</td>
</tr>
<tr>
<td>Best Bound</td>
</tr>
<tr>
<td>Nodes</td>
</tr>
<tr>
<td>Solutions Found</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 18.7.4.
Chapter 18: The Decomposition Algorithm

Output 18.7.4 Log

NOTE: There were 2916 observations read from the data set WORK.TASKDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3924 variables (0 free, 0 fixed).
NOTE: The problem has 3924 binary and 0 integer variables.
NOTE: The problem has 1776 linear constraints (768 LE, 1008 EQ, 0 GE, 0 range).
NOTE: The problem has 25252 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The SOLVE statement is executing in the distributed computing environment in single-machine mode.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value BASIC is applied.
NOTE: The MILP presolver removed 4 variables and 0 constraints.
NOTE: The MILP presolver removed 23 constraint coefficients.
NOTE: The MILP presolver modified 7297 constraint coefficients.
NOTE: The presolved problem has 3920 variables, 1776 constraints, and 25229 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The problem has a decomposable structure with 39 blocks. The largest block covers 1.126% of the constraints in the problem.
NOTE: The decomposition subproblems cover 3920 (100%) variables and 768 (43.24%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Iter</th>
<th>Best Bound</th>
<th>Best Objective</th>
<th>Master Integer</th>
<th>Best LP Gap</th>
<th>IP Gap</th>
<th>Time</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>44109.0000</td>
<td>37707.0000</td>
<td>37707.0000</td>
<td>14.51%</td>
<td>14.51%</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>44109.0000</td>
<td>37740.0000</td>
<td>37740.0000</td>
<td>14.44%</td>
<td>14.44%</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>44109.0000</td>
<td>37844.0000</td>
<td>37844.0000</td>
<td>14.20%</td>
<td>14.20%</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>44109.0000</td>
<td>37844.0000</td>
<td>38081.0000</td>
<td>14.20%</td>
<td>13.67%</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>44109.0000</td>
<td>39128.0000</td>
<td>39128.0000</td>
<td>11.29%</td>
<td>11.29%</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>44109.0000</td>
<td>39128.0000</td>
<td>39128.0000</td>
<td>11.29%</td>
<td>11.29%</td>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>22</td>
<td>44109.0000</td>
<td>39645.0000</td>
<td>39645.0000</td>
<td>10.12%</td>
<td>10.12%</td>
<td>16</td>
<td>6</td>
</tr>
<tr>
<td>25</td>
<td>44109.0000</td>
<td>39911.0000</td>
<td>39911.0000</td>
<td>9.52%</td>
<td>9.52%</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>30</td>
<td>44109.0000</td>
<td>39981.0000</td>
<td>39981.0000</td>
<td>9.36%</td>
<td>9.36%</td>
<td>28</td>
<td>9</td>
</tr>
<tr>
<td>32</td>
<td>44109.0000</td>
<td>40036.0000</td>
<td>40036.0000</td>
<td>9.23%</td>
<td>9.23%</td>
<td>30</td>
<td>9</td>
</tr>
<tr>
<td>34</td>
<td>44109.0000</td>
<td>40279.0000</td>
<td>40279.0000</td>
<td>8.68%</td>
<td>8.68%</td>
<td>33</td>
<td>10</td>
</tr>
<tr>
<td>35</td>
<td>44109.0000</td>
<td>40290.0000</td>
<td>40290.0000</td>
<td>8.66%</td>
<td>8.66%</td>
<td>35</td>
<td>10</td>
</tr>
<tr>
<td>37</td>
<td>44109.0000</td>
<td>40410.0000</td>
<td>40410.0000</td>
<td>8.39%</td>
<td>8.39%</td>
<td>38</td>
<td>11</td>
</tr>
<tr>
<td>38</td>
<td>44109.0000</td>
<td>40476.0000</td>
<td>40476.0000</td>
<td>8.24%</td>
<td>8.24%</td>
<td>42</td>
<td>12</td>
</tr>
<tr>
<td>39</td>
<td>44109.0000</td>
<td>40480.0000</td>
<td>40480.0000</td>
<td>8.23%</td>
<td>8.23%</td>
<td>44</td>
<td>12</td>
</tr>
<tr>
<td>40</td>
<td>44109.0000</td>
<td>40480.0000</td>
<td>40480.0000</td>
<td>8.23%</td>
<td>8.23%</td>
<td>44</td>
<td>13</td>
</tr>
<tr>
<td>41</td>
<td>43229.5360</td>
<td>40480.0000</td>
<td>40480.0000</td>
<td>6.36%</td>
<td>6.36%</td>
<td>50</td>
<td>14</td>
</tr>
<tr>
<td>43</td>
<td>43229.5360</td>
<td>40596.0000</td>
<td>40596.0000</td>
<td>6.09%</td>
<td>6.09%</td>
<td>53</td>
<td>14</td>
</tr>
<tr>
<td>49</td>
<td>43229.5360</td>
<td>40726.0000</td>
<td>40726.0000</td>
<td>5.79%</td>
<td>5.79%</td>
<td>63</td>
<td>17</td>
</tr>
<tr>
<td>50</td>
<td>43229.5360</td>
<td>40726.0000</td>
<td>40726.0000</td>
<td>5.79%</td>
<td>5.79%</td>
<td>65</td>
<td>17</td>
</tr>
</tbody>
</table>
The Trade-Off between Coverage and Subproblem Difficulty

The reformulation of this resource allocation problem provides a nice example of the potential trade-offs in modeling a problem for use with the decomposition algorithm. As seen in Example 18.2, the strength of the bound is an important factor in the overall performance of the algorithm, but it is not always correlated to the magnitude of the subproblem coverage. In the current example, the block size determines the number of blocks. Moreover, it determines the number of linking variables that are needed in the reformulation. At one extreme, if the block size is set to be $|S|$, then the number of blocks is 1, and the number of copies of original variables is 0. Using one block would be equivalent to the original staircase formulation and would not yield a model conducive to decomposition. As the number of blocks is increased, the number of linking variables increases (the size of the master problem), the strength of the decomposition bound decreases, and the difficulty of solving the subproblems decreases. In addition, as the number of blocks and their relative difficulty change, the efficient utilization of your machine’s parallel architecture can be affected.

The previous section used a block size of 20. The following statement calls the decomposition algorithm and uses a block size of 80:
%ResourceAllocation-Decomp(task_data=TaskData, capacity=100, block_size=80);

The solution summary is displayed in Output 18.7.5.

**Output 18.7.5**  Solution Summary

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Relative Gap</td>
</tr>
<tr>
<td>Absolute Gap</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
</tr>
<tr>
<td>Integer Infeasibility</td>
</tr>
<tr>
<td>Best Bound</td>
</tr>
<tr>
<td>Nodes</td>
</tr>
<tr>
<td>Solutions Found</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 18.7.6.

This version of the model provides a stronger initial bound and solves to optimality in the root node.
Output 18.7.6 Log

NOTE: There were 2916 observations read from the data set WORK.TASKDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 3151 variables (0 free, 0 fixed).
NOTE: The problem has 3151 binary and 0 integer variables.
NOTE: The problem has 1003 linear constraints (768 LE, 235 EQ, 0 GE, 0 range).
NOTE: The problem has 23706 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The SOLVE statement is executing in the distributed computing environment in single-machine mode.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value BASIC is applied.
NOTE: The MILP presolver removed 5 variables and 0 constraints.
NOTE: The MILP presolver removed 29 constraint coefficients.
NOTE: The MILP presolver modified 7295 constraint coefficients.
NOTE: The presolved problem has 3146 variables, 1003 constraints, and 23677 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The number of block threads has been reduced to 10 threads.
NOTE: The problem has a decomposable structure with 10 blocks. The largest block covers 7.976% of the constraints in the problem.
NOTE: The decomposition subproblems cover 3146 (100%) variables and 768 (76.57%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Iter</th>
<th>Bound</th>
<th>Objective</th>
<th>Master Bound</th>
<th>Best</th>
<th>Integer</th>
<th>Best</th>
<th>LP</th>
<th>IP</th>
<th>CPU</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>41762.0000</td>
<td>37733.0000</td>
<td>37733.0000</td>
<td>9.65%</td>
<td>9.65%</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>41762.0000</td>
<td>38690.0000</td>
<td>38690.0000</td>
<td>7.36%</td>
<td>7.36%</td>
<td>24</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>24</td>
<td>41762.0000</td>
<td>40919.0000</td>
<td>40919.0000</td>
<td>2.02%</td>
<td>2.02%</td>
<td>73</td>
<td>17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30</td>
<td>41762.0000</td>
<td>40919.0000</td>
<td>40919.0000</td>
<td>2.02%</td>
<td>2.02%</td>
<td>91</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>41762.0000</td>
<td>40980.0000</td>
<td>40980.0000</td>
<td>1.87%</td>
<td>1.87%</td>
<td>97</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>41762.0000</td>
<td>40982.0000</td>
<td>40982.0000</td>
<td>1.87%</td>
<td>1.87%</td>
<td>99</td>
<td>25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>40999.0000</td>
<td>40982.0000</td>
<td>40982.0000</td>
<td>0.04%</td>
<td>0.04%</td>
<td>105</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>41</td>
<td>40982.0000</td>
<td>40982.0000</td>
<td>40982.0000</td>
<td>0.00%</td>
<td>0.00%</td>
<td>110</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Node Active Sols Best Best Integer Bound Time Time
0 0 41 40982.0000 40982.0000 0.00% 110 31

NOTE: The Decomposition algorithm used 32 threads.
NOTE: The Decomposition algorithm time is 31.37 seconds.
Example 18.8: Vehicle Routing Problem

The vehicle routing problem (VRP) finds a minimum-cost routing of a fixed number of vehicles to service the demands of a set of customers. Define a set $C = \{2, \ldots, |C| + 1\}$ of customers, and a demand, $d_c$, for each customer $c$. Let $N = C \cup \{1\}$ be the set of nodes, including the vehicle depot, which are designated as node $i = 1$. Let $A = N \times N$ be the set of arcs, $V$ be the set of vehicles (each of which has capacity $L$), and $c_{ij}$ be the travel time from node $i$ to node $j$.

Let $y_{ik}$ be a binary variable that, if set to 1, indicates that node $i$ is visited by vehicle $k$. Let $z_{ijk}$ be a binary variable that, if set to 1, indicates that arc $(i, j)$ is traversed by vehicle $k$, and let $x_{ijk}$ be a continuous variable that denotes the amount of product (flow) on arc $(i, j)$ that is carried by vehicle $k$.

A VRP can be formulated as a MILP as follows:

\[
\begin{align*}
\text{minimize} & \quad \sum_{(i,j) \in A} \sum_{k \in V} c_{ij} z_{ijk} \\
\text{subject to} & \quad \sum_{k \in V} y_{ik} \geq 1 \quad i \in C \quad \text{(Assignment)} \\
& \quad \sum_{(i,j) \in A} z_{ijk} = y_{ik} \quad i \in N, k \in V \quad \text{(LeaveNode)} \\
& \quad \sum_{(j,i) \in A} z_{jik} = y_{ik} \quad i \in N, k \in V \quad \text{(EnterNode)} \\
& \quad \sum_{(j,i) \in A} x_{jik} - \sum_{(i,j) \in A} x_{ijk} = d_i y_{ik} \quad i \in C, k \in V \quad \text{(FlowBalance)} \\
& \quad x_{ijk} \leq L z_{ijk} \quad (i, j) \in A, k \in V \quad \text{(VehicleCapacity)} \\
& \quad y_{1k} = 1 \quad k \in V \quad \text{(Depot)} \\
& \quad x_{ijk} \geq 0 \quad (i, j) \in A, k \in V \\
& \quad y_{ik} \in \{0, 1\} \quad i \in N, k \in V \\
& \quad z_{ijk} \in \{0, 1\} \quad (i, j) \in A, k \in V
\end{align*}
\]

In this formulation, the Assignment constraints ensure that each customer is serviced by at least one vehicle. The objective function ensures that there exists an optimal solution that never assigns a customer to more than one vehicle. The LeaveNode and EnterNode constraints enforce the condition that if node $i$ is visited by vehicle $k$, then vehicle $k$ must use exactly one arc that enters node $i$ and one arc that leaves node $i$. Conversely, if node $i$ is not visited by vehicle $k$, then no arcs that enter or leave node $i$ can be used by vehicle $k$. The FlowBalance constraints define flow conservation at each node for each vehicle. That is, if a node $i$ is visited by vehicle $k$, then the amount of product from vehicle $k$ that enters and leaves that node must equal the demand at that node. Conversely, if node $i$ is not visited by vehicle $k$, then the amount of product from vehicle $k$ that enters and leaves that node must be 0. The VehicleCapacity constraints enforce the condition that the amount of product in each vehicle must always be less than or equal to the vehicle capacity $L$. Finally, the Depot constraints enforce the condition that each vehicle must start and end at the depot node.
In this formulation, the vehicle identifier is arbitrary. Consider a decomposition by vehicle, where the Assignment constraints form the master problem and all other constraints form identical routing subproblems. As described in the section “Special Case: Identical Blocks and Ryan-Foster Branching” on page 873, this is a situation in which an aggregate formulation can greatly improve performance by reducing symmetry. Because you know that there exists an optimal solution that satisfies the master Assignment constraints at equality, you can force the use of Ryan-Foster branching by specifying the option VARSEL=RYANFOSTER.

VRPLIB, located at [http://www.coin-or.org/SYMPHONY/branchandcut/VRP/data/index.htm](http://www.coin-or.org/SYMPHONY/branchandcut/VRP/data/index.htm), is a set of benchmark instances of the VRP. The following data set, vrpdata, represents an instance from VRPLIB that has 22 nodes and eight vehicles (P-n22-k8.vrp), which was originally described in Augerat et al. (1995). The data set lists each node, its coordinates, and its demand.

```plaintext
/* number of vehicles available */
%let num_vehicles = 8;
/* capacity of each vehicle */
%let capacity = 3000;
/* node, x coordinate, y coordinate, demand */
data vrpdata;
    input node x y demand;
datalines;
1 145 215 0
2 151 264 1100
3 159 261 700
4 130 254 800
5 128 252 1400
6 163 247 2100
7 146 246 400
8 161 242 800
9 142 239 100
10 163 236 500
11 148 232 600
12 128 231 1200
13 156 217 1300
14 129 214 1300
15 146 208 300
16 164 208 900
17 141 206 2100
18 147 193 1000
19 164 193 900
20 129 189 2500
21 155 185 1800
22 139 182 700
;
```

The following PROC OPTMODEL statements read in the data, declare the optimization model, and use the decomposition algorithm to solve it:

```plaintext
proc optmodel sessref="mysess";
    /* read the node location and demand data */
    set NODES;
    num x {NODES};
    num y {NODES};
    num demand {NODES};
    num capacity = &capacity;
```
num num_vehicles = &num_vehicles;
read data vrpdata into NODES=[node] x y demand;
set ARCS = {i in NODES, j in NODES: i ne j};
set VEHICLES = 1..num_vehicles;

/* define the depot as node 1 */
num depot = 1;

/* define the arc cost as the rounded Euclidean distance */
num cost {<i,j> in ARCS} = round(sqrt((x[i]-x[j])^2 + (y[i]-y[j])^2));

/* Flow[i,j,k] is the amount of demand carried on arc (i,j) by vehicle k */
var Flow {ARCS, VEHICLES} >= 0 <= capacity;

/* UseNode[i,k] = 1, if and only if node i is serviced by vehicle k */
var UseNode {NODES, VEHICLES} binary;

/* UseArc[i,j,k] = 1, if and only if arc (i,j) is traversed by vehicle k */
var UseArc {ARCS, VEHICLES} binary;

/* minimize the total distance traversed */
min TotalCost = sum {<i,j> in ARCS, k in VEHICLES} cost[i,j] * UseArc[i,j,k];

/* each non-depot node must be serviced by at least one vehicle */
con Assignment {i in NODES diff {depot}}:
  sum {k in VEHICLES} UseNode[i,k] >= 1;

/* each vehicle must start at the depot node */
for {k in VEHICLES} fix UseNode[depot,k] = 1;

/* some vehicle k traverses an arc that leaves node i 
   if and only if UseNode[i,k] = 1 */
con LeaveNode {i in NODES, k in VEHICLES}:
  sum {<(i),j> in ARCS} UseArc[i,j,k] = UseNode[i,k];

/* some vehicle k traverses an arc that enters node i 
   if and only if UseNode[i,k] = 1 */
con EnterNode {i in NODES, k in VEHICLES}:
  sum {<j,(i)> in ARCS} UseArc[j,i,k] = UseNode[i,k];

/* the amount of demand supplied by vehicle k to node i must equal demand 
   if UseNode[i,k] = 1; otherwise, it must equal 0 */
con FlowBalance {i in NODES diff {depot}, k in VEHICLES}:
  sum {<(i),j> in ARCS} Flow[j,i,k] - sum {<(i),j> in ARCS} Flow[i,j,k]
  = demand[i] * UseNode[i,k];

/* if UseArc[i,j,k] = 1, then the flow on arc (i,j) must be at most capacity 
   if UseArc[i,j,k] = 0, then no flow is allowed on arc (i,j) */
con VehicleCapacity {<i,j> in ARCS, k in VEHICLES}:
  Flow[i,j,k] <= Flow[i,j,k].ub * UseArc[i,j,k];

/* decomp by vehicle */
for {i in NODES, k in VEHICLES} do;
  LeaveNode[i,k].block = k;
  EnterNode[i,k].block = k;
end;
The following PROC OPTMODEL statement creates plot data for the optimal routing:

```optmodel
/* create solution data set */
create data solution_data from [i j k]=
    {<i,j> in ARCS, k in VEHICLES: UseArc[i,j,k].sol > 0.5}
    x1=x[i] y1=y[i] x2=x[j] y2=y[j]
    function='line' drawspace='datavalue';
quit;
```

The solution summary is displayed in Output 18.8.1.

