SAS® Optimization 8.2 Network Optimization Procedures

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2017. *SAS® Optimization 8.2: Network Optimization Procedures*. Cary, NC: SAS Institute Inc.

SAS® Optimization 8.2: Network Optimization Procedures

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

June 2018

Contents

Chapter 1. What's New in SAS Optimization 8.2 1
Chapter 2. Introduction 5
Chapter 3. The OPTNETWORK Procedure 11

Index **147**

Credits

Documentation

Writing	Matthew Galati
Editing	Anne Baxter, Ed Huddleston
Documentation Support	Tim Arnold
Technical Review	Charles B. Kelly, Ed Hughes, Rob Pratt

Software

PROC OPTNETWORK	Matthew Galati

Support Groups

Software Testing	Charles B. Kelly, Rui Kang, Yu-Min Lin, Elham Sadeghi
Technical Support	Tonya Chapman

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.

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

Overview of the OPTNETWORK Procedure **5**
About This Book **5**
Chapter Organization 5
Using CAS Sessions and CAS Engine Librefs 6
Loading a SAS Data Set onto a CAS Server 7
Typographical Conventions 8
Options Used in Examples 8
Where to Turn for More Information **9**
Online Documentation 9
SAS Technical Support Services 9

Overview of the OPTNETWORK Procedure

The OPTNETWORK procedure includes a number of graph theory and network optimization algorithms that can augment more generic mathematical optimization approaches. Many practical applications of optimization depend on an underlying network. For example, retailers face the problem of shipping goods from warehouses to stores in a distribution network to satisfy demand at minimum cost. Commuters choose routes in a road network to travel from home to work in the shortest amount of time.

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 is organized as follows:

Chapter 2, this chapter, provides an overview of the OPTNETWORK procedure, describes typographical conventions, and tells you where you can find more information.

Chapter 3 describes the OPTNETWORK procedure and is organized as follows:

- The "Overview" section briefly describes the analysis provided by the procedure.
- The "Getting Started" section provides a quick introduction to the procedure through a simple example.
- The "Syntax" section describes the SAS statements and options that control the procedure.
- The "Details" section discusses methodology and other topics, such as ODS tables.
- The "Examples" section contains examples that use the procedure.
- The "References" section contains references for the methodology.

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:

```
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:

  ```
  data mycas.Sample;
     input from $ to $ @@;
     datalines;
  A B A C B C
  ;
  ```

  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:

  ```
  data Sample;
     input from $ to $ @@;
     datalines;
  A B A C B C
  ;
  data mycas.Sample;
     set Sample;
  run;
  ```

- You can use the CASUTIL procedure as follows:

  ```
  proc casutil sessref=mysess;
     load data=Sample casout="Sample";
  quit;
  ```

  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 OPTNETWORK procedure statements use a data table that resides in the mycas caslib:

```
proc optnetwork links = mycas.Sample;
  ...statements...;
run;
```

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

```
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*.

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:

```
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:

```
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`.

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` for more information.

Chapter 3
The OPTNETWORK Procedure

Contents

Overview: OPTNETWORK Procedure **12**
Getting Started: OPTNETWORK Procedure **13**
Road Network Shortest Path 13
Syntax: OPTNETWORK Procedure **17**
Functional Summary 18
PROC OPTNETWORK Statement 23
BICONNECTEDCOMPONENTS Statement 26
BY Statement 26
CLIQUE Statement 26
CONNECTEDCOMPONENTS Statement 27
CYCLE Statement 28
DISPLAY Statement 29
DISPLAYOUT Statement 30
LINEARASSIGNMENT Statement 31
LINKSVAR Statement 31
MINCOSTFLOW Statement 32
MINCUT Statement 33
MINSPANTREE Statement 33
NODESSUBSETVAR Statement 34
NODESVAR Statement 34
PATH Statement 35
SHORTESTPATH Statement 36
SUMMARY Statement 37
TRANSITIVECLOSURE Statement 38
TSP Statement 38
Details: OPTNETWORK Procedure **41**
Graph Input Data 41
Execution Modes and Data Movement 48
Numeric Limitations 50
Size Limitations 51
Biconnected Components and Articulation Points 51
Clique Enumeration 54
Connected Components 58
Cycle Enumeration 62
Linear Assignment (Matching) 67
Minimum-Cost Network Flow 68

Minimum Cut 75
Minimum Spanning Tree 79
Path Enumeration 81
Shortest Path 85
Summary Statistics 96
Transitive Closure 103
Traveling Salesman Problem 105
Macro Variable _OROPTNETWORK_ 112
ODS Table Names 116
Examples: OPTNETWORK Procedure **117**
Example 3.1: Articulation Points in a Terrorist Network 117
Example 3.2: Cycle Enumeration for Kidney Donor Exchange 119
Example 3.3: Linear Assignment Problem for Minimizing Relay Times 125
Example 3.4: Minimum Spanning Tree for Computer Network Design 127
Example 3.5: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System 129
Example 3.6: Traveling Salesman Tour of US Capital Cities 132
Example 3.7: Connected Components for US Patent Citations 138
Example 3.8: Shortest Paths of the New York Road Network 140
Example 3.9: Shortest Path in a Road Network by Date and Time 141
References **145**

Overview: OPTNETWORK Procedure

The OPTNETWORK procedure includes a number of graph theory and network optimization algorithms that can augment more generic mathematical optimization approaches. Many practical applications of optimization depend on an underlying network. For example, retailers face the problem of shipping goods from warehouses to stores in a distribution network to satisfy demand at minimum cost. Commuters choose routes in a road network to travel from home to work in the shortest amount of time.

Networks also appear explicitly and implicitly in many other application contexts. Networks are often constructed from certain natural co-occurrence types of relationships—such as relationships among researchers who coauthor articles, actors who appear in the same movie, words or topics that occur in the same document, items that appear together in a shopping basket, terrorism suspects who travel together or are seen in the same location, and so on. In these types of relationship, the strength or frequency of interaction is modeled as weights on the links of the resulting network.

Although you can solve many network problems by using more general methods, such as linear programming or mixed integer linear programming, the special-purpose methods that the OPTNETWORK procedure implements require less user code and offer performance improvements of several orders of magnitude.

To support the myriad ways in which networks appear in optimization, the OPTNETWORK procedure makes no assumptions about the context or application from which the network arises. It provides a number of network analysis and optimization algorithms (listed in Table 3.1) that take an abstract graph or network

as input and help explain network structure and solve network optimization problems. Depending on the application, this type of network analysis can stand on its own and provide independent value, or it can provide input for subsequent work in optimization or other forms of analytics.