### Output 18.8.1  Solution Summary

#### The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Relative Gap</td>
</tr>
<tr>
<td>Absolute Gap</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
</tr>
<tr>
<td>Integer Infeasibility</td>
</tr>
<tr>
<td>Best Bound</td>
</tr>
<tr>
<td>Nodes</td>
</tr>
<tr>
<td>Solutions Found</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

The iteration log is displayed in Output 18.8.2.
Output 18.8.2 Log

NOTE: There were 22 observations read from the data set WORK.VRPDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 7568 variables (0 free, 8 fixed).
NOTE: The problem has 3872 binary and 0 integer variables.
NOTE: The problem has 4237 linear constraints (3696 LE, 520 EQ, 21 GE, 0 range).
NOTE: The problem has 22528 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The SOLVE statement is executing in the distributed computing environment in single-machine mode.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 8 variables and 0 constraints.
NOTE: The MILP presolver removed 16 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 7560 variables, 4237 constraints, and 22512 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: All blocks are identical and the master model is set covering.
WARNING: The master model is not a set partitioning and VARSEL=RYANFOSTER. The objective function must ensure that there exists at least one optimal solution that fulfills all of the master constraints at equality.
NOTE: The Decomposition algorithm is using an aggregate formulation and Ryan-Foster branching.
NOTE: The number of block threads has been reduced to 1 threads.
NOTE: The problem has a decomposable structure with 8 blocks. The largest block covers 12.44% of the constraints in the problem.
NOTE: The decomposition subproblems cover 7560 (100%) variables and 4216 (99.5%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Iter</th>
<th>Best Bound</th>
<th>Master Objective</th>
<th>Best Integer</th>
<th>LP Gap</th>
<th>IP Gap</th>
<th>CPU Time</th>
<th>Real Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOTE: Starting phase 1.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0.0000</td>
<td>20.0000</td>
<td>. 2.00e+01</td>
<td>. 1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0.0000</td>
<td>1.0278</td>
<td>. 1.03e+00</td>
<td>. 5</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>0.0000</td>
<td>0.0000</td>
<td>. 0.00%</td>
<td>. 7</td>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOTE: Starting phase 2.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>112.0000</td>
<td>887.0769</td>
<td>. 692.03%</td>
<td>. 7</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>112.0000</td>
<td>729.0000</td>
<td>729.0000</td>
<td>550.89%</td>
<td>550.89%</td>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>37</td>
<td>222.0577</td>
<td>704.2083</td>
<td>729.0000</td>
<td>217.13%</td>
<td>228.29%</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>38</td>
<td>299.8841</td>
<td>704.2083</td>
<td>729.0000</td>
<td>134.83%</td>
<td>143.09%</td>
<td>17</td>
<td>11</td>
</tr>
<tr>
<td>38</td>
<td>299.8841</td>
<td>683.6000</td>
<td>728.0000</td>
<td>127.95%</td>
<td>142.76%</td>
<td>19</td>
<td>12</td>
</tr>
<tr>
<td>40</td>
<td>311.3889</td>
<td>683.6000</td>
<td>728.0000</td>
<td>119.53%</td>
<td>133.79%</td>
<td>20</td>
<td>12</td>
</tr>
<tr>
<td>42</td>
<td>313.2000</td>
<td>683.6000</td>
<td>694.0000</td>
<td>118.26%</td>
<td>121.58%</td>
<td>22</td>
<td>13</td>
</tr>
<tr>
<td>44</td>
<td>313.2000</td>
<td>653.0000</td>
<td>653.0000</td>
<td>108.49%</td>
<td>108.49%</td>
<td>24</td>
<td>15</td>
</tr>
<tr>
<td>51</td>
<td>329.8333</td>
<td>616.5000</td>
<td>653.0000</td>
<td>86.91%</td>
<td>97.98%</td>
<td>29</td>
<td>18</td>
</tr>
<tr>
<td>52</td>
<td>458.1524</td>
<td>616.5000</td>
<td>653.0000</td>
<td>34.56%</td>
<td>42.53%</td>
<td>30</td>
<td>19</td>
</tr>
<tr>
<td>53</td>
<td>533.0000</td>
<td>607.0000</td>
<td>653.0000</td>
<td>13.88%</td>
<td>22.51%</td>
<td>32</td>
<td>20</td>
</tr>
</tbody>
</table>
The following call to PROC SGPLOT generates a plot of the optimal routing. The plot is displayed in Figure 18.8.3.

```
proc sgplot data=solution_data noautolegend;
   scatter x=x1 y=y1 / datalabel=i;
   vector x=x2 y=y2 / xorigin=x1 yorigin=y1 group=k noarrowheads;
   xaxis display=none;
   yaxis display=none;
run;
```
Example 18.9: ATM Cash Management in Single-Machine Mode

This example describes an optimization model that is used in the management of cash flow for a bank’s automated teller machine (ATM) network. The goal of the model is to determine a replenishment schedule for the bank to use in allocating cash inventory at its branches when servicing a preassigned subset of ATMs. Given a history of withdrawals per day for each ATM, the bank can use SAS forecasting tools to predict the expected cash need. The modeling of this prediction depends on various seasonal factors, including the days of the week, weeks of the month, holidays, typical salary disbursement days, location of the ATMs, and other demographic data. The prediction is a parametric mixture of models whose parameters depend on each ATM.

The optimization model performs a polynomial regression that minimizes the error (measured by the $L_1$ norm) between the predicted and actual withdrawals. The parameter settings in the regression determine the replenishment policy. The amount of cash that is allocated to each day is subject to a budget constraint. In addition, a constraint for each ATM limits the number of days that a cash-out (a situation in which the cash flow is less than the predicted withdrawal) can occur. The goal is to determine a policy for cash distribution that balances the predicted inventory levels while satisfying the budget and cash-out constraints. By keeping too much cash on hand for ATM fulfillment, the bank loses an investment opportunity. Moreover, regulatory agencies in many countries enforce a minimum cash reserve ratio at branch banks; according to regulatory
policy, the cash in ATMs or in transit does not contribute toward this threshold.

**Mixed Integer Nonlinear Programming Formulation**

The most natural formulation for this model is in the form of a mixed integer nonlinear program (MINLP). Let $A$ denote the set of ATMs and $D$ denote the set of days that are used in the training data. The predictive model fit is defined by the following data for each ATM $a$ on each day $d$: $c_{ad}$, $c_{ad}^x$, $c_{ad}^y$, $c_{ad}^z$, and $c_{ad}^u$. The model-fitting parameters define the variables $(x_a, y_a, u_a)$ for each ATM that, when applied to the predictive model, estimate the necessary cash flow per day per ATM. In addition, define a surrogate variable $f_{ad}$ for each ATM on each day that defines the cash inventory (replenished from the branch) minus withdrawals. The variable $f_{ad}$ also represents the error in the regression model. Let $B_d$ define the budget per day, $K_a$ define the limit on cash-outs per ATM, and $w_{ad}$ define the historical withdrawals at a particular ATM on a particular day. Then the following MINLP models this problem:

**Minimize**

\[
\sum_{a \in A} \sum_{d \in D} |f_{ad}|
\]

**Subject to**

\[
\begin{align*}
    c_{ad}^x x_a + c_{ad}^y y_a + c_{ad}^z u_a + c_{ad} - w_{ad} &= f_{ad} & a \in A, d \in D & \text{(CashFlowDefCon)} \\
    \sum_{a \in A} (f_{ad} + w_{ad}) &\leq B_d & d \in D & \text{(BudgetCon)} \\
    |\{d \in D | f_{ad} < 0\}| &\leq K_a & a \in A & \text{(CashOutLimitCon)} \\
    x_a, y_a &\in [0, 1] & a \in A \\
    u_a &\geq 0 & a \in A \\
    f_{ad} &\geq -w_{ad} & a \in A, d \in D
\end{align*}
\]

The CashFlowDefCon constraint defines the surrogate variable $f_{ad}$, which gives the estimated net cash flow. The BudgetCon and CashOutLimitCon constraints ensure that the solution satisfies the budget and cash-out constraints, respectively.

To express this model in a more standard form, you can first use some standard model reformulations to linearize the absolute value and the CashOutLimitCon constraint.

**Linearization of Absolute Value**

A well-known reformulation for linearizing the absolute value of a variable is to introduce one variable for each side of the absolute value. The following systems are equivalent:

**Minimize**

\[
|y|
\]

**Subject to**

\[
Ay \leq b
\]

is equivalent to

**Minimize**

\[
y^+ + y^-
\]

**Subject to**

\[
A(y^+ - y^-) \leq b \\
y^+, y^- \geq 0
\]
Let \( f_{ad}^+ \) and \( f_{ad}^- \) represent the positive and negative parts, respectively, of the net cash flow \( f_{ad} \). Then you can rewrite the model, removing the absolute value, as the following:

\[
\begin{align*}
\text{minimize} & \quad \sum_{a \in A} \sum_{d \in D} (f_{ad}^+ + f_{ad}^-) \\
\text{subject to} & \quad c_{ad}^x x_a + c_{ad}^y y_a + \sum_{a \in A} (f_{ad}^+ - f_{ad}^- + w_{ad}) \leq B_d \\
& \quad \sum_{d \in D} (f_{ad}^-) \leq K_a \\
& \quad x_a, y_a \in [0, 1] \\
& \quad u_a \geq 0 \\
& \quad f_{ad}^+ \geq 0 \\
& \quad f_{ad}^- \in [0, w_{ad}] \\
& \quad v_{ad} \in \{0, 1\}\end{align*}
\]

Modeling the Cash-Out Constraints
To count the number of times a cash-out occurs, you need to introduce a binary variable to keep track of when this event occurs. Let \( v_{ad} \) be an indicator variable that takes the value 1 when the net cash flow is negative. You can model the implication \( f_{ad}^- > 0 \Rightarrow v_{ad} = 1 \), or its contrapositive \( v_{ad} = 0 \Rightarrow f_{ad}^- \leq 0 \), by adding the constraint

\[
f_{ad}^- \leq w_{ad} v_{ad} \quad a \in A, \ d \in D
\]

Now you can model the cash-out constraint by counting the number of days that the net-cash flow is negative for each ATM, as follows:

\[
\sum_{d \in D} v_{ad} \leq K_a \quad a \in A
\]

The MINLP model can now be written as follows:

\[
\begin{align*}
\text{minimize} & \quad \sum_{a \in A} \sum_{d \in D} (f_{ad}^+ + f_{ad}^-) \\
\text{subject to} & \quad c_{ad}^x x_a + c_{ad}^y y_a + \sum_{a \in A} (f_{ad}^+ - f_{ad}^- + w_{ad}) \leq B_d \\
& \quad \sum_{d \in D} (f_{ad}^-) \leq K_a \\
& \quad x_a, y_a \in [0, 1] \\
& \quad u_a \geq 0 \\
& \quad f_{ad}^+ \geq 0 \\
& \quad f_{ad}^- \in [0, w_{ad}] \\
& \quad v_{ad} \in \{0, 1\}\end{align*}
\]
This MINLP is difficult to solve, in part because the prediction function is not convex. Another approach is to use mixed integer linear programming (MILP) to formulate an approximation of the problem, as described in the next section.

**Mixed Integer Linear Programming Approximation**

Because the predictive model is a forecast, finding the optimal parameters that are based on nondeterministic data is not of primary importance. Rather, you want to provide as good a solution as possible in a reasonable amount of time. So using MILP to approximate the MINLP is perfectly acceptable. In the original problem you have products of two continuous variables that are both bounded by 0 (lower bound) and 1 (upper bound). This arrangement enables you to create an approximate linear model by using a few standard modeling reformulations.

**Discretization of Continuous Variables**

The first step is to discretize one of the continuous variables $x_a$. The goal is to transform the product $x_a y_a$ of a continuous variable and another continuous variable instead to the product of a continuous variable and a binary variable. This transformation enables you to linearize the product form.

You must assume some level of approximation by defining a binary variable (from some discrete set) for each possible setting of the continuous variable. For example, if you let $n = 10$, then you allow $x$ to be chosen from the set $\{0.0, 0.1, 0.2, 0.3, ..., 1.0\}$. Let $T = \{0, 1, 2, ..., n\}$ represent the possible steps and $c_t = t/n$. Then you apply the following transformation to variable $x_a$:

$$\sum_{t \in T} c_t x_{at} = x_a$$

$$\sum_{t \in T} x_{at} = 1$$

$$x_{at} \in \{0, 1\} \quad t \in T$$
The MINLP model can now be approximated as the following:

\[
\begin{align*}
\text{minimize} & \quad \sum_{a \in A} \sum_{d \in D} \left( f^+_a f^-_d + f^-_a f^+_d \right) \\
\text{subject to} & \quad c^x_a \sum_{t \in T} c_t x_{at} + c^y_d y_a + \sum_{t \in T} c^z_d \sum_{t \in T} c_t x_{at} y_a + c^w_a u_a + c^v_d - w_{ad} = f^+_a f^-_d + f^-_a f^+_d & a \in A, d \in D \\
& \quad \sum_{t \in T} x_{at} = 1 & a \in A \\
& \quad \sum_{a \in A} \left( f^+_a f^-_d + w_{ad} \right) \leq B_d & d \in D \\
& \quad f^-_d \leq w_{ad} v_{ad} & a \in A, d \in D \\
& \quad \sum_{d \in D} v_{ad} \leq K_a & a \in A \\
& \quad y_a \in [0, 1] & a \in A \\
& \quad u_a \geq 0 & a \in A \\
& \quad f^+_a \geq 0 & a \in A, d \in D \\
& \quad f^-_a \in [0, w_{ad}] & a \in A, d \in D \\
& \quad v_{ad} \in \{0, 1\} & a \in A, d \in D \\
& \quad x_{at} \in \{0, 1\} & a \in A, t \in T \\
\end{align*}
\]

**Linearization of Products**

You still need to linearize the product terms \( x_{at} y_a \) in the cash flow constraint. Because these terms are products of a bounded continuous variable and a binary variable, you can linearize them by introducing for each product another variable, \( z_{at} \), which serves as a surrogate. In general, you know the following relationship between the original variables and their surrogates:

\[
\begin{align*}
& z_t = x_t y_t & t \in T \\
& \sum_{t \in T} x_t = 1 & \text{is equivalent to} \\
& x_t \in \{0, 1\} & t \in T \\
& y \in [0, 1] \\
& z_t \geq 0 & t \in T \\
& z_t \leq x_t & t \in T \\
& \sum_{t \in T} x_t = 1 \\
& \sum_{t \in T} z_t = y \\
& x_t \in \{0, 1\} & t \in T \\
& y \in [0, 1]
\end{align*}
\]

Using this relationship to replace each product form, you now can write the problem as an approximate MILP as follows:
Example 18.9: ATM Cash Management in Single-Machine Mode

\[ \begin{aligned}
\text{minimize} & \sum_{a \in A} \sum_{d \in D} (f_{ad}^+ + f_{ad}^-) \\
\text{subject to} & \sum_{t \in T} c_{at} x_{at} + c_{ad} y_{a} + \\
& c_{ad} \sum_{t \in T} z_{at} + c_{ad} u_{a} + c_{ad} - w_{ad} = f_{ad}^+ - f_{ad}^- \quad a \in A, \ d \in D \\
& \sum_{t \in T} x_{at} = 1 \quad a \in A \\
& \sum_{a \in A} (f_{ad}^+ - f_{ad}^- + w_{ad}) \leq B_d \quad d \in D \quad \text{(BudgetCon)} \\
& f_{ad}^- \leq w_{ad} v_{ad} \quad a \in A, \ d \in D \\
& \sum_{d \in D} v_{ad} \leq K_a \quad a \in A \\
& z_{at} \leq x_{at} \quad a \in A, \ t \in T \\
& \sum_{t \in T} z_{at} = y_{a} \quad a \in A \\
& z_{at} \geq 0 \quad a \in A, \ t \in T \\
& y_{a} \in [0, 1] \quad a \in A \\
& u_{a} \geq 0 \quad a \in A \\
& f_{ad}^+ \geq 0 \quad a \in A, \ d \in D \\
& f_{ad}^- \in [0, w_{ad}] \quad a \in A, \ d \in D \\
& v_{ad} \in \{0, 1\} \quad a \in A, \ d \in D \\
& x_{at} \in \{0, 1\} \quad a \in A, \ t \in T \\
\end{aligned} \]

**PROC OPTMODEL Code**

Because it is difficult to solve the MINLP model directly, the approximate MILP formulation is attractive. Unfortunately, the approximate MILP is much larger than the associated MINLP. Direct methods for solving this MILP do not work well. However, the problem is nicely suited for the decomposition algorithm.

When you examine the structure of the MILP model, you see clearly that the constraints can be easily decomposed by ATM. In fact, the only set of constraints that involve decision variables across ATMs is the BudgetCon constraint. That is, if you relax the budget constraint, you are left with independent blocks of constraints, one for each ATM.

To show how this is done in PROC OPTMODEL, consider the following data sets, which describe an example that tracks 20 ATMs over a period of 100 days. This particular example was submitted to MIPLIB 2010, which is a collection of difficult MILPs in the public domain (Koch et al. 2011).

The first data set, `budget_data`, provides the cash budget on each particular day:
Chapter 18: The Decomposition Algorithm

data budget_data;
  input d $ budget;
  datalines;
  DATE0  70079
  DATE1  66418
  DATE10  52656
  DATE11  50439
  DATE12  58688
  DATE13  45002
  DATE14  52369
... ;

The second data set, cashout_data, provides the limit on the number of cash-outs that are allowed at each ATM:

data cashout_data;
  input a $ cashOutLimit;
  datalines;
  ATM0  31
  ATM1  24
  ATM2  41
  ATM3  43
  ATM4  29
  ATM5  24
  ATM6  52
  ATM7  44
  ATM8  35
  ATM9  48
  ATM10  31
  ATM11  47
  ATM12  26
  ATM13  34
  ATM14  29
  ATM15  32
  ATM16  33
  ATM17  32
  ATM18  43
  ATM19  28
; 

The final data set, polyfit_data, provides the polynomial fit coefficients for each ATM on each date. It also provides the historical cash withdrawals.

data polyfit_data;
  input a $ d $ cx cy cz cu c withdrawal;
  datalines;
  ATM0 DATE0  2822  1984  -1984  1045  1373  780
  ATM0 DATE1  1337  2530  -2530  1510  174  2351
  ATM0 DATE2  2685  -67  67  145  2820  2288
  ATM0 DATE3 -595 -3135  3135  581  3319  1357
... 
  ATM19 DATE96 -734  3392  -3392  162  1648  914
  ATM19 DATE97 -1062  969  -969  444  1746  2264
The following PROC OPTMODEL statements read in the data and define the necessary sets and parameters:

```plaintext
proc optmodel sessref="mysess";
set<str> DATES;
set<str> ATMS;

/* cash budget per date */
um budget{DATES};

/* maximum number of cash-outs allowed at each atm */
um cashOutLimit{ATMS};

/* historical withdrawal amount per atm each date */
um withdrawal{ATMS, DATES};

/* polynomial fit coefficients for predicted cash flow needed */
um c {ATMS, DATES};
um cx{ATMS, DATES};
um cy{ATMS, DATES};
um cz{ATMS, DATES};
um cu{ATMS, DATES};

/* number of points used in approximation of continuous range */
um nSteps = 10;
set STEPS = {0..nSteps};

read data budget_data into DATES=[d] budget;
read data cashout_data into ATMS=[a] cashOutLimit;
read data polyfit_data into [a d] cx cy cz cu c withdrawal;
```

The following statements declare the variables:

```plaintext
var x{ATMS,STEPS} binary;
var v{ATMS,DATES} binary;
var z{ATMS,STEPS} >= 0 <= 1;
var y{ATMS} >= 0 <= 1;
var u{ATMS} >= 0;
var fPlus{ATMS,DATES} >= 0;
var fMinus{ATMS,DATES} >= 0 <= withdrawal[a,d];
```

The following statements declare the objective and the constraints:

```plaintext
min CashFlowDiff =
sum{a in ATMS, d in DATES} (fPlus[a,d] + fMinus[a,d]);

con BudgetCon{d in DATES}:
sum{a in ATMS} (fPlus[a,d] - fMinus[a,d] + withdrawal[a,d])
<= budget[d];

con CashFlowDefCon{a in ATMS, d in DATES}:
cx[a,d] * sum{t in STEPS} (t/nSteps) * x[a,t] +
```

ATM19 DATE98 7676 2308 2308 59 1388 972
ATM19 DATE99 3062 1308 1308 1080 654 698;

ATM19 DATE98 7676 2308 -2308 59 1388 972
ATM19 DATE99 3062 1308 -1308 1080 654 698;
The following statements define the block decomposition by ATM. The .block suffix expects numeric indices, whereas the set<str> ATMS statement declares a set of strings. You can create a mapping from the string identifier to a numeric identifier as follows:

```plaintext
/* create numeric block index */
num blockIndex {ATMS};
num index init 0;
for{a in ATMS} do;
  blockIndex[a] = index;
  index = index + 1;
end;
```

Then, each constraint can be added to its associated ATM block as follows:

```plaintext
/* define blocks for each ATM */
for{a in ATMS} do;
  PickOneStepCon[a].block = blockIndex[a];
  CashOutLimitCon[a].block = blockIndex[a];
  Linear2Con[a].block = blockIndex[a];
  for{d in DATES} do;
    CashFlowDefCon[a,d].block = blockIndex[a];
    CashOutLinkCon[a,d].block = blockIndex[a];
  end;
  for{t in STEPS}
    Linear1Con[a,t].block = blockIndex[a];
  end;
end;
```

The budget constraint links all the ATMs, and it remains in the master problem. Finally, the following statements use the decomposition algorithm to solve the problem to within 1% of proven optimality:

```plaintext
/* solve with the decomposition algorithm */
solve with milp / nthreads=4 relobjgap=0.01 decomp;
```

The solution summary is displayed in Output 18.9.1.
Output 18.9.1 Solution Summary Table

The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Relative Gap</td>
</tr>
<tr>
<td>Absolute Gap</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
</tr>
<tr>
<td>Integer Infeasibility</td>
</tr>
<tr>
<td>Best Bound</td>
</tr>
<tr>
<td>Nodes</td>
</tr>
<tr>
<td>Solutions Found</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 18.9.2.
### Output 18.9.2 Log

NOTE: There were 100 observations read from the data set WORK.BUDGET_DATA.
NOTE: There were 20 observations read from the data set WORK.CASHOUT_DATA.
NOTE: There were 2000 observations read from the data set WORK.POLYFIT_DATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6480 variables (0 free, 0 fixed).
NOTE: The problem has 2220 binary and 0 integer variables.
NOTE: The problem has 4380 linear constraints (2340 LE, 2040 EQ, 0 GE, 0 range).
NOTE: The problem has 58878 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The SOLVE statement is executing in the distributed computing environment in single-machine mode.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 561 variables and 390 constraints.
NOTE: The MILP presolver removed 21279 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 5919 variables, 3990 constraints, and 37599 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value USER is applied.
NOTE: The problem has a decomposable structure with 20 blocks. The largest block covers 5.138% of the constraints in the problem.
NOTE: The decomposition subproblems cover 5919 (100%) variables and 3890 (97.49%) constraints.
NOTE: The deterministic parallel mode is enabled.

<table>
<thead>
<tr>
<th>Iter</th>
<th>Best Bound</th>
<th>Master Objective</th>
<th>Best LP</th>
<th>IP Gap</th>
<th>CPU Real Time</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0000</td>
<td>1.1767</td>
<td>1.18e+00</td>
<td>76</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.00%</td>
<td>76</td>
<td>32</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2.4432e+06</td>
<td>2.7357e+06</td>
<td>11.97%</td>
<td>118</td>
<td>49</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>2.4524e+06</td>
<td>2.4904e+06</td>
<td>2.4904e+06</td>
<td>1.55%</td>
<td>1.55%</td>
<td>167</td>
</tr>
</tbody>
</table>

NOTE: The Decomposition algorithm stopped on the integer RELOBJGAP= option.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active Sols</th>
<th>Best Integer Bound</th>
<th>Best Gap</th>
<th>CPU Real Time</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>6</td>
<td>2.4640e+06</td>
<td>2.4524e+06</td>
<td>0.47%</td>
</tr>
</tbody>
</table>

NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 81.92 seconds.
NOTE: Optimal within relative gap.
NOTE: Objective = 2464019.7445.
Example 18.10: ATM Cash Management in Distributed Mode

This section illustrates how you can use PROC OPTMODEL and the decomposition algorithm in distributed mode. The problem is the same as the one described in Example 18.9 for managing the cash flow of a bank’s network of automated teller machines (ATMs). The only difference between single-machine and distributed mode is that the DISTRIBUTED= option is set to TRUE in the main solver statement. In this example, the CAS session is configured for 20 worker nodes, each having 32 cores.

The following statement changes the operating mode to distributed mode:

```plaintext
/* solve with the decomposition algorithm (in distributed mode) */
solve with milp / distributed=true nthreads=4 relobjgap=0.01 decomp;
```

When you specify DISTRIBUTED=TRUE and NTHREADS=4, each machine in your configured session processes the subproblems by using up to four threads simultaneously (per machine).

The solution summary is displayed in Output 18.10.1.

**Output 18.10.1  Solution Summary Table**

<table>
<thead>
<tr>
<th>The OPTMODEL Procedure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Solution Summary</td>
</tr>
<tr>
<td>Solver</td>
</tr>
<tr>
<td>Algorithm</td>
</tr>
<tr>
<td>Objective Function</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Objective Value</td>
</tr>
<tr>
<td>Relative Gap</td>
</tr>
<tr>
<td>Absolute Gap</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
</tr>
<tr>
<td>Integer Infeasibility</td>
</tr>
<tr>
<td>Best Bound</td>
</tr>
<tr>
<td>Nodes</td>
</tr>
<tr>
<td>Solutions Found</td>
</tr>
<tr>
<td>Iterations</td>
</tr>
<tr>
<td>Presolve Time</td>
</tr>
<tr>
<td>Solution Time</td>
</tr>
</tbody>
</table>

The iteration log, which contains the problem statistics, the progress of the solution, and the optimal objective value, is shown in Output 18.10.2.
Chapter 18: The Decomposition Algorithm

Output 18.10.2 Log

NOTE: There were 100 observations read from the data set WORK.BUDGET_DATA.
NOTE: There were 20 observations read from the data set WORK.CASHOUT_DATA.
NOTE: There were 2000 observations read from the data set WORK.POLYFIT_DATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 6480 variables (0 free, 0 fixed).
NOTE: The problem has 2220 binary and 0 integer variables.
NOTE: The problem has 4380 linear constraints (2340 LE, 2040 EQ, 0 GE, 0 range).
NOTE: The problem has 58878 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The SOLVE statement is executing in the distributed computing environment with 20 worker nodes.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 561 variables and 390 constraints.
NOTE: The MILP presolver removed 21279 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 5919 variables, 3990 constraints, and 37599 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in the distributed computing environment with 20 worker nodes.
NOTE: The DECOMP method value USER is applied.
NOTE: The problem has a decomposable structure with 20 blocks. The largest block covers 5.138% of the constraints in the problem.
NOTE: The decomposition subproblems cover 5919 (100%) variables and 3890 (97.49%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 4 threads.

<table>
<thead>
<tr>
<th>Iter</th>
<th>Best</th>
<th>Master</th>
<th>Best</th>
<th>LP</th>
<th>IP</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bound</td>
<td>Objective</td>
<td>Integer</td>
<td>Gap</td>
<td>Gap</td>
<td>Time</td>
</tr>
<tr>
<td>1</td>
<td>0.0000</td>
<td>1.1767</td>
<td>1.18e+00</td>
<td>.</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0.0000</td>
<td>0.0000</td>
<td>0.00%</td>
<td>.</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Starting phase 1.

<table>
<thead>
<tr>
<th>Iter</th>
<th>Best</th>
<th>Master</th>
<th>Best</th>
<th>LP</th>
<th>IP</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bound</td>
<td>Objective</td>
<td>Integer</td>
<td>Gap</td>
<td>Gap</td>
<td>Time</td>
</tr>
<tr>
<td>3</td>
<td>2.4432e+06</td>
<td>2.7219e+06</td>
<td>11.41%</td>
<td>.</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>2.4542e+06</td>
<td>2.4758e+06</td>
<td>2.4762e+06</td>
<td>0.88%</td>
<td>0.90%</td>
<td>56</td>
</tr>
</tbody>
</table>

NOTE: The Decomposition algorithm stopped on the integer RELOBJGAP= option.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>Best</th>
<th>Best</th>
<th>Gap</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Integer</td>
<td>Bound</td>
<td>Time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
<td>2.4762e+06</td>
<td>2.4542e+06</td>
<td>0.90%</td>
<td>56</td>
</tr>
</tbody>
</table>

NOTE: The Decomposition algorithm used 4 threads.
NOTE: The Decomposition algorithm time is 56.76 seconds.
NOTE: Optimal within relative gap.
NOTE: Objective = 2476233.3421.

Notice how this iteration log differs from the iteration log for single-machine mode in Example 18.9. In this case, the path is slightly altered because each block solve uses four threads (parallel branch-and-cut) to solve the subproblems as opposed to one thread (sequential branch-and-cut) when running in single-machine mode.
Example 18.11: Kidney Donor Exchange and METHOD=SET

This example looks at an application of integer programming to help create a kidney donor exchange. Suppose someone needs a kidney transplant and a family member is willing to be a donor. If the donor and recipient are incompatible (because of blood type, tissue mismatch, and so on), the transplant cannot happen. Now suppose two donor-recipient pairs, \(i\) and \(j\), are in this situation, but donor \(i\) is compatible with recipient \(j\) and donor \(j\) is compatible with recipient \(i\). Then two transplants can take place in a two-way swap, shown in Figure 18.8. More generally, an \(n\)-way swap can be performed involving \(n\) donors and \(n\) recipients (Willingham 2009).

To model this problem, define a directed graph as follows. Each node is an incompatible donor-recipient pair. Link \((i, j)\) exists if the donor from node \(i\) is compatible with the recipient from node \(j\), as shown in Figure 18.9. Let \(N\) define the set of nodes and \(A\) define the set of arcs. The link weight, \(w_{ij}\), is a measure of the quality of the match. By introducing dummy links whose weight is 0, you can also include altruistic donors who have no recipients or recipients who have no donors. The idea is to find a maximum-weight node-disjoint union of directed cycles. You want the union to be node-disjoint so that no kidney is donated more than once, and you want cycles so that the donor from node \(i\) donates a kidney if and only if the recipient from node \(i\) receives a kidney.

Without any other constraints, the problem could be solved as a linear assignment problem. But doing so would allow arbitrarily long cycles in the solution. For practical considerations (such as travel) and to mitigate risk, each cycle must have no more than \(L\) links. The kidney exchange problem is to find a maximum-weight node-disjoint union of short directed cycles.

Define an index set \(M = \{1, \ldots, |N|/2\}\) of candidate disjoint unions of short cycles (called matchings). Let \(x_{ijm}\) be a binary variable that, if set to 1, indicates that arc \((i, j)\) is in a matching \(m\). Let \(y_{im}\) be a binary variable that, if set to 1, indicates that node \(i\) is covered by matching \(m\). In addition, let \(s_i\) be a binary slack variable that, if set to 1, indicates that node \(i\) is not covered by any matching.