Table 3.1 Algorithm Classes in PROC OPTNETWORK

Algorithm Class	PROC OPTNETWORK Statement
Biconnected components	BICONNECTEDCOMPONENTS
Clique enumeration	CLIQUE
Connected components	CONNECTEDCOMPONENTS
Cycle enumeration	CYCLE
Weighted matching	LINEARASSIGNMENT
Minimum-cost network flow	MINCOSTFLOW
Minimum cut	MINCUT
Minimum spanning tree	MINSPANTREE
Path enumeration	PATH
Shortest path	SHORTESTPATH
Graph summary	SUMMARY
Transitive closure	TRANSITIVECLOSURE
Traveling salesman	TSP

As input, the OPTNETWORK procedure expects *graph* $G = (N, A)$, which is defined over a set N of nodes and a set A of arcs. A *node* is an abstract representation of some entity (or object), and an *arc* defines the relationship (or connection) between two nodes. The terms *node* and *vertex* are interchangeable in describing an entity. The term *arc* is interchangeable with the term *edge* or *link* in describing a connection. Similarly, the terms *graph* and *network* are interchangeable.

You can also access these network algorithms via the network solver in PROC OPTMODEL. For more information, see Chapter 15, "The Network Solver" (*SAS Optimization: Mathematical Optimization Procedures*).

Getting Started: OPTNETWORK Procedure

Because graphs are abstract objects, their analyses have applications in many different fields of study, including social sciences, linguistics, biology, transportation, marketing, and so on. This chapter demonstrates a few potential applications through simple examples.

This section presents an introductory example for getting started with the OPTNETWORK procedure. For more information about the expected input formats and the available algorithms, see the sections "Details: OPTNETWORK Procedure" on page 41 and "Examples: OPTNETWORK Procedure" on page 117.

Road Network Shortest Path

Consider the following road network between a SAS employee's home in Raleigh, North Carolina, and SAS headquarters nearby in Cary. In this road network (graph), the links are the roads and the nodes are

intersections of the roads. For each road, you assign a *link attribute* in the variable 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), which gives an approximate number of minutes to travel between two nodes based on the length of the road and the typical speed during normal traffic patterns. These statements assume that the CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```
data mycas.LinkSetInRoadNC10am;
   input start_inter $1-20 end_inter $21-40 miles miles_per_hour;
   time_to_travel = miles * 1/miles_per_hour * 60;
   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
;
```

Using PROC OPTNETWORK, you want to find the route that yields the shortest path between home (614 Capital Boulevard) and SAS headquarters (SAS Campus Drive). This can be done using the SHORTESTPATH statement as follows:

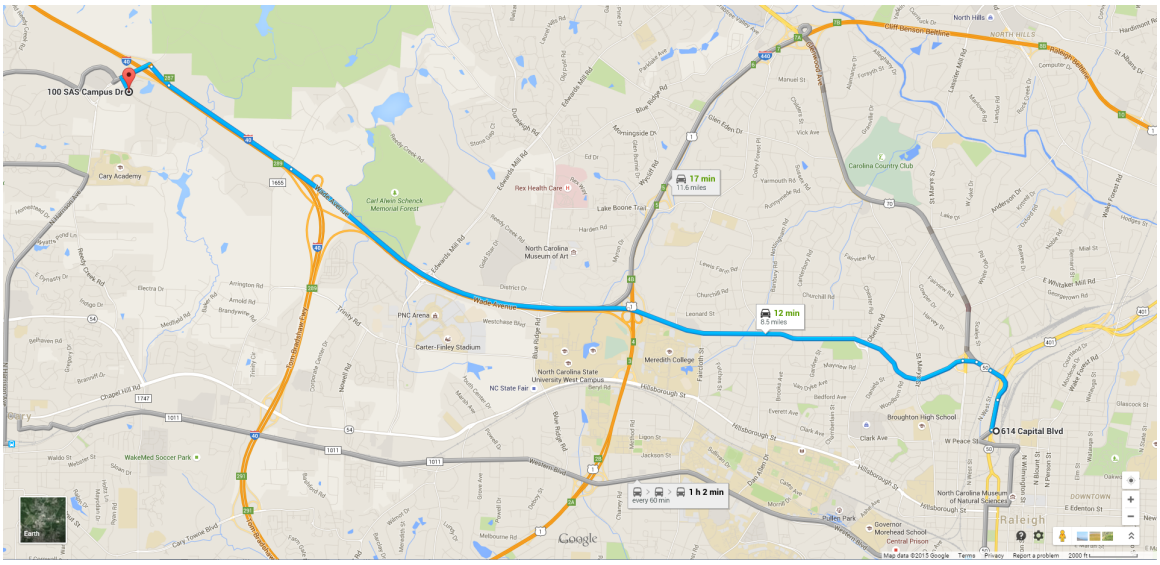
```
proc optnetwork
   links       = mycas.LinkSetInRoadNC10am;
   linksVar
      from     = start_inter
      to       = end_inter
      weight   = time_to_travel;
   shortestPath
      outPaths = mycas.ShortPath
      source   = "614CapitalBlvd"
      sink     = "SASCampusDrive";
run;
```

For more information about shortest path algorithms in PROC OPTNETWORK, see the section "Shortest Path" on page 85. Figure 3.1 displays the output data table mycas.ShortPath, which shows the best route to take to minimize travel time at 10:00 a.m. on a workday. This route is also shown in Google Maps in Figure 3.2.

Figure 3.1 Shortest Path for Road Network at 10:00 A.M.

order	start_inter	end_inter	time_to_travel
1	614CapitalBlvd	Capital/WadeAve	1.4400
2	Capital/WadeAve	WadeAve/RaleighExpy	4.5000
3	WadeAve/RaleighExpy	RaleighExpy/US40W	3.0000
4	RaleighExpy/US40W	US40W/HarrisonAve	1.4182
5	US40W/HarrisonAve	SASCampusDrive	1.2000
			11.5582

Figure 3.2 Shortest Path for Road Network at 10:00 A.M. in Google Maps



Now suppose that it is the evening rush hour (5:00–7:00 p.m.) and the time that it takes to travel this route 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 because of rush-hour traffic. The following data table lists approximate travel times and speeds for driving in the opposite direction:

```
data mycas.LinkSetInRoadNC5pm;
   input start_inter $1-20 end_inter $21-40 miles miles_per_hour;
   time_to_travel = miles * 1/miles_per_hour * 60;
   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  25 /*high traffic*/
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
;
```

The following statements are similar to those in the first PROC OPTNETWORK run, except that they use the data table mycas.LinkSetInRoadNC5pm and the SOURCE= and SINK= option values are reversed:

```
proc optnetwork
   links        = mycas.LinkSetInRoadNC5pm;
   linksVar
      from      = start_inter
      to        = end_inter
      weight    = time_to_travel;
   shortestPath
      outPaths  = mycas.ShortPath
      source    = "SASCampusDrive"
      sink      = "614CapitalBlvd";
run;
```

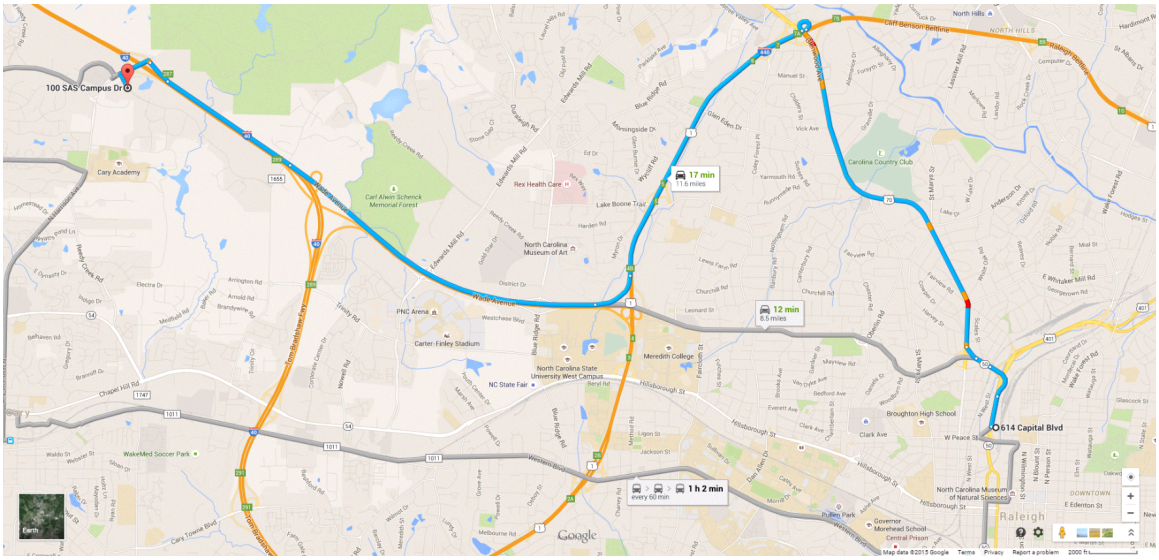
Now, the output data table mycas.ShortPath, shown in Figure 3.3, shows the best route for going home. Because the traffic on Wade Avenue is usually heavy at this time of day, the best route home is different from the best route to work.

Figure 3.3 Shortest Path for Road Network at 5:00 P.M.

order	start_inter	end_inter	time_to_travel
1	SASCampusDrive	US40W/HarrisonAve	1.2000
2	US40W/HarrisonAve	RaleighExpy/US40W	1.4182
3	RaleighExpy/US40W	US440W/RaleighExpy	3.0000
4	US440W/RaleighExpy	US70W/US440W	2.7000
5	US70W/US440W	Capital/US70W	3.2000
6	Capital/US70W	614CapitalBlvd	1.4400
			12.9582

This new route is shown in Google Maps in Figure 3.4.

Figure 3.4 Shortest Path for Road Network at 5:00 P.M. in Google Maps



Syntax: OPTNETWORK Procedure

PROC OPTNETWORK statements are divided into four main categories:

PROC Statement

PROC OPTNETWORK < *options* > ;

The PROC statement invokes the procedure and sets option values that are used across multiple algorithms.

Data Input Statements

LINKSVAR < *options* > ;
NODESVAR < *options* > ;
NODESSUBSETVAR < *options* > ;

Data input statements control the names of the variables that PROC OPTNETWORK expects in the data input.

Algorithm Statements

BICONNECTEDCOMPONENTS ;
CLIQUE < *options* > ;
CONNECTEDCOMPONENTS < *options* > ;
CYCLE < *options* > ;
LINEARASSIGNMENT < *options* > ;
MINCOSTFLOW < *options* > ;
MINCUT < *options* > ;
MINSPANTREE < *options* > ;
PATH < *options* > ;
SHORTESTPATH < *options* > ;
SUMMARY < *options* > ;
TRANSITIVECLOSURE < *options* > ;
TSP < *options* > ;

Algorithm statements determine which algorithm is run and set options for each individual algorithm.

Standard Statements

BY *variables* ;
DISPLAY < *table-list* > < / *options* > ;
DISPLAYOUT *table-spec-list* < / *options* > ;

Standard statements control BY-group processing and manage ODS tables.

The following section provides a quick summary of each statement and its options. Each statement is then described in more detail in its own section. The PROC OPTNETWORK statement is described first, and sections that describe all the other statements are presented in alphabetical order (they are not ordered according to their category).

Functional Summary

Table 3.2 summarizes the statements and options available in the OPTNETWORK procedure.

Table 3.2 Functional Summary of Statements and Options

Description	Statement	Option
Input		
Specifies the links data table	PROC OPTNETWORK	LINKS=
Specifies the nodes data table	PROC OPTNETWORK	NODES=
Specifies the nodes subset data table	PROC OPTNETWORK	NODESSUBSET=
Output		
Specifies the links output data table	PROC OPTNETWORK	OUTLINKS=
Specifies the nodes output data table	PROC OPTNETWORK	OUTNODES=
Options		
Specifies the graph direction	PROC OPTNETWORK	DIRECTION=
Includes self-links	PROC OPTNETWORK	INCLUDESELFLINK
Specifies the index offset for identifiers	PROC OPTNETWORK	INDEXOFFSET=
Specifies the desired frequency (in number of seconds) between log entries	PROC OPTNETWORK	LOGFREQTIME=
Specifies the overall log level	PROC OPTNETWORK	LOGLEVEL=
Specifies the maximum number of threads to use for multithreaded processing	PROC OPTNETWORK	NTHREADS=
Specifies that the input graph data are in a standardized format	PROC OPTNETWORK	STANDARDIZEDLABELS
Specifies whether time units are in CPU time or real time	PROC OPTNETWORK	TIMETYPE=
Data Input Statements		
Specifies the data variable name for the auxiliary link weights	LINKSVAR	AUXWEIGHT=
Specifies the data variable name for the *from* nodes	LINKSVAR	FROM=
Specifies the data variable name for the link lower bounds	LINKSVAR	LOWER=
Specifies the data variable name for the *to* nodes	LINKSVAR	TO=
Specifies the data variable name for the link upper bounds	LINKSVAR	UPPER=
Specifies the data variable name for the link weights	LINKSVAR	WEIGHT=
Specifies the data variable name for the node lower bounds	NODESVAR	LOWER=
Specifies the data variable name for the nodes	NODESVAR	NODE=

Table 3.2 (continued)

Description	Statement	Option
Specifies the data variable name for the node upper bounds	NODESVAR	UPPER=
Specifies the data variable name for the node weights	NODESVAR	WEIGHT=
Specifies the data variable name for the nodes	NODESSUBSETVAR	NODE=
Specifies the data variable name for the sink indicator	NODESSUBSETVAR	SINK=
Specifies the data variable name for the source indicator	NODESSUBSETVAR	SOURCE=
Algorithm Statements		
CLIQUE Statement		
Specifies the maximum number of cliques to return during clique enumeration	CLIQUE	MAXCLIQUES=
Specifies the maximum amount of time to spend finding cliques	CLIQUE	MAXTIME=
Specifies the output data table for cliques	CLIQUE	OUT=
CONNECTEDCOMPONENTS Statement		
Specifies the algorithm to use for connected components	CONNECTEDCOMPONENTS	ALGORITHM=
Specifies the internal graph format	CONNECTEDCOMPONENTS	INTERNALFORMAT=
CYCLE Statement		
Specifies the algorithm to use for cycle enumeration	CYCLE	ALGORITHM=
Specifies the maximum number of cycles to return during cycle enumeration	CYCLE	MAXCYCLES=
Specifies the maximum length for the cycles found	CYCLE	MAXLENGTH=
Specifies the maximum link weight for the cycles found	CYCLE	MAXLINKWEIGHT=
Specifies the maximum node weight for the cycles found	CYCLE	MAXNODEWEIGHT=
Specifies the maximum amount of time to spend finding cycles	CYCLE	MAXTIME=
Specifies the minimum length for the cycles found	CYCLE	MINLENGTH=
Specifies the minimum link weight for the cycles found	CYCLE	MINLINKWEIGHT=
Specifies the minimum node weight for the cycles found	CYCLE	MINNODEWEIGHT=

Table 3.2 (continued)

Description	Statement	Option
Specifies the output data table for cycles	CYCLE	OUT=
LINEARASSIGNMENT Statement		
Specifies the output data table for linear assignment	LINEARASSIGNMENT	OUT=
MINCOSTFLOW Statement		
Specifies the internal graph format	MINCOSTFLOW	INTERNALFORMAT=
Specifies the iteration log frequency	MINCOSTFLOW	LOGFREQ=
Specifies the maximum amount of time to spend calculating minimum-cost network flows	MINCOSTFLOW	MAXTIME=
MINCUT Statement		
Specifies the maximum number of cuts to return	MINCUT	MAXCUTS=
Specifies the maximum weight of the cuts to return	MINCUT	MAXWEIGHT=
Specifies the output data table for minimum cut sets	MINCUT	OUTCUTSETS=
Specifies the output data table for minimum cut partitions	MINCUT	OUTPARTITIONS=
MINSPANTREE Statement		
Specifies the internal graph format	MINSPANTREE	INTERNALFORMAT=
Specifies the output data table for a minimum spanning tree	MINSPANTREE	OUT=
PATH Statement		
Specifies the internal graph format	PATH	INTERNALFORMAT=
Specifies the maximum length for the paths found	PATH	MAXLENGTH=
Specifies the maximum link weight for the paths found	PATH	MAXLINKWEIGHT=
Specifies the maximum node weight for the paths found	PATH	MAXNODEWEIGHT=
Specifies the maximum amount of time to spend finding paths	PATH	MAXTIME=
Specifies the minimum length for the paths found	PATH	MINLENGTH=
Specifies the minimum link weight for the paths found	PATH	MINLINKWEIGHT=

Table 3.2 (continued)

Description	Statement	Option
Specifies the minimum node weight for the paths found	PATH	MINNODEWEIGHT=
Specifies the output data table for path links	PATH	OUTPATHSLINKS=