The calculation of the default value for the NTHREADS= option in the DECOMPSUBPROB statement is described on page 870.
The kidney donor exchange can be formulated as a MILP as follows:

\[
\text{maximize } \sum_{(i,j) \in A} \sum_{m \in M} w_{ij} x_{ijm} \\
\text{subject to } \sum_{m \in M} y_{im} + s_i = 1 \quad i \in N \quad \text{(Packing)} \\
\sum_{(i,j) \in A} x_{ijm} = y_{im} \quad i \in N, m \in M \quad \text{(Donate)} \\
\sum_{(i,j) \in A} x_{ijm} = y_{jm} \quad j \in N, m \in M \quad \text{(Receive)} \\
\sum_{(i,j) \in A} x_{ijm} \leq L \quad m \in M \quad \text{(Cardinality)} \\
x_{ijm} \in \{0,1\} \quad (i,j) \in A, m \in M \\
y_{im} \in \{0,1\} \quad i \in N, m \in M \\
s_i \in \{0,1\} \quad i \in N
\]

In this formulation, the Packing constraints ensure that each node is covered by at most one matching. The Donate and Receive constraints enforce the condition that if node \(i\) is covered by matching \(m\), then the matching \(m\) must use exactly one arc that leaves node \(i\) (Donate) and one arc that enters node \(i\) (Receive). Conversely, if node \(i\) is not covered by matching \(m\), then no arcs that enter or leave node \(i\) can be used by matching \(m\). The Cardinality constraints enforce the condition that the number of arcs in matching \(m\) must not exceed \(L\).

In this formulation, the matching identifier is arbitrary. Because it is not necessary to cover each incompatible donor-recipient pair (node), the Packing constraints can be modeled by using set partitioning constraints and the slack variable \(s\). Consider a decomposition by matching, in which the Packing constraints form the master problem and all other constraints form identical matching subproblems. As described in the section “Special Case: Identical Blocks and Ryan-Foster Branching” on page 873, this is a situation in which an aggregate formulation and Ryan-Foster branching can greatly improve performance by reducing symmetry.

The following DATA step sets up the problem by first creating a random graph on \(n\) nodes with link probability \(p\) and Uniform(0,1) weight:

```plaintext
/* create random graph on n nodes with arc probability p 
and uniform(0,1) weight */
%let n = 100;
%let p = 0.02;
data ArcData;
call streaminit(1);
do i = 0 to &n - 1;
do j = 0 to &n - 1;
  if i eq j then continue;
  else if rand('UNIFORM') < &p then do;
    weight = rand('UNIFORM');
    output;
  end;
end;
end;un;
```
In this case, you can specify METHOD=SET and let the decomposition algorithm automatically detect the set partitioning master constraints (Packing) and each independent matching subproblem. The following PROC OPTMODEL statements read in the data, declare the optimization model, and use the decomposition algorithm to solve it:

```plaintext
%let max_length = 10;
proc optmodel sessref="mysess";
  set <num,num> ARCS;
  num weight {ARCS};
  read data ArcData into ARCS=[i j] weight;
  print weight;
  set NODES = union {<i,j> in ARCS} {i,j};
  set MATCHINGS = 1..card(NODES)/2;

  /* UseNode[i,m] = 1 if node i is used in matching m, 0 otherwise */
  var UseNode {NODES, MATCHINGS} binary;

  /* UseArc[i,j,m] = 1 if arc (i,j) is used in matching m, 0 otherwise */
  var UseArc {ARCS, MATCHINGS} binary;

  /* maximize total weight of arcs used */
  max TotalWeight
    = sum {<i,j> in ARCS, m in MATCHINGS} weight[i,j] * UseArc[i,j,m];

  /* each node appears in at most one matching */
  /* rewrite as set partitioning (so decomp uses identical blocks)
  sum{} x <= 1 => sum{} x + s = 1, s >= 0 with no associated cost */
  var Slack {NODES} binary;
  con Packing {i in NODES}:
    sum {m in MATCHINGS} UseNode[i,m] + Slack[i] = 1;

  /* at most one recipient for each donor */
  con Donate {i in NODES, m in MATCHINGS}:
    sum {<(i),j> in ARCS} UseArc[i,j,m] = UseNode[i,m];

  /* at most one donor for each recipient */
  con Receive {j in NODES, m in MATCHINGS}:
    sum {<i,(j)> in ARCS} UseArc[i,j,m] = UseNode[j,m];

  /* exclude long matchings */
  con Cardinality {m in MATCHINGS}:
    sum {<i,j> in ARCS} UseArc[i,j,m] <= &max_length;

  /* automatically decompose using METHOD=SET */
  solve with milp / presolver=basic decomp=(method=set);

  /* save solution to a data set */
  create data Solution from
    [m i j]= {m in MATCHINGS, <i,j> in ARCS: UseArc[i,j,m].sol > 0.5}
    weight[i,j];
quit;
```

In this case, the PRESOLVER=BASIC option ensures that the model maintains its specified symmetry,
enabling the algorithm to use the aggregate formulation and Ryan-Foster branching. The solution summary is displayed in Output 18.11.1.

### Output 18.11.1 Solution Summary

#### The OPTMODEL Procedure

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Solver</td>
<td>MILP</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Decomposition</td>
</tr>
<tr>
<td>Objective Function</td>
<td>TotalWeight</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Optimal within Relative Gap</td>
</tr>
<tr>
<td>Objective Value</td>
<td>24.850855395</td>
</tr>
</tbody>
</table>

| Relative Gap       | 2.101533E-14 |
| Absolute Gap       | 5.222489E-13 |
| Primal Infeasibility | 1.776357E-15 |
| Bound Infeasibility | 4.440892E-16 |
| Integer Infeasibility | 4.440892E-16 |

| Best Bound         | 24.850855395 |
| Nodes              | 9 |
| Solutions Found    | 13 |
| Iterations         | 103 |
| Presolve Time      | 0.09 |
| Solution Time      | 70.95 |

The iteration log is displayed in Output 18.11.2.
Output 18.11.2 Log

NOTE: There were 208 observations read from the data set WORK.ARCDATA.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 15092 variables (0 free, 0 fixed).
NOTE: The problem has 15092 binary and 0 integer variables.
NOTE: The problem has 9751 linear constraints (49 LE, 9702 EQ, 0 GE, 0 range).
NOTE: The problem has 45080 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The SOLVE statement is executing in the distributed computing environment in single-machine mode.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value BASIC is applied.
NOTE: The MILP presolver removed 5081 variables and 3366 constraints.
NOTE: The MILP presolver removed 15175 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 10011 variables, 6385 constraints, and 29905 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The Decomposition algorithm is used.
NOTE: The Decomposition algorithm is executing in single-machine mode.
NOTE: The DECOMP method value SET is applied.
NOTE: All blocks are identical and the master model is set partitioning.
NOTE: The Decomposition algorithm is using an aggregate formulation and Ryan-Foster branching.
NOTE: The number of block threads has been reduced to 1 threads.
NOTE: The problem has a decomposable structure with 49 blocks. The largest block covers 2.02% of the constraints in the problem.
NOTE: The decomposition subproblems cover 9947 (99.36%) variables and 6321 (99%) constraints.
NOTE: The deterministic parallel mode is enabled.
NOTE: The Decomposition algorithm is using up to 32 threads.

<table>
<thead>
<tr>
<th>Iter</th>
<th>Best Bound</th>
<th>Master Objective</th>
<th>Best Objective</th>
<th>LP Integer</th>
<th>Gap</th>
<th>IP Gap</th>
<th>CPU Time</th>
<th>Real Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>350.7353</td>
<td>10.9864</td>
<td>10.9864</td>
<td>96.87%</td>
<td>96.87%</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>350.7353</td>
<td>10.9864</td>
<td>10.9864</td>
<td>96.87%</td>
<td>96.87%</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>325.7221</td>
<td>10.9864</td>
<td>10.9864</td>
<td>96.70%</td>
<td>96.63%</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>5</td>
<td>319.9236</td>
<td>14.8476</td>
<td>14.8476</td>
<td>95.36%</td>
<td>95.36%</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>6</td>
<td>319.9236</td>
<td>15.4286</td>
<td>15.4286</td>
<td>95.18%</td>
<td>95.18%</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>8</td>
<td>318.8627</td>
<td>19.5618</td>
<td>19.5618</td>
<td>93.87%</td>
<td>93.87%</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>9</td>
<td>272.3116</td>
<td>19.5618</td>
<td>19.5618</td>
<td>92.82%</td>
<td>92.82%</td>
<td>1.0</td>
<td>1.0</td>
<td>1.0</td>
</tr>
<tr>
<td>10</td>
<td>253.5043</td>
<td>19.5618</td>
<td>19.5618</td>
<td>92.82%</td>
<td>92.82%</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>11</td>
<td>253.5043</td>
<td>19.8454</td>
<td>19.8454</td>
<td>92.17%</td>
<td>92.17%</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>12</td>
<td>197.4825</td>
<td>19.8454</td>
<td>19.8454</td>
<td>89.95%</td>
<td>89.95%</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>16</td>
<td>164.3646</td>
<td>22.1585</td>
<td>19.8454</td>
<td>86.52%</td>
<td>87.93%</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>20</td>
<td>164.3646</td>
<td>23.4248</td>
<td>20.7543</td>
<td>85.75%</td>
<td>87.37%</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>22</td>
<td>164.3646</td>
<td>23.7833</td>
<td>21.3572</td>
<td>85.53%</td>
<td>87.01%</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>23</td>
<td>141.0426</td>
<td>23.9158</td>
<td>21.3572</td>
<td>83.04%</td>
<td>84.86%</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>24</td>
<td>115.2171</td>
<td>24.1040</td>
<td>21.3572</td>
<td>79.08%</td>
<td>81.46%</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>25</td>
<td>94.3847</td>
<td>24.2346</td>
<td>21.3572</td>
<td>74.32%</td>
<td>77.37%</td>
<td>2.0</td>
<td>2.0</td>
<td>2.0</td>
</tr>
</tbody>
</table>
The solution is a set of arcs that define a union of short directed cycles (matchings). The following call to PROC OPTNETWORK extracts the corresponding cycles from the list of arcs and outputs them to the data table mycas.Cycles. The DATA step assumes that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```
data mycas.Solution;
  set Solution;
run;
proc optnetwork
direction = directed
links = mycas.Solution;
linksVar
  from = i
to = j;
cycle
  maxCycles = all
  out = mycas.Cycles;
run;
```

For more information about PROC OPTNETWORK, see SAS Optimization: Network Optimization Procedures. Alternatively, you can extract the cycles by using the SOLVE WITH NETWORK statement in PROC OPTMODEL (see Chapter 15, “The Network Solver”). The optimal donor exchanges from the output data.
Example 18.11: Kidney Donor Exchange and METHOD=SET

Table mycas.Cycles are displayed in Figure 18.10.

**Figure 18.10** Optimal Donor Exchanges

<table>
<thead>
<tr>
<th>cycle=1</th>
<th>order</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>99</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>33</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>24</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>97</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>31</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle=2</th>
<th>order</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>26</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>84</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>53</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>62</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle=3</th>
<th>order</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>56</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>83</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>45</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>63</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>92</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>3</td>
</tr>
</tbody>
</table>
Figure 18.10 continued

<table>
<thead>
<tr>
<th>cycle=4</th>
<th>order</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>93</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>51</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>87</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>78</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>43</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>41</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle=5</th>
<th>order</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>85</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>82</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>73</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle=6</th>
<th>order</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>38</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>79</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>71</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>38</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle=7</th>
<th>order</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>77</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>89</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>77</td>
<td></td>
</tr>
</tbody>
</table>

References


Subject Index

data table options, 33

active nodes
  OPTMILP procedure, 179
  OPTMODEL procedure, MILP solver, 621

active-set method
  overview, 792

active-set primal-dual algorithm, 786

.ACTIVITY_ variable
  DUALOUT= data table, 122, 178

aggregation operators
  OPTMODEL procedure, 341

algorithm, 506, 870

alldifferent constraints
  specifying, 38

array specification, 38

assignment strategy, 29, 459
  MAX, 37
  MIN, 37
  options, 33
  variable, 37, 448

backtracking search, 27, 457

Bard function, 414, 592

basis, 116, 508

bin packing constraints
  specifying, 44

.BLOCK_ variable
  BLOCKS= data table, 871

block-angular structure
  decomposition algorithm, 851, 896, 900, 908, 913

block-diagonal structure
  decomposition algorithm, 851, 868, 892

blocks
  decomposition algorithm, 851, 871

BLOCKS= data table
  blocks, 871
  DECOMP statement, 871
  decomposition algorithm, 871
  variables, 871

branch-and-bound
  control options, 180, 622

branch-and-price
  decomposition algorithm, 872

branching priorities
  OPTMILP procedure, 182
  OPTMODEL procedure, MILP solver, 624

branching priority
  MPS-format data table, 262

branching variable
  OPTMILP procedure, 179
  OPTMODEL procedure, MILP solver, 621

Branin function, 569

CAS tables
  definitions of DUALOUT= data table variables, 236
  definitions of PRIMALOUT= data table variables, 235
dual output data table, 232, 236
input data table, 231
primal output data table, 234, 235

CLOSEFILE statement
  OPTMODEL procedure, 369

CLP examples
  statement and option cross-reference tables, 104

CLP procedure
  data table options, 33
  assignment strategy, 29, 33
  consistency techniques, 29
  constraint data table, 29, 47
details, 46
  domain options, 33
  functional summary, 33
general options, 33
  getting started, 29
  macro variable _ORCLP_, 52
  macro variable _ORCLPEVS_, 53
  objective function options, 33
  options classified by function, 33
  output control options, 34
  overview, 26, 28
  scheduling solver, 46
  selection strategy, 29, 34
  solution data table, 49
  standard CSP statements, 34
  standard solver, 46
  syntax, 33
table of syntax elements, 33

CLP solver
  assignment strategy, 459
  consistency techniques, 459
details, 457
  getting started, 442
  overview, 442, 457
  selection strategy, 459

column generation
decomposition algorithm, 872
columns, 364
complementarity
  OPTMODEL procedure, 362
concurrent LP, 129, 516
consistency techniques, 29, 459
constrained optimization
  overview, 789
constraint bodies
  OPTMODEL procedure, 379
constraint declaration
  OPTMODEL procedure, 289
constraint programming
  finite domain, 27, 458
constraint propagation, 27, 458
constraint satisfaction problem (CSP), 26, 442
  backtracking search, 27, 457
  constraint propagation, 27, 458
  definition, 26, 442
  scheduling CSP, 28
  solving techniques, 27, 457
  standard CSP, 28, 29, 457
constraint status
  LP solver, 518
constraint data table, 29, 35, 47, 49
    _ID_ variable, 47, 49
    _RHS_ variable, 37, 47, 49
    _TYPE_ variable, 47–49
constraints
  OPTMODEL procedure, 274, 376
control flow
  OPTMODEL procedure, 368
conversions
  OPTMODEL procedure, 403
converting MPS-format file
    examples, 268
    MPS2SASD macro, 264
covariance matrix, 782
coverage
  decomposition algorithm, 851, 877, 889, 917
cutting planes
  OPTMILP procedure, 182
  OPTMODEL procedure, MILP solver, 624
Dantzig-Wolfe method
  decomposition algorithm, 872
  block-angular structure, 851, 896, 900, 908, 913
  block-diagonal structure, 851, 868, 892
  blocks, 851, 871
  BLOCKS= data table, 871
  branch-and-price, 872
  column generation, 872
  coverage, 851, 877, 889, 917
  Dantzig-Wolfe method, 872
details, 871
examples, 879
introductory example, 852
Lagrangian decomposition, 908, 909
LP solver, 510
master problem, 850, 851, 872
MILP solver, 619
OPTLP procedure, 118
OPTMILP procedure, 176
overview, 850
pricing out variables, 872
relaxation, 850, 889
Ryan-Foster branching, 874
separable region, 851
set covering, 876
set packing, 877
set partitioning, 874
subproblem, 850, 852, 871, 872
decomposition algorithm examples
  ATM cash management in distributed mode, 937
  ATM cash management in single-machine mode, 926
  bin packing problem, 902
  block-angular structure, 896, 900
  block-diagonal structure, 892
  generalized assignment problem, 885
  kidney donor exchange, 939
  multicommodity flow, 879
  resource allocation, 907
  vehicle routing problem, 920
derivative-free optimization
  LSO solver, 567
details
  LSO solver, 576
displayed output to log
  LSO solver, 581
domain, 26, 35, 442
    bounds, 48
    distribution strategy, 28, 459
dual value
  OPTMODEL procedure, 385
DUALIN= data table
  OPTLP procedure, 119
  variables, 119
dualization, 114, 506
DUALOUT= data table
   OPTLP procedure, 121, 122
   OPTMILP procedure, 178
   variables, 121, 122, 178

element constraints
   specifying, 39
evaluate variable selection strategies, 35
examples, see OPTLP examples, see OPTMODEL
   examples, see OPTQP examples, 29, 442
   alphabet blocks problem, 468
   Balanced Incomplete Block Design, 89
   Car Painting Problem, 64
   Car Sequencing Problem, 71
   converting MPS-format file, 268
   Eight Queens, 31
   Eight Queens problem, 445
   fixed MPS-format file, 267
   free MPS-format file, 267
   logic-based puzzles, 54, 461
   LSO solver, 583
   Magic Square problem, 467
   MPS-format data table, 266
   Pi Day Sudoku, 58
   Pi Day sudoku problem, 464
   Progressive Party Problem, 94
   Radio Link Frequency Assignment Problem, 101
   Scene Allocation Problem, 67
   scene allocation problem, 480
   Scheduling a Major Basketball Conference, 78
   Send More Money, 29
   Send More Money problem, 442
   statement and option cross-reference tables
      (CLP), 104
      Sudoku, 54
      sudoku problem, 462
      work-shift scheduling problem, 471
expressions
   OPTMODEL procedure, 344
fixed MPS-format file
   examples, 267
FOR statement
   OPTMODEL procedure, 368
formatted output
   OPTMODEL procedure, 368
free MPS-format file
   examples, 267
function caching
   LSO solver, 580
function expressions
   OPTMODEL procedure, 347
functional summary
   CLP procedure, 33
   OPTMODEL procedure, 282
genetic algorithms, 577
global cardinality constraints
   specifying, 40
global solution, 362
graph theory and network analysis, 660
_ID_ variable
   constraint data table, 47, 49
identifier expressions
   OPTMODEL procedure, 347
IF expression, 353
IIS
   Irreducible Infeasible Set, 244, 836
   PROC OPTQP statement, 232
   SOLVE WITH QP statement, 831
IIS option
   OPTLP procedure, 133
   OPTMODEL procedure, LP solver, 519
impure functions
   OPTMODEL procedure, 341
index sets, 340
   implicit set slicing, 407
   index-set-item, 349
   OPTMODEL procedure, 348
input data table, 35, 47
integer variables
   OPTMODEL procedure, 384
interior point algorithm
   complementarity, 239, 834
dual infeasibility, 232, 238, 831, 834
duality gap, 231, 239, 831, 834
   overview, 237, 791, 833
   primal infeasibility, 234, 238, 832, 834
   stopping criteria, 238, 834
interior point primal-dual algorithm, 786
intermediate variable, 436
irreducible infeasible set
   OPTLP procedure, 133
OPTMODEL procedure, LP solver, 519
iteration log
crossover algorithm, 128, 516
interior point algorithm, 128, 515
LOGFREQ= option, 232, 831
LOGLEVEL= option, 232
LP solver, 514–516
network simplex algorithm, 127, 514
OPTLP procedure, 126–128
OPTQP procedure, 239
primal and dual simplex algorithms, 126, 514
QP solver, 835
stopping criteria, 238, 834

Johnson function, 595

Karush-Kuhn-Tucker conditions, 363
key columns, 364, 366
key set, 309

Lagrange multipliers, 363
Lagrangian decomposition
decomposition algorithm, 908, 909
Lagrangian function, 363
_LBOUND_ variable
PRIMALOUT= data table, 120, 178
lexicographic constraints
specifying, 41
linear constraints, 47
specifying, 35, 42, 47
linear programming, see also OPTLP procedure, see also OPTMODEL procedure
OPTMODEL procedure, 274
list form
PRINT statement, 321
local minimum
first-order necessary conditions, 363
second-order necessary conditions, 364
local search optimization, 576, 583
local solution, 362
look-ahead schemas, 28, 458
look-back schemas, 28, 458
LP solver
concurrent LP, 516
constraint status, 518
iteration log, 514–516
problem statistics, 517
variable status, 518
LP solver examples
diet problem, 522
finding an irreducible infeasible set, 536
generalized networks, 547
maximum flow, 551
production, inventory, distribution, 554
shortest path, 562
two-person zero-sum game, 533
using the network simplex algorithm, 539
_L_RHS_ variable
DUALOUT= data table, 122, 178
LSO solver
details, 576
displayed output to log, 581
examples, 583
function caching, 580
functional summary, 573
multiple objectives, 578
overview, 567, 577
termination messages, 581
time limit, 575
LSO solver examples
bound-constrained optimization, 569
introductory examples, 569–571
linear constraint, 570
maximum-likelihood estimates, 571
nonlinear constraints, 571

macro variable
_ORCLP_. 52
_ORCLPEVS_. 53
.OROPTMODEL_., 408, 460, 520, 740, 801, 837

macro variable
.OROPTMILP_., 191

OPTMODEL procedure
.OROPTMODEL_. variable, 582
OROPTLP
.OROPTLP_., 135
OROPTMODEL
.OROPTMODEL_., 629
master problem
decomposition algorithm, 850, 851, 872
matrix form
PRINT statement, 322
memory limit, 21
method, 862
migration to PROC OPTMODEL
from PROC NETFLOW, 547, 551, 554, 562
from PROC NLP, 410
MILP solver
problem statistics, 628
MILP solver examples
facility location, 203, 642
miplib, 198
multicommodity problems, 636
scheduling, 211, 632
simple integer linear program, 194
traveling salesman problem, 651
model building
PROC OPTMODEL, 13
model update
OPTMODEL procedure, 392
Moore-Penrose conditions, 797
MPS format, 331
MPS-format data table, 255
  bound type, 261
  branching priority, 262
  converting MPS-format file, 268
  examples, 266
  length of variables, 265
  range, 260
  row type, 257
  sections, 257
  variables, 256
MPS-format file, 264
MPS2SASD macro
  converting MPS-format file, 264
%MPS2SASD macro, 227, 231
multicommodity problems
  MILP solver examples, 636
multiple objectives
  LSO solver, 578
multiple solutions
  OPTMODEL procedure, 398
multiple subproblems
  OPTMODEL procedure, 397
multithreaded parallel computing, 21
network solver
  overview, 660
NLP solver
  covariance matrix, 782, 797