Specifies the output data table for path nodes	PATH	OUTPATHSNODES=
Specifies the sink node for path calculations	PATH	SINK=
Specifies the source node for path calculations	PATH	SOURCE=
SHORTESTPATH Statement		
Specifies the maximum path weight	SHORTESTPATH	MAXPATHWEIGHT=
Specifies the output data table for shortest paths	SHORTESTPATH	OUTPATHS=
Specifies the output data table for shortest path summaries	SHORTESTPATH	OUTWEIGHTS=
Specifies the sink node for shortest path calculations	SHORTESTPATH	SINK=
Specifies the source node for shortest path calculations	SHORTESTPATH	SOURCE=
SUMMARY Statement		
Calculates information about biconnected components	SUMMARY	BICONNECTEDCOMPONENTS
Calculates information about connected components	SUMMARY	CONNECTEDCOMPONENTS
Calculates the approximate diameter and chooses the weight type	SUMMARY	DIAMETERAPPROX=
Specifies the output data table for summary results	SUMMARY	OUT=
Calculates information about shortest paths and chooses the weight type	SUMMARY	SHORTESTPATH=
TRANSITIVECLOSURE Statement		
Specifies the output data table for transitive closure results	TRANSITIVECLOSURE	OUT=
TSP Statement		
Specifies the stopping criterion based on the absolute objective gap	TSP	ABSOBJGAP=
Specifies the cutoff value for branch-and-bound node removal	TSP	CUTOFF=

Table 3.2 (continued)

Description	Statement	Option
Specifies the overall cut strategy level	TSP	CUTSTRATEGY=
Specifies the initial and primal heuristics level	TSP	HEURISTICS=
Specifies the frequency of printing the branch-and-bound node log	TSP	LOGFREQ=
Specifies the maximum number of branch-and-bound nodes to be processed	TSP	MAXNODES=
Specifies the maximum number of solutions to be found	TSP	MAXSOLS=
Specifies the maximum amount of time to spend solving the traveling salesman problem	TSP	MAXTIME=
Specifies whether to use a mixed integer linear programming solver	TSP	MILP=
Specifies the output data table for the traveling salesman problem	TSP	OUT=
Specifies the stopping criterion based on the relative objective gap	TSP	RELOBJGAP=
Specifies the stopping criterion based on the target objective value	TSP	TARGET=

Table 3.3 lists the supported DIRECTION= option values in the PROC OPTNETWORK statement.

Table 3.3 Supported Input Formats by Statement

	DIRECTION=	
Statement	UNDIRECTED	DIRECTED
BICONNECTEDCOMPONENTS	X	
CLIQUE	X	
CONNECTEDCOMPONENTS		
ALGORITHM=		
DFS	X	X
PARALLEL, UNIONFIND	X	
CYCLE	X	X
LINEARASSIGNMENT		X
MINCOSTFLOW		X
MINCUT	X	
MINSPANTREE	X	
PATH	X	X
SHORTESTPATH	X	X
SUMMARY		
BICONNECTEDCOMPONENTS	X	

Table 3.3 (continued)

	DIRECTION=	
Statement	UNDIRECTED	DIRECTED
DIAMETERAPPROX=	X	
otherwise	X	X
TRANSITIVECLOSURE	X	X
TSP	X	X

For each algorithm statement in the OPTNETWORK procedure, Table 3.4 indicates which output data table options you can specify and whether the algorithm populates the data tables that are specified in the OUTNODES= and OUTLINKS= options in the PROC OPTNETWORK statement.

Table 3.4 Output Options by Statement

Statement	OUTNODES=	OUTLINKS=	Algorithm Statement Options
BICONNECTEDCOMPONENTS	X	X	
CLIQUE			OUT=
CONNECTEDCOMPONENTS	X	X	
CYCLE			OUT=
LINEARASSIGNMENT			OUT=
MINCOSTFLOW		X	
MINCUT			OUTCUTSETS=, OUTPARTITIONS=
MINSPANTREE			OUT=
PATH			OUTPATHSLINKS=, OUTPATHSNODES=
SHORTESTPATH			OUTPATHS=, OUTWEIGHTS=
SUMMARY	X	X	OUT=
TRANSITIVECLOSURE			OUT=
TSP	X		OUT=

PROC OPTNETWORK Statement

PROC OPTNETWORK < *options* > **;**

The PROC OPTNETWORK statement invokes the OPTNETWORK procedure. You can specify the following *options* to define the input and output data tables, the log levels, and various other processing controls:

DIRECTION=DIRECTED | UNDIRECTED

specifies whether the input graph should be considered directed or undirected. You can specify the following values:

DIRECTED specifies the graph as directed. In a directed graph, each link (i, j) has a direction that defines how something (such as information) can flow over that link. In link (i, j), the flow is from node i to node j ($i \rightarrow j$). Node i is called the *source* (*tail*) node, and node j is called the *sink* (*head*) node.

UNDIRECTED specifies the graph as undirected. In an undirected graph, each link $\{i, j\}$ has no direction and the flow can be in either direction. That is, $\{i, j\} = \{j, i\}$.

By default, DIRECTION=UNDIRECTED. For more information, see the section "Graph Input Data" on page 41.

INCLUDESELFLINK
includes self-links, such as (i, i), when an input graph is read. By default, when PROC OPTNETWORK reads the LINKS= data table, it removes all self-links.

INDEXOFFSET=*number*
specifies the index offset for identifiers in the log and results output data tables. For example, if three cycles are found in cycle enumeration, they are labeled cycles 1, 2, and 3 by default. If INDEXOFFSET=4, they are labeled cycles 4, 5, and 6. The value of *number* must be an integer greater than or equal to 0. By default, INDEXOFFSET=1.

LINKS=*CAS-libref.data-table*
specifies the input data table that contains the graph link information. *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 6 in Chapter 2, "Introduction."

For more information about this input table, see the section "Links Input Data" on page 41.

LOGFREQTIME=*number*
LOGFREQUENCYTIME=*number*
controls the frequency, in *number* of seconds, for displaying iteration logs for some algorithms. This option is useful for computationally intensive algorithms. Setting this value too low can hurt algorithm performance. The value of *number* can be any integer greater than or equal to 1. By default, LOGFREQTIME=5.

LOGLEVEL=NONE | BASIC | MODERATE | AGGRESSIVE
controls the amount of information that is displayed in the SAS log. You can specify the following values:

NONE turns off all procedure-related messages in the SAS log.

BASIC displays a brief summary of the algorithmic processing.

MODERATE displays a moderately detailed summary of the input, output, and algorithmic processing.

AGGRESSIVE displays a more detailed summary of the input, output, and algorithmic processing.

By default, LOGLEVEL=BASIC.

NODES=*CAS-libref.data-table*

specifies the input data table that contains the graph node information. *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 6 in Chapter 2, "Introduction."

For more information about this input table, see the section "Nodes Input Data" on page 45.

NODESSUBSET=*CAS-libref.data-table*

specifies the input data table that contains the graph node subset information. *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 6 in Chapter 2, "Introduction."

For more information about this input table, see the section "Nodes Subset Input Data" on page 45.

NTHREADS=*number*

specifies the maximum number of threads to use for multithreaded processing. Some of the algorithms can take advantage of multicore machines and can run faster when *number* is greater than 1 (see Table 3.5 for a list). Algorithms that cannot take advantage of this option use only one thread even if *number* is greater than 1. For distributed execution, *number* specifies the maximum number of threads to use on each machine. 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.

OUTLINKS=*CAS-libref.data-table*

specifies the output data table to contain the graph link information along with any results from the algorithms that calculate metrics on links. *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 output data table. For more information about this two-level name, see the section "Using CAS Sessions and CAS Engine Librefs" on page 6 in Chapter 2, "Introduction."

For examples of the content of this output data table, see the various algorithm sections.

OUTNODES=*CAS-libref.data-table*

specifies the output data table to contain the graph node information along with any results from the algorithms that calculate metrics on nodes. *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 output data table. For more information about this two-level name, see the section "Using CAS Sessions and CAS Engine Librefs" on page 6 in Chapter 2, "Introduction."

For examples of the content of this output data table, see the various algorithm sections.

STANDARDIZEDLABELS

specifies that the input graph data are in a standardized format, as described in the section "Standardized Labels" on page 46.

TIMETYPE=CPU | REAL

specifies whether CPU time or real time is used for each algorithm's MAXTIME= option (where applicable). You can specify the following values:

CPU specifies units of CPU time. The time restriction is applied per processing machine (not across all machines).

REAL specifies units of real time.

By default, TIMETYPE=REAL.

BICONNECTEDCOMPONENTS Statement

BICONNECTEDCOMPONENTS ;

The BICONNECTEDCOMPONENTS statement requests that PROC OPTNETWORK find biconnected components and articulation points of an undirected input graph. For more information, see the section "Biconnected Components and Articulation Points" on page 51.

BY Statement

BY *variables* **;**

You can specify a BY statement in PROC OPTNETWORK to obtain separate analyses of observations in groups that are defined by the values of the BY variables. If you specify more than one BY statement, only the last one specified is used. For more information, see the discussion of BY-group processing in *SAS Language Reference: Concepts*.

The BY statement in PROC OPTNETWORK is not supported when either a nodes or nodes subset data table is used. The BY variable must come from the LINKS= data table. An example of this is shown in "Example 3.9: Shortest Path in a Road Network by Date and Time" on page 141.

All parameter settings apply to each individual group independently (not to the entire process as a whole). For example, when a stopping criterion such as the MAXTIME= option is specified for a particular algorithm, this limit pertains to each individual group as it is processed.

CLIQUE Statement

CLIQUE < *options* > **;**

The CLIQUE statement invokes an algorithm that finds maximal cliques in the input graph. For more information about maximal cliques, see the section "Clique Enumeration" on page 54.

You can specify the following *options*:

MAXCLIQUES=*number* **| ALL**
specifies the maximum number of cliques to return during clique enumeration. You can specify either a *number* (which can be any 32-bit integer greater than or equal to 1) or ALL (which represents the maximum that can be represented by a 32-bit integer). By default, MAXCLIQUES=1.

MAXTIME=*number*
specifies the maximum amount of time to spend finding cliques. The type of time (either CPU time or real time) is determined by the value of the TIMETYPE= option in the PROC OPTNETWORK statement. The default is the largest number that can be represented by a double.

OUT=*CAS-libref.data-table*
specifies the output data table to contain the maximal cliques. *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 output data table. For more information about this two-level name, see the section "Using CAS Sessions and CAS Engine Librefs" on page 6 in Chapter 2, "Introduction."

CONNECTEDCOMPONENTS Statement

CONNECTEDCOMPONENTS < *options* > **;**

The CONNECTEDCOMPONENTS statement invokes an algorithm that finds the connected components of the input graph. For more information about connected components, see the section "Connected Components" on page 58. You can specify the following *options*:

ALGORITHM=AUTOMATIC | DFS | PARALLEL | UNIONFIND
specifies the algorithm to use for calculating connected components. You can specify the following values:

AUTOMATIC automatically determines the algorithm for connected components.

DFS uses the depth-first search algorithm for connected components.

PARALLEL uses the distributed parallel union-find algorithm for connected components. You can specify this value when the number of machines in your session is greater than 1. You can use this algorithm only with undirected graphs.

UNIONFIND uses the union-find algorithm for connected components. You can use this algorithm only with undirected graphs.

By default, ALGORITHM=UNIONFIND for undirected graphs, and ALGORITHM=DFS for directed graphs.

INTERNALFORMAT=FULL | THIN
specifies the internal graph format for the connected components algorithm to use. You can specify the following values:

FULL stores the graph in standard (adjacency-list-based) format.

THIN stores the graph in thin (simple list of links) format. This option can improve performance in some cases both by reducing memory and by simplifying the construction of the internal data structures. This option causes PROC OPTNETWORK to skip the removal of duplicate links when it reads in the graph (which has no effect on the resulting components).

By default, INTERNALFORMAT=THIN. You cannot use the option INTERNALFORMAT=FULL with ALGORITHM=PARALLEL. If you do, it will reset to the default value of THIN. For more information, see the section "Graph Input Data" on page 41.

CYCLE Statement

CYCLE < *options* > **;**

The CYCLE statement invokes an algorithm that finds the cycles (or the existence of a cycle) in the input graph. For more information about cycles, see the section "Cycle Enumeration" on page 62.

You can specify the following *options*:

ALGORITHM=BACKTRACK | BUILD
specifies which algorithm to use in enumerating cycles. You can specify the following values:

BACKTRACK uses a backtracking algorithm based on Johnson (1975).

BUILD uses a building algorithm based on Liu and Wang (2006).

By default, ALGORITHM=BACKTRACK for MAXLENGTH greater than 20; otherwise, ALGORITHM=BUILD.

MAXCYCLES=*number* **| ALL**
specifies the maximum number of cycles to return during cycle enumeration. You can specify either a *number* (which can be any 32-bit integer greater than or equal to 1) or ALL (which represents the maximum that can be represented by a 32-bit integer). By default, MAXCYCLES=1.

MAXLENGTH=*number*
specifies the maximum number of links in a cycle. Any cycle whose length is greater than *number* is removed from the results. The default is the largest number that can be represented by a 32-bit integer, which causes no cycles to be removed from the results.

MAXLINKWEIGHT=*number*
specifies the maximum sum of link weights in a cycle. Any cycle whose sum of link weights is greater than *number* is removed from the results. The default is the largest number that can be represented by a double, which causes no cycles to be removed from the results.

MAXNODEWEIGHT=*number*
specifies the maximum sum of node weights in a cycle. Any cycle whose sum of node weights is greater than *number* is removed from the results. The default is the largest number that can be represented by a double, which causes no cycles to be removed from the results.

MAXTIME=*number*
specifies the maximum amount of time to spend finding cycles. The type of time (either CPU time or real time) is determined by the value of the TIMETYPE= option in the PROC OPTNETWORK statement. The default is the largest number that can be represented by a double.