  eigenvalue tolerance, 783
  singularity criterion, 782, 784
  solver termination messages, 800
NLP solver examples
  finding an irreducible infeasible set, 820
  maximum likelihood Weibull estimation, 818
  solving highly nonlinear optimization problems, 803
  solving large-scale NLP problems, 810
  solving NLP problems that have several local
    minima, 811
  solving NLP problems with range constraints, 807
  solving unconstrained and bound-constrained
    optimization problems, 805
node selection
  OPTMILP procedure, 181
  OPTMODEL procedure, MILP solver, 622
number of threads, 118, 176, 621
nthreads, 576
numerical difficulties, 22

_OBJCOEF_ variable
  PRIMALOUT= data table, 120
objective declarations
  OPTMODEL procedure, 276, 291
objective function, 48
objective functions
  OPTMODEL procedure, 274, 276, 291
objective value
  OPTMODEL procedure, 274
objectives
  OPTMODEL procedure, 276
_OBJ_ID_ variable
  DUALOUT= data table, 121, 178
  PRIMALOUT= data table, 120, 177
ODS
  ODS table names, 240
  OPTQP procedure, 240
  PRINTLEVEL= option, 234
  problem statistics, 243, 835
  summaries, 828
ODS table names
  CLP procedure, 50
  OPTLP procedure, 129
  OPTMILP procedure, 186
  OPTMODEL procedure, 371
CLP procedure
  ODS table names, 50
ODS variable names
  OPTMODEL procedure, 372
operators
  OPTMODEL procedure, 345
optimal solution
  OPTMODEL procedure, 274
optimal value
  OPTMODEL procedure, 274
optimality conditions
  OPTMODEL procedure, 361
optimization
  introduction, 9
  optimization modeling language, 272
  optimization variable
    OPTMODEL procedure, 274
  optimization variables
    OPTMODEL procedure, 276
options classified by function, see functional summary
OPTLP examples
  diet optimization problem, 142
  finding an irreducible infeasible set, 153
  oil refinery problem, 137
  reoptimizing after adding a new constraint, 149
  reoptimizing after modifying the objective
    function, 145
  reoptimizing after modifying the right-hand side,
    147
using the interior point algorithm, 141
using the network simplex algorithm, 156

OPTLP procedure
algorithm, 113
basis, 116
concurrent LP, 129
crossover, 117
data, 112
decomposition algorithm, 118
definitions of DUALIN= data table variables, 119
definitions of DUALOUT= data set variables, 121, 122
definitions of DUALOUT= data table variables, 121, 122
definitions of DUALOUT= data set variables, 121
definitions of DUALOUT= data table variables, 122
definitions of PRIMALIN= data table variables, 119
definitions of PRIMALIN= data table variables, 119
definitions of PRIMALOUT= data set variables, 120, 121
definitions of PRIMALOUT= data table variables, 120
DUALIN= data table, 119
duality gap, 118
dualization, 114
DUALOUT= data table, 121, 122
feasibility tolerance, 115
functional summary, 111
IIS option, 133
interior point algorithm, 124
introductory example, 108
iteration log, 126–128
memory limit, 21
multithreaded parallel computing, 21
network simplex algorithm, 124
number of threads, 118
numerical difficulties, 22
ODS table names, 186
_OROPTLP_ macro variable, 135
preprocessing, 114
pricer, 117
PRIMALIN= data table, 119
PRIMALOUT= data table, 120, 121
problem statistics, 132
queue size, 117
random seed, 117
scaling, 117

OPTMILP procedure
active nodes, 179
branch-and-bound, 180
branching priorities, 182
branching variable, 172
cutting planes, 179
data, 167
decomposition algorithm, 176
definitions of DUALOUT= data table variables, 178
definitions of DUALOUT= data table variables, 178
definitions of PRIMALIN= data table variables, 177
definitions of PRIMALOUT= data table variables, 177, 178
DUALOUT= data table, 178
functional summary, 165
introductory example, 162
memory limit, 21
multithreaded parallel computing, 21
node selection, 181
number of threads, 176
numerical difficulties, 22
OEDS table names, 186
_ORTOPTMILP_ macro variable, 191
presolver, 182
PRIMALIN= data table, 177
PRIMALOUT= data table, 177, 178
probing, 182
problem statistics, 190
random seed, 171
variable selection, 181

OPTMODEL examples
chemical equilibrium, 435
matrix square root, 413
model construction, 416
multiple subproblems, 422
reading from and creating a data set, 414
set manipulation, 421
sparse modeling, 430
SUBMIT statement, 426
traveling salesman problem, 426

OPTMODEL expression extensions, 351
aggregation expression, 355

OPTMODEL procedure
aggregation operators, 341
CLOSEFILE statement, 369
complementarity, 362
constraint bodies, 379
constraints, 376
control flow, 368
conversions, 403
data set input/output, 364
declaration statements, 288
dual value, 385
dualization, 506
expressions, 344
FCMP routines, 403
feasible region, 362
feasible solution, 362
FILE statement, 369
first-order necessary conditions, 363
FOR statement, 368
formatted output, 368
function expressions, 347
functional summary, 282
global solution, 362
identifier expressions, 347
impure functions, 341
index sets, 348
integer variables, 384
Karush-Kuhn-Tucker conditions, 363
Lagrange multipliers, 363
Lagrangian function, 363
local solution, 362
macro variable _OROPTMODEL_, 408
memory limit, 21
model update, 392
multiple solutions, 398
multiple subproblems, 397
multithreaded parallel computing, 21
numerical difficulties, 22
objective declarations, 276, 291
ODS table names, 371
ODS variable names, 372
operators, 345
optimality conditions, 361
optimization variables, 276
options classified by function, 282
_OROPTMODEL_NUM_KEYS_ parameter, 408
_OROPTMODEL_NUM_ parameter, 408
_OROPTMODEL_STR_KEYS_ parameter, 408
_OROPTMODEL_STR_ parameter, 408
overview, 272
parameters, 292, 340
presolver, 392
primary expressions, 346
PRINT statement, 369
programming statements, 297
PUT statement, 368
range constraints, 387
reduced costs, 391
RESET OPTIONS statement, 400
second-order necessary conditions, 364
second-order sufficient conditions, 364
_SOLUTION_STATUS_ parameter, 408
_STATUS_ parameter, 408
strict local solution, 362
suffix names, 379, 381
table of syntax elements, 282
threaded and distributed processing, 407
variable declaration, 276, 296
OPTMODEL procedure, CLP solver
macro variable _OROPTMODEL_, 460
OPTMODEL procedure, DECOMP algorithm
method, 862
OPTMODEL procedure, DECOMPSUBPROB
algorithm, 870
OPTMODEL procedure, LP solver
algorithm2, 506
basis, 508
feasibility tolerance, 507
functional summary, 504
IIS option, 519
introductory example, 502
macro variable _OROPTMODEL_, 520
network simplex algorithm, 511
preprocessing, 507
presolver, 507
pricing, 509
queue size, 509
scaling, 509
solver, 506
OPTMODEL procedure, MILP solver
active nodes, 621
branch-and-bound, 622
branching priorities, 624
branching variable, 621
cutting planes, 624
functional summary, 609
introductory example, 608
node selection, 622
_OROPTMODEL_ macro variable, 629
presolve, 624
probing, 624
variable selection, 623
OPTMODEL procedure, network solver
macro variable _OROPTMODEL_, 740
OPTMODEL procedure, NLP solver
details, 789
functional summary, 781
introductory examples, 771
macro variable _OROPTMODEL_, 801
solver, 786
technique, 786
OPTQP examples
covariance matrix, 249
data fitting, 246
estimation, 246
linear least squares, 246
Markowitz model, 249
portfolio optimization, 249
portfolio selection with transactions, 252
short-sell, 251
OPTQP procedure
  examples, 246
  functional summary, 231
  IIS, 244
  interior point algorithm, 237
  iteration log, 239
  memory limit, 21
  multithreaded parallel computing, 21
  numerical difficulties, 22
  ODS, 240
  options, 231
  _OROPTQP_ macro variable, 245
  output data tables, 235
  overview, 223
  QPS format, 226
  quadratic programming, 223
  syntax, 230
  _ORCLPEVS_ macro variable, 53
  _ORCLP_ macro variable, 52
  _OROPTMODEL_ macro variable, 460, 520, 740, 801, 837
  _OROPTMODEL_NUM_KEYS_ parameter
    _OROPTMODEL_ procedure, 408
  _OROPTMODEL_NUM_ parameter
    _OROPTMODEL_ procedure, 408
  _OROPTMODEL_STR_KEYS_ parameter
    _OROPTMODEL_ procedure, 408
  _OROPTMODEL_STR_ parameter
    _OROPTMODEL_ procedure, 408
  _OROPTQP_ macro variable, 245
  output control options, 34
  output data table variables
    _ACTIVITY_ variable, 237
    _LBOUND_ variable, 235
    _L_RHS_ variable, 237
    _TYPE_ variable, 235
    _OBJ_ID_ variable, 235, 236
    _RHS_ variable, 237
    _RHS_ID_ variable, 235, 236
    _ROW_ variable, 236
    _STATUS_ variable, 236, 237
    _VAR_ variable, 235
    _UBOUND_ variable, 235
    _U_RHS_ variable, 237
    _VALUE_ variable, 236, 237
    _VAR_ variable, 235
  output data table, 36
  overview
    LSO solver, 577
    network solver, 660
    optimization, 9
    OPTMODEL procedure, 272
    OPTQP procedure, 223
  QP solver, 825
  parallel processing, 873
  parameters, 343
    initialization, 343
    _OPTMODEL_ procedure, 292, 340
    _OROPTMODEL_NUM_KEYS_ parameter, 408
    _OROPTMODEL_NUM_ parameter, 408
    _OROPTMODEL_STR_KEYS_ parameter, 408
    _OROPTMODEL_STR_ parameter, 408
    parameter declarations, 292
    parameter options, 292
    _SOLUTION_STATUS_ parameter, 408
    _STATUS_ parameter, 408
    _PDIGITS_ option, 370
    positive semidefinite matrix, 224, 826
    predicates, 450
    preprocessing, 36, 37
    presolve
      _OPTMILP_ procedure, 182
      _OPTMODEL_ procedure, MILP solver, 624
    presolver, 114, 507
    pricing, 117, 509
    pricing out variables
      decomposition algorithm, 872
    _PRIMALIN= data table
      _OPTLP_ procedure, 119
      _OPTMILP_ procedure, 177
      variables, 119, 177
    _PRIMALOUT= data table
      _OPTLP_ procedure, 120, 121
      _OPTMILP_ procedure, 177, 178
      variables, 120, 121, 177, 178
    primary expressions
      _OPTMODEL_ procedure, 346
    PRINT statement
      list form, 321
      matrix form, 322
      _OPTMODEL_ procedure, 369
    probing
      _OPTMILP_ procedure, 182
      _OPTMODEL_ procedure, MILP solver, 624
    PROC _OPTMODEL_
      model building, 13
    PROC _OPTQP_
      dual infeasibility, 232
      dual output data table, 232
      duality gap, 231
      IIS, 232
      input data table, 231
      log frequency, 232
      log level, 232
      maximum iteration, 232
      maximum time, 233
number of threads, 233
objective sense, 233
ODS print level, 234
parallel mode, 233
presolver level, 233
primal infeasibility, 234
primal output data table, 234
time type, 234
programming statements
control, 298
general, 298
input/output, 298
looping, 298
model, 298
OPTMODEL procedure, 297
PUT statement
OPTMODEL procedure, 368
PWIDTH= option, 370
QP solver
examples, 838
functional summary, 830
IIS, 836
interior point algorithm, 833
iteration log, 835
macro variable _OROPTMODEL_, 837
options, 831
overview, 825
problem statistics, 835
QPS format, 829
quadratic programming, 825
syntax, 830
QP solver examples
covariance matrix, 841
data fitting, 839
estimation, 839
linear least squares, 839
Markowitz model, 841
portfolio optimization, 841
portfolio selection with transactions, 845
short-sell, 844
QPS format, 332
QPS format file, 264
quadratic programming
overview, 223, 825
quadratic matrix, 224, 825
queue size, 117, 509
random numbers
seed, 576
random seed, 117, 171, 509, 614
range constraints
OPTMODEL procedure, 387
_R_COST_ variable
PRIMALOUT= data table, 121
READ DATA statement
trim option, 328
reduced costs
OPTMODEL procedure, 391
reified constraints
specifying, 45
relaxation
decomposition algorithm, 850, 889
RESET OPTIONS statement
OPTMODEL procedure, 400
_RHS_ variable
constraint data table, 37, 47, 49
DUALOUT= data table, 122, 178
_RHS_ID_ variable
DUALOUT= data table, 121, 178
PRIMALOUT= data table, 120, 177
_ROW_ variable
BLOCKS= data table, 871
DUALIN= data table, 119
DUALOUT= data table, 121, 178
Ryan-Foster branching
decomposition algorithm, 874
satisfiability problem (SAT), 27, 457
scalar types, 292, 342
scaling, 117, 509
scheduling
MILP solver examples, 632
scheduling CSP, 28
scheduling mode
CLP procedure, 459
second-order necessary conditions, 364
local minimum, 364
second-order sufficient conditions, 364
strict local minimum, 364
selection strategy, 29, 459
DOMDDEG, 37
DOMDEG, 37
DOMWDEG, 38
FIFO, 37
MAXC, 37
MAXCS, 37
MINR, 29, 37, 459
MINRMAXC, 37
options, 34
value, 37
variable, 37
WDEG, 38
separable region
decomposition algorithm, 851
set covering
decomposition algorithm, 876
set packing
decomposition algorithm, 877
set partitioning
decomposition algorithm, 874
set types, 292
singularity, 797
absolute singularity criterion, 782
relative singularity criterion, 784
solution data table, 49
_SOLUTION_STATUS_ parameter
OPTMODEL procedure, 408
SOLVE WITH LP statement
crossover, 509
duality gap, 510
SOLVE WITH MILP
number of threads, 621
SOLVE WITH QP statement
dual infeasibility, 831
duality gap, 831
IIS, 831
log frequency, 831
maximum iteration, 831
maximum time, 831
number of threads, 832
presolver level, 832
primal infeasibility, 832
time type, 832
solver types, 46
standard CSP, 28, 29, 457
_STATUS_ parameter
OPTMODEL procedure, 408
_STATUS_ variable
 DUALIN= data table, 119
 DUALOUT= data table, 122
 PRIMALIN= data table, 119
 PRIMALOUT= data table, 120
strict local minimum
second-order sufficient conditions, 364
strict local solution, 362
subproblem
decomposition algorithm, 850, 852, 871, 872
suffix names
OPTMODEL procedure, 379, 381
suffixes, 366, 380
syntax tables, 33
table of syntax elements, see functional summary
termination criteria, 36, 37
time limit, 575
termination messages
LSO solver, 581
threaded and distributed processing
OPTMODEL procedure, 407
traveling salesman problem
MILP solver examples, 651
trim option
READ DATA statement, 328
tuples, 342
_TYPE_ variable
constraint data table, 47–49
 DUALOUT= data table, 121, 178
 PRIMALOUT= data table, 120, 177
 _UBOUND_ variable
 PRIMALIN= data table, 119, 177
 PRIMALOUT= data table, 120, 177
_VALUE_ variable
 DUALOUT= data table, 122
 PRIMALIN= data table, 122
 PRIMALOUT= data table, 120, 178
_VAR_ variable
 PRIMALIN= data table, 119, 177
 PRIMALOUT= data table, 120, 177
variable declaration
OPTMODEL procedure, 274
 variable
PRIMALIN= data table, 119, 177
 PRIMALOUT= data table, 120, 177
variable selection, 28, 458
 OPTMILP procedure, 181
 OPTMODEL procedure, MILP solver, 623
variable status
LP solver, 518
Syntax Index

%MPS2SASD macro parameters
  FORMAT= option, 265
  MAXLEN= option, 265
  MPSFILE= option, 265
  OUTDATA= option, 265

ABSFCONV= option
  SOLVE WITH LSO statement, 575

ABSOBJGAP= option
  DECOMP statement, 860
  PROC OPTMILP statement, 168
  SOLVE WITH MILP statement, 612

ABSOBJGAP= suboption
  TSP= option, 678

ABSOLUTEOBJECTIVEGAP= option
  DECOMP statement, 860

ALGORITHM2= option
  PROC OPTLP statement, 114
  SOLVE WITH LP statement, 506

ALGORITHM= option
  DECOMP statement, 870
  PROC OPTLP statement, 113
  SOLVE WITH LP statement, 506
  SOLVE WITH NLP statement, 786

ALGORITHM= suboption
  CONNECTEDCOMPONENTS= option, 675
  CYCLE= option, 676

ALLDIFF predicate, 451

ALLDIFF statement, 29, 38

ALLSOLNS option, see FINDALLSOLNS option

AND aggregation expression
  OPTMODEL expression extensions, 351

ARRAY statement, 29, 38

ARTPOINTS= suboption
  OUT= option, 673

ASINGULAR= option
  SOLVE WITH NLP statement, 782, 797

assignment statement
  OPTMODEL procedure, 298

ASSIGNMENTS= suboption
  OUT= option, 673

BASIS= option
  PROC OPTLP statement, 116
  SOLVE WITH LP statement, 508

BICONCOMP= suboption
  OUT= option, 673

BICONNECTEDCOMPONENTS option
  algorithm options, 675

BLOCKS= option
  DECOMP statement, 860

CACHEMAX= option
  SOLVE WITH LSO statement, 575

CACHETOL= option
  SOLVE WITH LSO statement, 575

CALL statement
  OPTMODEL procedure, 298

CARD function
  OPTMODEL expression extensions, 351

CDigits= option
  PROC OPTMODEL statement, 284

CLIQUE= option
  algorithm options, 675

CLIQUEs= suboption
  OUT= option, 673

CLOSEFILE statement
  OPTMODEL procedure, 299

CLOSURE= suboption
  OUT= option, 673

CLP solver, 447

COFOR statement
  OPTMODEL procedure, 299

COL keyword
  CREATE DATA statement, 306, 309
  READ DATA statement, 329

CONCOMP= suboption
  OUT= option, 673

CONCURRENT= option
  PROC OPTMILP statement, 175

SOLVE WITH MILP statement, 620

CONDATA= option
  PROC CLP statement, 29, 35, 43, 45, 47, 49

CONFLICTSEARCH= option