MINLENGTH=*number*
specifies the minimum number of links in a cycle. Any cycle that has fewer links than *number* is removed from the results. By default, MINLENGTH=1 and no cycles are removed from the results.

MINLINKWEIGHT=*number*
specifies the minimum sum of link weights in a cycle. Any cycle whose sum of link weights is less than *number* is removed from the results. The default is the largest (in magnitude) negative number that can be represented by a double, which causes no cycles to be removed from the results.

MINNODEWEIGHT=*number*
specifies the minimum sum of node weights in a cycle. Any cycle whose sum of node weights is less than *number* is removed from the results. The default is the largest (in magnitude) negative number that can be represented by a double, which causes no cycles to be removed from the results.

OUT=*CAS-libref.data-table*
specifies the output data table to contain the cycles found. *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 output data table. For more information about this two-level name, see the section "Using CAS Sessions and CAS Engine Librefs" on page 6 in Chapter 2, "Introduction."

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The procedure processes the DISPLAY statement on a CAS server and thus sends only a subset of ODS tables to the SAS client. Because ODS statements are processed on a SAS client, first all the generated display tables are sent to the client, and then the client creates a subset.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables that are sent to the client after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 116. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create CAS output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of CAS output tables to create. Each entry in the list has either a *key=value* format or a *key* format:

key=value specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the CAS output table name.

key specifies *key* as the ODS table name and also as the CAS output table name.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output CAS tables for all display tables. The name of the created output CAS table is the same as the corresponding display table name. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing CAS output table of the same name.

REPEATED
replicates all CAS output tables on all nodes.

The output tables that the OPTNETWORK procedure produces when you use the DISPLAYOUT statement are a transposed version of the displayed tables. This allows for easier use in subsequent analyses, especially when it is used together with BY-group processing. An example of using the DISPLAYOUT statement is shown in "Example 3.9: Shortest Path in a Road Network by Date and Time" on page 141.

LINEARASSIGNMENT Statement

LINEARASSIGNMENT < *options* > **;**

LAP < *options* > **;**

The LINEARASSIGNMENT statement invokes an algorithm that 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 graph. You define the input data as a directed graph by specifying the LINKS= option in the PROC OPTNETWORK statement, where the costs are defined as link weights. Internally, the graph is treated as a bipartite graph in which the *from* nodes define one part and the *to* nodes define the other part.

The linear assignment problem is described in the section "Linear Assignment (Matching)" on page 67.

You can specify the following *option* in the LINEARASSIGNMENT statement:

OUT=*CAS-libref.data-table*
specifies the output data table to contain the solution to the linear assignment problem.

LINKSVAR Statement

LINKSVAR < *options* > **;**

The LINKSVAR statement enables you to explicitly specify the data variable names for PROC OPTNETWORK to use when it reads the data table that you specify in the LINKS= option in the PROC OPTNETWORK statement. For more information about the format of the links input data table, see the section "Links Input Data" on page 41.

You can specify the following *options*:

AUXWEIGHT=*column*
specifies the name of the data variable for the auxiliary link weights. The value of the *column* variable must be numeric.

FROM=*column*
specifies the name of the data variable for the *from* nodes. The value of the *column* variable can be numeric or character.

LOWER=*column*
specifies the name of the data variable for the link lower bounds. The value of the *column* variable must be numeric.

TO=*column*
specifies the name of the data variable for the *to* nodes. The value of the *column* variable can be numeric or character.

UPPER=*column*
specifies the name of the data variable for the link upper bounds. The value of the *column* variable must be numeric.

WEIGHT=*column*
specifies the name of the data variable for the link weights. The value of the *column* variable must be numeric.

MINCOSTFLOW Statement

MINCOSTFLOW < *options* > **;**

MCF < *options* > **;**

The MINCOSTFLOW statement invokes an algorithm that solves the minimum-cost network flow problem on an input graph. The minimum-cost network flow problem is described in the section "Minimum-Cost Network Flow" on page 68.

You can specify the following *options* in the MINCOSTFLOW statement:

INTERNALFORMAT=FULL | THIN
specifies the internal graph format for the minimum-cost network flow algorithm to use. You can specify the following values:

FULL stores the graph in standard (adjacency-list-based) format.

THIN stores the graph in thin (simple list of links) format. This option can improve performance in some cases both by reducing memory and by simplifying the construction of the internal data structures. This option causes PROC OPTNETWORK to skip the removal of duplicate links when it reads in the graph.

By default, INTERNALFORMAT=FULL. For more information, see the section "Graph Input Data" on page 41.

LOGFREQ=*number*
LOGFREQUENCY=*number*
controls the frequency for displaying iteration logs for minimum-cost network flow calculations that use the network simplex algorithm. For graphs that contain one component, this option displays progress every *number* of simplex iterations; the default is 10,000. For graphs that contain multiple components, when you also specify LOGLEVEL=MODERATE, this option displays progress after processing every *number* of components; the default is based on the number of components. When you also specify LOGLEVEL=AGGRESSIVE, the simplex iteration log for each component is displayed with a frequency of *number*.

The value of *number* can be any integer greater than or equal to 1. Setting this value too low can hurt performance on large-scale graphs.

MAXTIME=*number*
specifies the maximum amount of time to spend calculating minimum-cost network flows. The type of time (either CPU time or real time) is determined by the value of the TIMETYPE= option in the PROC OPTNETWORK statement. The default is the largest number that can be represented by a double.

MINCUT Statement

MINCUT < *options* > **;**

The MINCUT statement invokes an algorithm that finds the minimum link-weighted cut of an input graph. The minimum-cut problem is described in the section "Minimum Cut" on page 75.

You can specify the following *options*:

MAXCUTS=*number*
specifies the maximum number of cuts for the algorithm to return. The minimal cut and any others that it finds during the search, up to *number*, are returned. By default, MAXCUTS=1.

MAXWEIGHT=*number*