  PROC OPTMILP statement, 172

SOLVE WITH MILP statement, 615

CONNECTEDCOMPONENTS= option
  algorithm options, 675

CONSTRAINT option
  EXPAND statement, 314

CONSTRAINT statement
  OPTMODEL procedure, 289

CONTINUE statement
  OPTMODEL procedure, 305

COV= option
  SOLVE WITH NLP statement, 782

COVest=() option
SOLVE WITH NLP statement, 795
COVEST=() option
SOLVE WITH NLP statement, 782
COVOUT= option
SOLVE WITH NLP statement, 783
COVSING= option
SOLVE WITH NLP statement, 783, 797
CREATE DATA statement
COL keyword, 306, 309
OPTMODEL procedure, 305
CROSS expression
OPTMODEL expression extensions, 352
CROSSOVER= option
PROC OPTLP statement, 117
SOLVE WITH LP statement, 509
CUTCLIQUE= option
PROC OPTMILP statement, 174
SOLVE WITH MILP statement, 618
CUTFLOWCOVER= option
PROC OPTMILP statement, 174
SOLVE WITH MILP statement, 618
CUTFLOWPATH= option
PROC OPTMILP statement, 174
SOLVE WITH MILP statement, 618
CUTGOMORY= option
PROC OPTMILP statement, 174
SOLVE WITH MILP statement, 618
CUTGUB= option
PROC OPTMILP statement, 174
SOLVE WITH MILP statement, 618
CUTIMPLIED= option
PROC OPTMILP statement, 174
SOLVE WITH MILP statement, 618
CUTKAPSPACK= option
PROC OPTMILP statement, 174
SOLVE WITH MILP statement, 618
CUTLAP= option
PROC OPTMILP statement, 175
SOLVE WITH MILP statement, 619
CUTMILIFTED= option
PROC OPTMILP statement, 175
SOLVE WITH MILP statement, 619
CUTMIR= option
PROC OPTMILP statement, 175
SOLVE WITH MILP statement, 619
CUTMULTICOMMODITY= option
PROC OPTMILP statement, 175
SOLVE WITH MILP statement, 619
CUTOFF= option
PROC OPTMILP statement, 168
SOLVE WITH MILP statement, 612
CUTOFF= suboption
TSP= option, 678
CUTS= option
PROC OPTMILP statement, 175
SOLVE WITH MILP statement, 619
CUTSETS= suboption
OUT= option, 673
CUTSFACCTOR= option
PROC OPTMILP statement, 175
SOLVE WITH MILP statement, 619
CUTSTRATEGY= option
PROC OPTMILP statement, 175
SOLVE WITH MILP statement, 619
CUTSTRATEGY= suboption
TSP= option, 678
CUTZEROHALF= option
PROC OPTMILP statement, 175
SOLVE WITH MILP statement, 619
CYCLE= option
algorithm options, 675
CYCLES= suboption
OUT= option, 673
DATA= option
PROC OPTLP statement, 112
PROC OPTMILP statement, 167
PROC OPTQP statement, 231
DECOMPMASTERIP=() option
SOLVE WITH MILP statement, 620
DECOMPMASTERIP statement
DECOMP option, 865
OPTMILP procedure, 176
DECOMPMASTER=() option
SOLVE WITH LP statement, 510
SOLVE WITH MILP statement, 620
DECOMPMASTER statement
DECOMP option, 863
OPTLP procedure, 118
OPTMILP procedure, 176
DECOMP option
DECOMPMASTERIP statement, 865
DECOMPMASTER statement, 863
DECOMP statement, 860
DECOMPSUBPROB statement, 867
syntax, 855
DECOMP=() option
SOLVE WITH LP statement, 510
SOLVE WITH MILP statement, 619
DECOMP statement
ABSOBJGAP= option, 860
ABSOLUTEOBJECTIVEGAP= option, 860
BLOCKS= option, 860
DECOMP option, 860
HYBRID= option, 861
LOGFREQ= option, 861
LOGLEVEL= option, 861
MAXBLOCKS= option, 862
MAXITER= option, 862
METHOD= option, 862
NBLOCKS= option, 862
NTHREADS= option, 863
NUMBLOCKS= option, 862
NUMTHREADS= option, 863
OPTLP procedure, 118
OPTMILP procedure, 176
RELOBJGAP= option, 863
DECOMPSUBPROB=() option
  SOLVE WITH LP statement, 510
  SOLVE WITH MILP statement, 620
DECOMPSUBPROB statement
  DECOMP option, 867
  OPTLP procedure, 118
  OPTMILP procedure, 176
DECOMPMASTER statement
  INITPRESOLVER= option, 864
  NTHREADS= option, 865
  NUMTHREADS= option, 865
DECOMPMASTERIP statement
  NTHREADS= option, 866
  NUMTHREADS= option, 866
  PRIMALIN= option, 867
DECOMPSUBPROB statement
  ALGORITHM= option, 870
  INITPRESOLVER= option, 870
  NTHREADS= option, 870
  NUMTHREADS= option, 870
  PRIMALIN= option, 871
  SOL= option, 870
  SOLVER= option, 870
DECRMAXTIME option
  PROC CLP statement, 35
DETERMINISTIC= option
  PROC OPTLP statement, 118
  PROC OPTMILP statement, 176
  SOLVE WITH LP statement, 510
  SOLVE WITH MILP statement, 620
DIFF expression
  OPTMODEL expression extensions, 352
DISTRIBUTED= option
  PROC OPTMILP statement, 176
  SOLVE WITH MILP statement, 620
DO statement
  END keyword, 311
  OPTMODEL procedure, 311
DO statement, iterative
  END keyword, 311
  OPTMODEL procedure, 311
  UNTIL keyword, 311
  WHILE keyword, 311
DO UNTIL statement
  END keyword, 312
  OPTMODEL procedure, 313
DO WHILE statement
  END keyword, 313
  OPTMODEL procedure, 313
DOM= option, see DOMAIN= option
  PROC CLP statement, 35
DOMDEG selection strategy, 37
DOMDEG selection strategy, 37
DOMWDEG selection strategy, 38
DROP statement
  OPTMODEL procedure, 313
DUALIN= option
  PROC OPTLP statement, 112
DUALITYGAP= option
  PROC OPTLP statement, 118
  PROC OPTQP statement, 231
  SOLVE WITH LP statement, 510
  SOLVE WITH QP statement, 831
DUALIZE= option
  PROC OPTLP statement, 114
  SOLVE WITH LP statement, 506
DUALOUT= option
  PROC OPTLP statement, 112
  PROC OPTMILP statement, 167
  PROC OPTQP statement, 232
DUALTOL= option
  PROC OPTQP statement, 232
  SOLVE WITH QP statement, 831
ELEMENT predicate, 452
ELEMENT statement, 29, 39
ELSE keyword
  IF statement, 318
EMPHASIS= option
  PROC OPTLP statement, 284
  PROC OPTMILP statement, 168
  SOLVE WITH MILP statement, 612
END keyword
  DO statement, 311
  DO statement, iterative, 311
  DO UNTIL statement, 312
  DO WHILE statement, 313
Equality expression
  OPTMODEL expression extensions, 352
ERRORLIMIT= option
  PROC OPTMODEL statement, 284
EVALVARSEL= option
  PROC CLP statement, 35
EXPAND statement
  CONSTRAINT option, 314
  FIX option, 315
  IIS option, 315
  IMPVAR option, 315
  OBJECTIVE option, 315
OMITTED option, 315
OPTMODEL procedure, 314
SOLVE option, 314
VAR option, 315

FD= option
PROC OPTMODEL statement, 284

FDIGITS= option
PROC OPTMODEL statement, 285

FEASTOL= option
PROC OPTLP statement, 115
PROC OPTMILP statement, 168
PROC OPTQP statement, 234
SOLVE WITH LP statement, 507
SOLVE WITH LSO statement, 575
SOLVE WITH MILP statement, 612
SOLVE WITH NLP statement, 787
SOLVE WITH QP statement, 832

FIFO selection strategy, 37

FILE statement
OPTMODEL procedure, 316

FINDALL option, see FINDALLSOLNS option

FINDALLSOLNS option
PROC CLP statement, 36
SOLVE WITH CLP statement, 448

FIX option
EXPAND statement, 315

FIX statement
OPTMODEL procedure, 317

FLOW= suboption
OUT= option, 673

FOR statement
OPTMODEL procedure, 318

FORCEFD= option
PROC OPTMODEL statement, 285

FORCEPRESOLVE= option
PROC OPTMODEL statement, 285

FOREACH statement, 29, 38, 40

FOREST= suboption
OUT= option, 673

FORMAT= option
%MPS2SASD macro parameters, 265

function expressions
OF keyword, 348

GCC predicate, 453
GCC statement, 29, 40

GRAPH_DIRECTION= option
SOLVE WITH NETWORK statement, 670

HEURISTICS= option
PROC OPTMILP statement, 171
SOLVE WITH MILP statement, 615

HEURISTICS= suboption
TSP= option, 679

HYBRID= option
DECOMP statement, 861

IF expression
OPTMODEL expression extensions, 353

IF statement
ELSE keyword, 318
OPTMODEL procedure, 318
THEN keyword, 318

IIS option
EXPAND statement, 315

IIS= option
PROC OPTLP statement, 114
PROC OPTQP statement, 232
SOLVE WITH LP statement, 506
SOLVE WITH NLP statement, 787
SOLVE WITH QP statement, 831

IMPVAR option
EXPAND statement, 315

IMPVAR statement
OPTMODEL procedure, 290

IN expression
OPTMODEL expression extensions, 354

IN keyword
index sets, 349

INCLUDE= suboption
LINKS= option, 671
NODES= option, 672

INCLUDE_SELFLINK option
SOLVE WITH NETWORK statement, 670

index sets
IN keyword, 349
index set expression, 354
index-set-item, 349

INIT keyword
NUMBER statement, 292
SET statement, 292
STRING statement, 292
VAR statement, 297

INITPRESOLV= option
DECOMPMASTER statement, 864
DECOMPSUBBPROB statement, 870

INITVAR option
PROC OPTMODEL statement, 285

INTER aggregation expression
OPTMODEL expression extensions, 355

INTER expression
OPTMODEL expression extensions, 354

INTFUZZ= option
PROC OPTMODEL statement, 286

INTTTOL= option
PROC OPTMILP statement, 169

INTTOL= option
PROC OPTMILP statement, 169
SOLVE WITH MILP statement, 613

LB= option
OBJ statement, 43

LEAVE statement
OPTMODEL procedure, 319

LEXICO predicate, 455
LEXICO statement, 29
LINCON statement, 29, 35, 42, 47
LINEARASSIGNMENT option
algorithm options, 677
LINKS= option, 671
LINKS= suboption
OUT= option, 673
SUBGRAPH= option, 674

LOGFREQ= option
DECOMP statement, 861
PROC OPTLP statement, 115
PROC OPTMILP statement, 169, 859
PROC OPTQP statement, 232
SOLVE WITH LP statement, 507
SOLVE WITH LSO statement, 576
SOLVE WITH MILP statement, 613
SOLVE WITH NETWORK statement, 670
SOLVE WITH NLP statement, 786
SOLVE WITH QP statement, 831

LOGLEVEL= option
DECOMP statement, 861
PROC OPTLP statement, 115
PROC OPTMILP statement, 169
PROC OPTQP statement, 232
SOLVE WITH LP statement, 507
SOLVE WITH LSO statement, 576
SOLVE WITH MILP statement, 613
SOLVE WITH NETWORK statement, 670

LOWER= suboption
LINKS= option, 672
NODES= option, 672

LTRIM option
READ DATA statement, 328

MAX aggregation expression
OPTMODEL expression extensions, 355
MAX assignment strategy, 37
MAX statement
OPTMODEL procedure, 291
MAXBLOCKS= option
DECOMP statement, 862
MAXC selection strategy, 37
MAXCLIQUE= suboption
CLIQUE= option, 675
MAXCS selection strategy, 37
MAXCUTS= suboption
MINCUT= option, 677

MAXCYCLES= suboption
CYCLE= option, 676
MAXFUNC= option
SOLVE WITH LSO statement, 575
MAXGEN= option
SOLVE WITH LSO statement, 575
MAXITER= option
DECOMP statement, 862
PROC OPTLP statement, 115
PROC OPTQP statement, 232
SOLVE WITH LP statement, 508
SOLVE WITH NLP statement, 787
SOLVE WITH QP statement, 831

MAXLABELLEN= option
PROC OPTMODEL statement, 286
MAXLEN= option
%MPS2SASD macro parameters, 265
MAXLENGTH= suboption
CYCLE= option, 676
MAXLINKWEIGHT= suboption
CYCLE= option, 676
MAXNODES= option
PROC OPTMILP statement, 169
SOLVE WITH MILP statement, 613
MAXNODES= suboption
TSP= option, 679
MAXNODEWEIGHT= suboption
CYCLE= option, 676
MAXPATHWEIGHT= suboption
SHORTESTPATH= option, 677
MAXPOOLSOLS= option
SOLVE WITH MILP statement, 613
MAXSOLNS= option
PROC CLP statement, 36
SOLVE WITH CLP statement, 448
MAXSOLS= option
PROC OPTMILP statement, 169
SOLVE WITH MILP statement, 614
MAXSOLS= suboption
TSP= option, 679
MAXTIME= option
PROC CLP statement, 36, 52, 53
PROC OPTLP statement, 116
PROC OPTMILP statement, 170
PROC OPTQP statement, 233
SOLVE WITH CLP statement, 448
SOLVE WITH LP statement, 508
SOLVE WITH LSO statement, 575
SOLVE WITH MILP statement, 614
SOLVE WITH NETWORK statement, 671
SOLVE WITH NLP statement, 788
SOLVE WITH QP statement, 831

MAXWEIGHT= suboption
MINCUT= option, 677
METHOD= option
   DECOMP statement, 862
MILP= suboption
   TSP= option, 679
MIN aggregation expression
   OPTMODEL expression extensions, 355
MIN assignment strategy, 37
MIN statement
   OPTMODEL procedure, 291
MINCOSTFLOW option
   algorithm options, 677
MINCUT= option
   algorithm options, 677
MINLENGTH= suboption
   CYCLE= option, 676
MINLINKWEIGHT= suboption
   CYCLE= option, 676
MINNODEWEIGHT= suboption
   CYCLE= option, 676
MINR selection strategy, 29, 37
MINRMAXC selection strategy, 37
MINSPANTREE option
   algorithm options, 677
MISSCHECK option
   PROC OPTMODEL statement, 286
MPSFILE= option
   %MPS2SASD macro parameters, 265
MSGLIMIT= option
   PROC OPTMODEL statement, 286
MSINGULAR= option
   SOLVE WITH NLP statement, 784, 797
BNDRANGE= suboption
   SOLVE WITH NLP statement, 785
MS=() option
   SOLVE WITH NLP statement, 784
MULTISTART=() option
   SOLVE WITH NLP statement, 784
DISTTOL= suboption
   SOLVE WITH NLP statement, 785
LOGLEVEL= suboption
   SOLVE WITH NLP statement, 785
MAXSTARTS= suboption
   SOLVE WITH NLP statement, 786
MAXTIME= suboption
   SOLVE WITH NLP statement, 785
PRINTLEVEL= suboption
   SOLVE WITH NLP statement, 785
NABSFCONV= option
   SOLVE WITH LSO statement, 575
NBLOCKS= option
   DECOMP statement, 862
NDF= suboption
   SOLVE WITH NLP statement, 784
network solver, 665
NGLOBAL= option
   SOLVE WITH LSO statement, 575
Nil constant expression
   OPTMODEL expression extensions, 355
NLOCAL= option
   SOLVE WITH LSO statement, 575
NODES= option
   SOLVE WITH NETWORK statement, 672
NODES= suboption
   OUT= option, 674
SUBGRAPH= option, 674
NODESEL= option
   PROC OPTMILP statement, 172
   SOLVE WITH MILP statement, 616
NINITVAR option
   PROC OPTMODEL statement, 285
NOMISSCHECK option
   PROC OPTMODEL statement, 286
NOOBJECTIVE keyword
   SOLVE statement, 334
NOPREPROCESS
   PROC CLP statement, 36
NOPREPROCESS option
   SOLVE WITH CLP statement, 448
NOTRIM option
   READ DATA statement, 328
NTERMS= option
   SOLVE WITH NLP statement, 784
NTHREADS= option
   DECOMP statement, 863
   DECOMPMASTER statement, 865
   DECOMPMASTERIP statement, 866
   DECOMPSUBPROB statement, 870
   PROC OPTLP statement, 118
   PROC OPTMILP statement, 176
   PROC OPTMODEL statement, 286
   PROC OPTQP statement, 233
   SOLVE WITH LP statement, 510
   SOLVE WITH LSO statement, 576
   SOLVE WITH MILP statement, 621
   SOLVE WITH NETWORK statement, 671
   SOLVE WITH NLP statement, 788
   SOLVE WITH QP statement, 832
null statement
   OPTMODEL procedure, 320
NUMBER statement
   INIT keyword, 292
   OPTMODEL statement, 292
NUMBLOCKS= option
   DECOMP statement, 862
NUMTHREADS= option
   DECOMP statement, 863
   DECOMPMASTER statement, 865
DECOMPMASTERIP statement, 866
DECOMPSUBPROB statement, 870
PROC OPTQP statement, 233

OBJ statement, 43, 47
   LB= option, 43
   TOL= option, 44
   UB= option, 44

OBJECTIVE keyword
   SOLVE statement, 334

OBJECTIVE option
   EXPAND statement, 315

OBJLIMIT= option
   SOLVE WITH NLP statement, 788

OBJSENSE= option
   PROC OPTLP statement, 113
   PROC OPTMILP statement, 167
   PROC OPTQP statement, 233

OBJTOL= option
   SOLVE WITH CLP statement, 448

OF keyword
   function expressions, 348

OMITTED option
   EXPAND statement, 315

OPTLP procedure, 111
   DECOMPMASTER statement, 118
   DECOMP statement, 118
   DECOMPSUBPROB statement, 118

OPTMILP procedure, 165
   DECOMPMASTER statement, 176
   DECOMP statement, 176
   DECOMPSUBPROB statement, 176

OPTMODEL expression extensions
   AND aggregation expression, 351
   CARD function, 351
   CROSS expression, 352
   DIFF expression, 352
   Equality expression, 352
   IF expression, 353
   IN expression, 354
   index set expression, 354
   INTER aggregation expression, 355
   INTER expression, 354
   MAX aggregation expression, 355
   MIN aggregation expression, 355
   Nil constant expression, 355
   OR aggregation expression, 356
   PROD aggregation expression, 356
   range expression, 356
   set constructor expression, 357
   set literal expression, 357
   SETOF aggregation expression, 358
   SLICE expression, 358
   SUM aggregation expression, 359
   SYMDIFF expression, 359
   tuple expression, 360
   UNION aggregation expression, 361
   UNION expression, 360
   WITHIN expression, 361

OPTMODEL Procedure, 280

OPTMODEL procedure
   assignment statement, 298
   CALL statement, 298
   CLOSEFILE statement, 299
   COFOR statement, 299
   CONSTRAINT statement, 289
   CONTINUE statement, 305
   CREATE DATA statement, 305
   DO statement, 311
   DO statement, iterative, 311
   DO UNTIL statement, 312
   DO WHILE statement, 313
   DROP statement, 313
   EXPAND statement, 314
   FILE statement, 316
   FIX statement, 317
   FOR statement, 318
   IF statement, 318
   IMPV AR statement, 290
   LEAVE statement, 319
   MAX statement, 291
   MIN statement, 291
   null statement, 320
   NUMBER statement, 292
   PRINT statement, 320
   PROFILE statement, 324
   PUT statement, 325
   QUIT Statement, 326
   READ DATA statement, 326
   RESET OPTIONS statement, 330
   RESTORE statement, 331
   SAVE MPS statement, 331
   SAVE QPS statement, 332
   SET statement, 292
   SOLVE statement, 334
   STOP statement, 336
   STRING statement, 292
   SUBMIT statement, 336
   UNFIX statement, 339
   USE PROBLEM statement, 339
   VAR statement, 296

OPTMODEL procedure, LP solver
   syntax, 504

OPTMODEL procedure, MILP solver, 609

OPTMODEL procedure, NLP solver
   syntax, 781

OPTMODEL procedure, QP solver
<table>
<thead>
<tr>
<th>Syntax Index</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>syntax, 830</strong></td>
</tr>
<tr>
<td><strong>OPTQP procedure</strong></td>
</tr>
<tr>
<td><strong>syntax, 230</strong></td>
</tr>
<tr>