specifies the maximum weight of the cuts for the algorithm to return. Only cuts whose weight is less than or equal to *number* are returned. The default is the largest number that can be represented by a double.

OUTCUTSETS=*CAS-libref.data-table*
OUT=*CAS-libref.data-table*
specifies the output data table to contain the minimum cut sets to the minimum-cut problem.

OUTPARTITIONS=*CAS-libref.data-table*
specifies the output data table to contain the minimum cut partitions to the minimum-cut problem.

MINSPANTREE Statement

MINSPANTREE < *options* > **;**

MST < *options* > **;**

The MINSPANTREE statement invokes an algorithm that solves the minimum link-weighted spanning tree problem on an input graph. The minimum spanning tree problem is described in the section "Minimum Spanning Tree" on page 79.

You can specify the following *options* in the MINSPANTREE statement:

INTERNALFORMAT=FULL | THIN
specifies the internal graph format for the minimum spanning tree algorithm to use. You can specify the following values:

FULL stores the graph in standard (adjacency-list-based) format.

THIN stores the graph in thin (simple list of links) format. This option can improve performance in some cases both by reducing memory and by simplifying the construction of the internal data structures. This option causes PROC OPTNETWORK to skip the removal of duplicate links when it reads in the graph.

By default, INTERNALFORMAT=THIN. For more information, see the section "Graph Input Data" on page 41.

OUT=*CAS-libref.data-table*
specifies the output data table to contain the solution to the minimum link-weighted spanning tree problem.

NODESSUBSETVAR Statement

NODESSUBSETVAR < *options* > **;**

The NODESSUBSETVAR statement enables you to explicitly specify the data variable names for PROC OPTNETWORK to use when it reads the data table that you specify in the NODESSUBSET= option in the PROC OPTNETWORK statement. For more information about the format of the node subset input data table, see the section "Nodes Input Data" on page 45.

You can specify the following *options*:

NODE=*column*
specifies the name of the data variable for the nodes. The value of the *column* variable can be numeric or character.

SINK=*column*
specifies the name of the data variable for the sink indicator. The value of the *column* variable must be numeric.

SOURCE=*column*
specifies the name of the data variable for the source indicator. The value of the *column* variable must be numeric.

NODESVAR Statement

NODESVAR < *options* > **;**

The NODESVAR statement enables you to explicitly specify the data variable names for PROC OPTNETWORK to use when it reads the data table that you specify in the NODES= option in the PROC OPTNETWORK statement. For more information about the format of the node input data table, see the section "Nodes Input Data" on page 45.

You can specify the following *options*:

LOWER=*column*
specifies the name of the data variable for the node lower bounds. The value of the *column* variable must be numeric.

NODE=*column*
specifies the name of the data variable for the nodes. The value of the *column* variable can be numeric or character.

UPPER=*column*
specifies the name of the data variable for the node upper bounds. The value of the *column* variable must be numeric.

WEIGHT=*column*
specifies the name of the data variable for the node weights. The value of the *column* variable must be numeric.

PATH Statement

PATH < *options* > **;**

The PATH statement invokes an algorithm that finds the paths in the input graph. For more information about paths, see the section "Path Enumeration" on page 81.

You can specify the following *options*:

INTERNALFORMAT=FULL | THIN
specifies the internal graph format for the path enumeration algorithm to use. You can specify the following values:

FULL stores the graph in standard (adjacency-list-based) format.

THIN stores the graph in thin (simple list of links) format. This option can improve performance in some cases both by reducing memory and by simplifying the construction of the internal data structures. This option causes PROC OPTNETWORK to skip the removal of duplicate links when it reads in the graph.

By default, INTERNALFORMAT=THIN. For more information, see the section "Graph Input Data" on page 41.

MAXLENGTH=*number*
specifies the maximum number of links in a path. Any path whose length is greater than *number* is removed from the results. The default is the largest number that can be represented by a 32-bit integer, which causes no paths to be removed from the results.

MAXLINKWEIGHT=*number*
specifies the maximum sum of link weights in a path. Any path whose sum of link weights is greater than *number* is removed from the results. The default is the largest number that can be represented by a double, which causes no paths to be removed from the results.

MAXNODEWEIGHT=*number*
specifies the maximum sum of node weights in a path. Any path whose sum of node weights is greater than *number* is removed from the results. The default is the largest number that can be represented by a double, which causes no paths to be removed from the results.

MAXTIME=*number*
specifies the maximum amount of time to spend finding paths. The type of time (either CPU time or real time) is determined by the value of the TIMETYPE= option in the PROC OPTNETWORK statement. The default is the largest number that can be represented by a double.

MINLENGTH=*number*
specifies the minimum number of links in a path. Any path that has fewer links than *number* is removed from the results. By default, MINLENGTH=1 and no paths are removed from the results.

MINLINKWEIGHT=*number*
specifies the minimum sum of link weights in a path. Any path whose sum of link weights is less than *number* is removed from the results. The default is the largest (in magnitude) negative number that can be represented by a double, which causes no paths to be removed from the results.

MINNODEWEIGHT=*number*
specifies the minimum sum of node weights in a path. Any path whose sum of node weights is less than *number* is removed from the results. The default is the largest (in magnitude) negative number that can be represented by a double, which causes no paths to be removed from the results.

OUTPATHSLINKS=*CAS-libref.data-table*

OUTPATHS=*CAS-libref.data-table*
specifies the output data table to contain the path links. *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 output data table. For more information about this two-level name, see the section "Using CAS Sessions and CAS Engine Librefs" on page 6 in Chapter 2, "Introduction."

OUTPATHSNODES=*CAS-libref.data-table*
specifies the output data table to contain the path nodes. *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 output data table. For more information about this two-level name, see the section "Using CAS Sessions and CAS Engine Librefs" on page 6 in Chapter 2, "Introduction."

SINK=*sink-node*
specifies the sink node for path calculations. This setting overrides the use of the variable sink in the data table that you specify in the NODESSUBSET= option in the PROC OPTNETWORK statement.

SOURCE=*source-node*
specifies the source node for path calculations. This setting overrides the use of the variable source in the data table that you specify in the NODESSUBSET= option in the PROC OPTNETWORK statement.

SHORTESTPATH Statement

SHORTESTPATH < *options* > **;**