<td><strong>OPTTOL= option</strong></td>
</tr>
<tr>
<td>PROC OPTLP statement, 116</td>
</tr>
<tr>
<td>PROC OPTMILP statement, 170</td>
</tr>
<tr>
<td>PROC OPTQP statement, 232</td>
</tr>
<tr>
<td>SOLVE WITH LP statement, 508</td>
</tr>
<tr>
<td>SOLVE WITH MILP statement, 614</td>
</tr>
<tr>
<td>SOLVE WITH NLP statement, 788</td>
</tr>
<tr>
<td>SOLVE WITH QP statement, 831</td>
</tr>
<tr>
<td><strong>OR aggregation expression</strong></td>
</tr>
<tr>
<td><strong>OPTMODEL expression extensions, 356</strong></td>
</tr>
<tr>
<td><strong>ORDER= suboption</strong></td>
</tr>
<tr>
<td>OUT= option, 674</td>
</tr>
<tr>
<td><strong>OUT= option</strong></td>
</tr>
<tr>
<td>PROC CLP statement, 29, 36, 49</td>
</tr>
<tr>
<td>SOLVE WITH NETWORK statement, 672</td>
</tr>
<tr>
<td><strong>OUTDATA= option</strong></td>
</tr>
<tr>
<td>%MPS2SASD macro parameters, 265</td>
</tr>
<tr>
<td><strong>PACK predicate, 455</strong></td>
</tr>
<tr>
<td><strong>PACK statement, 29, 44</strong></td>
</tr>
<tr>
<td><strong><em>PAGE</em> keyword</strong></td>
</tr>
<tr>
<td>PRINT statement, 320</td>
</tr>
<tr>
<td>PUT statement, 326</td>
</tr>
<tr>
<td><strong>PARALLELMODE= option</strong></td>
</tr>
<tr>
<td>PROC OPTQP statement, 233</td>
</tr>
<tr>
<td><strong>PARETOMAX= option</strong></td>
</tr>
<tr>
<td>SOLVE WITH LSO statement, 576</td>
</tr>
<tr>
<td><strong>PARTITIONS= suboption</strong></td>
</tr>
<tr>
<td>OUT= option, 674</td>
</tr>
<tr>
<td><strong>PDIGITS= option</strong></td>
</tr>
<tr>
<td>PROC OPTMODEL statement, 287</td>
</tr>
<tr>
<td><strong>PMATRIX= option</strong></td>
</tr>
<tr>
<td>PROC OPTMODEL statement, 287</td>
</tr>
<tr>
<td><strong>POPSIZE= option</strong></td>
</tr>
<tr>
<td>SOLVE WITH LSO statement, 575</td>
</tr>
<tr>
<td><strong>PREPROCESS</strong></td>
</tr>
<tr>
<td>PROC CLP statement, 37</td>
</tr>
<tr>
<td><strong>PREPROCESS option</strong></td>
</tr>
<tr>
<td>SOLVE WITH CLP statement, 448</td>
</tr>
<tr>
<td><strong>PRESOLVER= option</strong></td>
</tr>
<tr>
<td>PROC OPTLP statement, 114</td>
</tr>
<tr>
<td>PROC OPTMILP statement, 168</td>
</tr>
<tr>
<td>PROC OPTMODEL statement, 287</td>
</tr>
<tr>
<td>PROC OPTQP statement, 233</td>
</tr>
<tr>
<td>SOLVE WITH LP statement, 507</td>
</tr>
<tr>
<td>SOLVE WITH MILP statement, 611</td>
</tr>
<tr>
<td>SOLVE WITH QP statement, 832</td>
</tr>
<tr>
<td><strong>PRESTOL= option</strong></td>
</tr>
<tr>
<td>PROC OPTMODEL statement, 287</td>
</tr>
<tr>
<td><strong>PRICETYPE= option</strong></td>
</tr>
<tr>
<td>PROC OPTLP statement, 117</td>
</tr>
<tr>
<td>SOLVE WITH LP statement, 509</td>
</tr>
<tr>
<td><strong>PRIMALIN option</strong></td>
</tr>
<tr>
<td>SOLVE WITH MILP statement, 612</td>
</tr>
<tr>
<td><strong>PRIMALIN= option</strong></td>
</tr>
<tr>
<td>DECOMPMASTERIP statement, 867</td>
</tr>
<tr>
<td>DECOMPSUBPROB statement, 871</td>
</tr>
<tr>
<td>PROC OPTLP statement, 113</td>
</tr>
<tr>
<td>PROC OPTMILP statement, 167</td>
</tr>
<tr>
<td>SOLVE WITH LSO statement, 574</td>
</tr>
<tr>
<td><strong>PRIMALOUT= option</strong></td>
</tr>
<tr>
<td>PROC OPTLP statement, 113</td>
</tr>
<tr>
<td>PROC OPTMILP statement, 167</td>
</tr>
<tr>
<td>PROC OPTQP statement, 234</td>
</tr>
<tr>
<td><strong>PRIMALTOL= option</strong></td>
</tr>
<tr>
<td>PROC OPTQP statement, 234</td>
</tr>
<tr>
<td><strong>PRINT statement</strong></td>
</tr>
<tr>
<td>OPTMODEL procedure, 320</td>
</tr>
<tr>
<td><em>PAGE</em> keyword, 320</td>
</tr>
<tr>
<td><strong>PRINTFREQ= option</strong></td>
</tr>
<tr>
<td>PROC OPTLP statement, 115</td>
</tr>
<tr>
<td>PROC OPTMILP statement, 169, 859</td>
</tr>
<tr>
<td>PROC OPTQP statement, 232</td>
</tr>
<tr>
<td>SOLVE WITH LP statement, 507</td>
</tr>
<tr>
<td>SOLVE WITH MILP statement, 613</td>
</tr>
<tr>
<td>SOLVE WITH NLP statement, 786</td>
</tr>
<tr>
<td>SOLVE WITH QP statement, 831</td>
</tr>
<tr>
<td><strong>PRINTLEVEL2= option</strong></td>
</tr>
<tr>
<td>PROC OPTQP statement, 232</td>
</tr>
<tr>
<td><strong>PRINTLEVEL= option</strong></td>
</tr>
<tr>
<td>PROC OPTLP statement, 116</td>
</tr>
<tr>
<td>PROC OPTMILP statement, 170</td>
</tr>
<tr>
<td>PROC OPTMODEL statement, 288</td>
</tr>
<tr>
<td>PROC OPTQP statement, 234</td>
</tr>
<tr>
<td><strong>PRIORITY= option</strong></td>
</tr>
<tr>
<td>PROC OPTMILP statement, 172</td>
</tr>
<tr>
<td>SOLVE WITH MILP statement, 616</td>
</tr>
<tr>
<td><strong>PROBE= option</strong></td>
</tr>
<tr>
<td>PROC OPTMILP statement, 170</td>
</tr>
<tr>
<td>SOLVE WITH MILP statement, 614</td>
</tr>
<tr>
<td><strong>PROC CLP statement, 35, see TIMETYPE= option,</strong></td>
</tr>
<tr>
<td><strong>see MAXTIME= option</strong></td>
</tr>
<tr>
<td><strong>CONDATA= option, 29, 35, 43, 45, 47, 49</strong></td>
</tr>
<tr>
<td><strong>DECRMAXTIME option, 35</strong></td>
</tr>
<tr>
<td><strong>DOMAIN= option, 35</strong></td>
</tr>
<tr>
<td><strong>EVALUARESEL= option, 35</strong></td>
</tr>
<tr>
<td><strong>FINDALLSOLNS option, 36</strong></td>
</tr>
<tr>
<td><strong>MAXSOLNS= option, 36</strong></td>
</tr>
<tr>
<td><strong>MAXTIME= option, 36, 52, 53</strong></td>
</tr>
<tr>
<td><strong>NOPREPROCESS, 36</strong></td>
</tr>
<tr>
<td>OUT= option, 29, 36, 49</td>
</tr>
<tr>
<td><strong>PREPROCESS, 37</strong></td>
</tr>
<tr>
<td><strong>SHOWPROGRESS option, 37</strong></td>
</tr>
<tr>
<td><strong>TIMETYPE= option, 37</strong></td>
</tr>
</tbody>
</table>
USECONDATA VARS= option, 37
VARASSIGN= option, 29, 37
VARESELECT= option, 29, 37

PROC OPTLP statement
  ALGORITHM2= option, 114
  ALGORITHM= option, 113
  BASIS= option, 116
  CROSEROVER= option, 117
  DATA= option, 112
  DETERMINISTIC= option, 118
  DUALIN= option, 112
  DUALITYGAP= option, 118
  DUALIZE= option, 114
  DUALOUT= option, 112
  FEASTOL= option, 115
  IIS= option, 114
  LOGFREQ= option, 115
  LOGLEVEL= option, 116
  MAXITER= option, 115
  MAXTIME= option, 116
  NTHREADS= option, 118
  OBJSENSE= option, 113
  OPTTOL= option, 116
  PRESOLVER= option, 114
  PRICETYPE= option, 117
  PRIMALIN= option, 113
  PRIMALOUT= option, 113
  PRINTFREQ= option, 115
  PRINTLEVEL= option, 116
  QUEUESIZE= option, 117
  SAVE_ONLY_IF_OPTIMAL option, 113
  SCALE= option, 117
  SEED= option, 117
  SOL= option, 113
  SOLVER2= option, 114
  SOLVER= option, 113
  TIMETYPE= option, 116

PROC OPTMILP statement
  ABSOBJGAP= option, 168
  CONCURRENT= option, 175
  CONFLICTSEARCH= option, 172
  CUTCLIQUE= option, 174
  CUTFLOWCOVER= option, 174
  CUTFLOWPATH= option, 174
  CUTGOMORY= option, 174
  CUTGUB= option, 174
  CUTIMPLIED= option, 175
  CUTFKNAPOSACK= option, 175
  CUTFLOW= option, 175
  CUTMILIFTED= option, 175
  CUTMIR= option, 175
  CUTMULTICOMMODITY= option, 175
  CUTOFF= option, 168
  CUTS= option, 175

PROC OPTMODEL statement

PROC OPTQP statement
  DATA= option, 231
  DUALITYGAP= option, 231
  DUALOUT= option, 232
  DUALTOL= option, 232
  FEASTOL= option, 234
  IIS= option, 232
  LOGFREQ= option, 232
  LOGLEVEL= option, 232
  MAXITER= option, 232
  MAXTIME= option, 233
  NTHREADS= option, 233
  NUMTHREADS= option, 233
  OBJSENSE= option, 233
  OPTTOL= option, 232
  PARALLELMODE= option, 233
PRESOLVER= option, 233
PRIMALOUT= option, 234
PRIMALTOL= option, 234
PRINTFREQ= option, 232
PRINTLEVEL2= option, 232
PRINTLEVEL= option, 234
TIMETYPE= option, 234

PROD aggregation expression
OPTMODEL expression extensions, 356

PUT statement
_PAGE_ keyword, 326

PRINTWIDTH= option
PROC OPTMODEL statement, 288

QUEUESIZE= option
PROC OPTLP statement, 117
SOLVE WITH LP statement, 509

QUIT Statement
OPTMODEL procedure, 326

range expression
OPTMODEL expression extensions, 356

READ DATA statement
COL keyword, 329
INTO keyword, 326
LTRIM option, 328
NOTRIM option, 328
OPTMODEL procedure, 326
RTRIM option, 328
TRIM option, 328

REIFY predicate, 456
REIFY statement, 29, 45

RELAXINT keyword
SOLVE statement, 334

RELOBJGAP= option
DECOMP statement, 863
PROC OPTMILP statement, 170
SOLVE WITH MILP statement, 614

RELOBJGAP= suboption
TSP= option, 680

RESET OPTIONS statement
OPTMODEL procedure, 330

RESTARTS= option
PROC OPTMILP statement, 172
SOLVE WITH MILP statement, 616

RESTORE statement
OPTMODEL procedure, 331

RTRIM option
READ DATA statement, 328

SAVE MPS statement
OPTMODEL procedure, 331

SAVE QPS statement
OPTMODEL procedure, 332

SAVE_ONLY_IF_OPTIMAL option
PROC OPTLP statement, 113

SCALE= option
PROC OPTLP statement, 117
PROC OPTMILP statement, 171
SOLVE WITH LP statement, 509
SOLVE WITH MILP statement, 614

SEED= option
PROC OPTLP statement, 117
PROC OPTMILP statement, 171
SOLVE WITH LP statement, 509
SOLVE WITH LSO statement, 576
SOLVE WITH MILP statement, 614
SOLVE WITH NLP statement, 784

SESSREF= option
PROC OPTMODEL statement, 288

SESSUUID= option
PROC OPTMODEL statement, 288

set constructor expression
OPTMODEL expression extensions, 357

set literal expression
OPTMODEL expression extensions, 357

SET statement
INIT keyword, 292
OPTMODEL procedure, 292

SETOF aggregation expression
OPTMODEL expression extensions, 358

SHORTESTPATH= option
algorithm options, 677

SHOWPROGRESS option
PROC CLP statement, 37
SOLVE WITH CLP statement, 448

SIGSQ= option
SOLVE WITH NLP statement, 784, 798

SINK= suboption
SHORTESTPATH= option, 677

SLICE expression
OPTMODEL expression extensions, 358

SOL= option
DECOMPSUBPROB statement, 870
PROC OPTLP statement, 113
SOLVE WITH LP statement, 506

SOLTYPE= option
SOLVE WITH NLP statement, 784, 787

SOLVE option
EXPAND statement, 314

SOLVE statement
NOOBJECTIVE keyword, 334
OBJECTIVE keyword, 334
OPTMODEL procedure, 334
RELAXINT keyword, 334
VARASSIGN= option, 459
VARESELECT= option, 459
WITH keyword, 334

SOLVE WITH CLP statement
statement options, 448
SOLVE WITH LP statement
  ALGORITHM2= option, 506
  ALGORITHM= option, 506
  BASIS= option, 508
  CROSSOVER= option, 509
  DECOMPMaster=() option, 510
  DECOMP=() option, 510
  DECOMPSubprob=() option, 510
  DETERMINISTIC= option, 510
  DualityGAP= option, 510
  Dualize= option, 506
  FEASTOL= option, 507
  IIS= option, 506
  LOGFREQ= option, 507
  LOGLEVEL= option, 507
  MAXITER= option, 508
  MAXTIME= option, 508
  NTHREADS= option, 508
  OPTTOL= option, 508
  PRESolver= option, 507
  PRICETYPE= option, 509
  PRINTFREQ= option, 507
  QUEUESIZE= option, 509
  SCALE= option, 509
  SEED= option, 509
  SOL= option, 506
  SOLVER2= option, 506
  SOLVER= option, 506
  TIMETYPE= option, 508
SOLVE WITH LSO statement
  ABSFCONV= option, 575
  CACHEMAX= option, 576
  CACHETOL= option, 576
  FEASTOL= option, 576
  input data set options, 574
  LOGFREQ= option, 576
  LOGLEVEL= option, 576
  MAXFUNC= option, 575
  MAXTIME= option, 575
  NABSFCONV= option, 575
  NGLOBAL= option, 575
  NLOCAL= option, 575
  NTHREADS= option, 576
  optimization control options, 575
  ParetoMAX= option, 576
  POPSIZe= option, 575
  PRIMALIN= option, 574
  SEED= option, 576
  stopping condition options, 575
  technical options, 576
SOLVE WITH MILP statement
  ABSOBJGAP= option, 612
  CONCURRENT= option, 620
  CONFLICTSEARCH= option, 615
  CUTCLIQUE= option, 618
  CUTFLOWCOVER= option, 618
  CUTFLOWPATH= option, 618
  CUTGOMORY= option, 618
  CUTGUB= option, 618
  CUTILIMPLIED= option, 618
  CUTKNPACK= option, 619
  CUTCUTLAP= option, 619
  CUTMLILIFTED= option, 619
  CUTMRIR= option, 619
  CUTMULTICOMMODITY= option, 619
  CUTOFF= option, 612
  CUTS= option, 619
  CUTSFAC= option, 619
  CUTSTRATEGY= option, 619
  CUTZEROHALF= option, 619
  DECOMPMasterIP=() option, 620
  DECOMPMaster=() option, 620
  DECOMP=() option, 619
  DECOMPSUBprob=() option, 620
  DETERMINISTIC= option, 620
  DISTRIBUTED= option, 620
  EMPHASIS= option, 612
  FEASTOL= option, 612
  HEURISTICS= option, 615
  INTTOL= option, 613
  LOGFREQ= option, 613
  LOGLEVEL= option, 613
  MAXNODES= option, 613
  MAXPOOLSOLS= option, 613
  MAXSOLS= option, 614
  MAXTIME= option, 614
  NODESEL= option, 616
  OPTTOL= option, 614
  PRESolver= option, 611
  PRIMALIN option, 612
  PRINTFREQ= option, 613
  PRIORITY= option, 616
  PROBE= option, 614
  RELOBJGAP= option, 614
  RESTARTS= option, 616
  SCALE= option, 614
  SEED= option, 614
  STRONGITER= option, 616
  STRONGLEN= option, 617
  SYMMETRY= option, 617
  TARGET= option, 615
  TIMETYPE= option, 615
  VARSEL= option, 617
SOLVE WITH NETWORK statement
  statement options, 670
SOLVE WITH NLP statement
  ALGORITHM= option, 786
  ASINGULAR= option, 782
  COV= option, 782
  COVEST=() option, 782
  COVOUT= option, 783
  COVSING= option, 783
  FEASTOL= option, 787
  IIS= option, 787
  LOGFREQ= option, 786
  MAXITER= option, 787
  MAXTIME= option, 788
  MSINGULAR= option, 784
  BNDRANGE= suboption, 785
  MS=() option, 784
  MULTISTART=() option, 784
  DISTTOL= suboption, 785
  LOGLEVEL= suboption, 785
  MAXSTARTS= suboption, 786
  MAXTIME= suboption, 785
  PRINTLEVEL= suboption, 785
  NDF= suboption, 784
  NTERMS= option, 784
  NTHREADS= option, 788
  OBJLIMIT= option, 788
  OPTTOL= option, 788
  PRINTFREQ= option, 786
  SEED= option, 784
  SIGSQ= option, 784
  SOLTYPE= option, 786, 787
  SOLVER= option, 786
  TECH= option, 786
  TECHNIQUE= option, 786
  TIMETYPE= option, 788
  VARDEF= option, 784

SOLVE WITH QP statement
  DUALITYGAP= option, 831
  DUALTOL= option, 831
  FEASTOL= option, 832
  IIS= option, 831
  LOGFREQ= option, 831
  MAXITER= option, 831
  MAXTIME= option, 831
  NTHREADS= option, 832
  OPTTOL= option, 831
  PRESOLVER= option, 832
  PRIMALTOL= option, 832
  PRINTFREQ= option, 831
  TIMETYPE= option, 832

SOLVE WITH NETWORK statement
  SOURCE= suboption
  SHORTESTPATH= option, 678
  SPPATHS= suboption
  OUT= option, 674
  SPWEIGHTS= suboption
  OUT= option, 674
  STOP statement
  OPTMODEL procedure, 336

STRING statement
  INIT keyword, 292
  OPTMODEL procedure, 292

STRONGITER= option
  PROC OPTMILP statement, 173, 859
  SOLVE WITH MILP statement, 616

STRONGLEN= option
  PROC OPTMILP statement, 173
  SOLVE WITH MILP statement, 617

SUBGRAPH= option
  SOLVE WITH NETWORK statement, 674

SUBMIT statement
  OPTMODEL procedure, 336

SUM aggregation expression
  OPTMODEL expression extensions, 359

SYMDIFF expression
  OPTMODEL expression extensions, 359

SYMMETRY= option
  PROC OPTMILP statement, 173
  SOLVE WITH MILP statement, 617

TARGET= option
  PROC OPTMILP statement, 171
  SOLVE WITH MILP statement, 615

TARGET= suboption
  TSP= option, 680

TECH= option
  SOLVE WITH NLP statement, 786

THEN keyword
  IF statement, 318

TIMETYPE= option
  PROC CLP statement, 37
  PROC OPTLP statement, 116
  PROC OPTMILP statement, 171
  PROC OPTQP statement, 234
  SOLVE WITH CLP statement, 448
  SOLVE WITH LP statement, 508
  SOLVE WITH MILP statement, 615
  SOLVE WITH NETWORK statement, 671
  SOLVE WITH NLP statement, 788
  SOLVE WITH QP statement, 832
TOL= option
   OBJ statement, 44

TOUR= suboption
   OUT= option, 674

TRANSITIVECLOSURE option
   algorithm options, 678

TRIM option
   READ DATA statement, 328

TSP= option
   algorithm options, 678

tuple expression
   OPTMODEL expression extensions, 360

UB= option
   OBJ statement, 44

UNFIX statement
   OPTMODEL procedure, 339

UNION aggregation expression
   OPTMODEL expression extensions, 361

UNION expression
   OPTMODEL expression extensions, 360

UNTIL keyword
   DO statement, iterative, 311

UPPER= suboption
   LINKS= option, 672
   NODES= option, 672

USE PROBLEM statement
   OPTMODEL procedure, 339

USECONDATA V ARS= option
   PROC CLP statement, 37

V AR option
   EXPAND statement, 315

V AR statement
   INIT keyword, 297
   OPTMODEL procedure, 296

V ARASSIGN= option
   PROC CLP statement, 29, 37
   SOLVE statement, 459
   SOLVE WITH CLP statement, 448

V ARDEF= option
   SOLVE WITH NLP statement, 784, 795

V ARFUZZ= option
   PROC OPTMODEL statement, 288

V ARIABLE statement, 29, 35, 37, 38, 43, 45–47

V ARSEL= option
   PROC OPTMILP statement, 173, 859
   SOLVE WITH MILP statement, 617

V ARSELECT= option
   PROC CLP statement, 29, 37
   SOLVE statement, 459
   SOLVE WITH CLP statement, 449

WDEG selection strategy, 38

WEIGHT= suboption
   LINKS= option, 672
   NODES= option, 672

WHILE keyword
   DO statement, iterative, 311

WITH keyword
   SOLVE statement, 334

WITHIN expression
   OPTMODEL expression extensions, 361
Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore for additional books and resources.