The SHORTESTPATH statement invokes an algorithm that calculates shortest paths between pairs of nodes in the input graph. By default, PROC OPTNETWORK finds a shortest path for each possible combination of source and sink nodes. For more information about the shortest path algorithm, see the section "Shortest Path" on page 85.

You can specify the following *options*:

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 largest number that can be represented by a double, which causes no paths to be removed from the results.

OUTPATHS=*CAS-libref.data-table*
OUT=*CAS-libref.data-table*
specifies the output data table to contain the shortest paths. *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 output data table. For more information about this two-level name, see the section "Using CAS Sessions and CAS Engine Librefs" on page 6 in Chapter 2, "Introduction."

OUTWEIGHTS=*CAS-libref.data-table*
specifies the output data table to contain the shortest path summaries. *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 output data table. For more information about this two-level name, see the section "Using CAS Sessions and CAS Engine Librefs" on page 6 in Chapter 2, "Introduction."

SINK=*sink-node*
specifies the sink node for shortest path calculations. This setting overrides the use of the variable sink in the data table that you specify in the NODESSUBSET= option in the PROC OPTNETWORK statement.

SOURCE=*source-node*
specifies the source node for shortest path calculations. This setting overrides the use of the variable source in the data table that you specify in the NODESSUBSET= option in the PROC OPTNETWORK statement.

SUMMARY Statement

SUMMARY < *options* > **;**

The SUMMARY statement invokes an algorithm that calculates various summary metrics for an input graph. For more information about summary metrics, see the section "Summary Statistics" on page 96.

You can specify the following *options*:

BICONNECTEDCOMPONENTS
calculates information about biconnected components. You can use this option only for an undirected graph.

CONNECTEDCOMPONENTS
calculates information about connected components.

DIAMETERAPPROX=WEIGHT | UNWEIGHT | BOTH
calculates information about the approximate diameter and specifies which type of calculation to perform. Use this option when calculating the exact diameter (by calculating all shortest paths) is too computationally expensive. You can specify the following values:

WEIGHT calculates the approximate diameter by using the weighted graph.

UNWEIGHT calculates the approximate diameter by using the unweighted graph.

BOTH calculates the approximate diameter by using both the weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (if you use 1.0 for each link weight). This option works only for undirected graphs.

OUT=*CAS-libref.data-table*
specifies the output data table to contain the summary results. *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 output data table. For more information about this two-level name, see the section "Using CAS Sessions and CAS Engine Librefs" on page 6 in Chapter 2, "Introduction."

SHORTESTPATH=WEIGHT | UNWEIGHT | BOTH
calculates information about shortest paths and specifies which type of calculation to perform. You can specify the following values:

WEIGHT calculates shortest paths by using the weighted graph.

UNWEIGHT calculates shortest paths by using the unweighted graph.

BOTH calculates shortest paths by using both the weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (if you use 1.0 for each link weight).

TRANSITIVECLOSURE Statement

TRANSITIVECLOSURE < *option* > **;**

The TRANSITIVECLOSURE statement invokes an algorithm that calculates the transitive closure of an input graph. For more information about transitive closure, see the section "Transitive Closure" on page 103.

You can specify the following *option*:

OUT=*CAS-libref.data-table*
specifies the output data table to contain the transitive closure results. *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 output data table. For more information about this two-level name, see the section "Using CAS Sessions and CAS Engine Librefs" on page 6 in Chapter 2, "Introduction."

TSP Statement

TSP < *options* > **;**

The TSP statement invokes an algorithm that solves the traveling salesman problem, which is described in the section "Traveling Salesman Problem" on page 105. 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 options are the same as those described for the OPTMILP procedure in the *SAS/OR User's Guide: Mathematical Programming*.

You can specify the following *options*:

ABSOBJGAP=*number*

ABSOLUTEOBJECTIVEGAP=*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; the default value is 1E–6.

CUTOFF=*number*

cuts off any branch-and-bound nodes in a minimization problem that has an objective value greater than *number*. The default is the largest number that can be represented by a double.

CUTSTRATEGY=AUTOMATIC | NONE | MODERATE | AGGRESSIVE

specifies the level of mixed integer linear programming cutting planes to be generated by PROC OPTNETWORK. TSP-specific cutting planes are always generated. You can specify the following values:

AUTOMATIC generates cutting planes based on a strategy determined by the mixed integer linear programming solver.

NONE disables the generation of mixed integer linear programming cutting planes (some TSP-specific cutting planes are still active for validity).

MODERATE uses a moderate cutting strategy.

AGGRESSIVE uses an aggressive cutting strategy.

By default, CUTSTRATEGY=NONE.

HEURISTICS=AUTOMATIC | NONE | BASIC | MODERATE | AGGRESSIVE

controls the level of initial and primal heuristics that PROC OPTNETWORK applies. This level determines how frequently PROC OPTNETWORK 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. You can specify the following values:

AUTOMATIC applies the default level of heuristics.

NONE disables all initial and primal heuristics.

BASIC applies basic initial and primal heuristics at low frequency.

MODERATE applies most initial and primal heuristics at moderate frequency.

AGGRESSIVE applies all initial primal heuristics at high frequency.

By default, HEURISTICS=AUTOMATIC.

LOGFREQ=*number*

LOGFREQUENCY=*number*

specifies the time interval (in seconds) for printing information in the node log. The default value is 5. If *number* is set to 0, then the node log is disabled. If *number* is positive, then the root node processing information is printed and, if possible, an entry is made every *number* seconds. An entry is also made each time a better integer solution is found.

The value of *number* can be any integer greater than or equal to 0.

MAXNODES=*number*

specifies the maximum number of branch-and-bound nodes to be processed. The default is the largest number that can be represented by a 32-bit integer.

MAXSOLS=*number*

specifies a stopping criterion. If the *number* of solutions has been found, then the procedure stops. The default is the largest number that can be represented by a 32-bit integer.

MAXTIME=*number*

specifies the maximum amount of time to spend solving the traveling salesman problem. The type of time (either CPU time or real time) is determined by the value of the TIMETYPE= option in the PROC OPTNETWORK statement. The default is the largest number that can be represented by a double.

MILP=TRUE | FALSE

specifies whether to use a mixed integer linear programming (MILP) solver to solve 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 PROC OPTNETWORK uses its initial heuristics to find a feasible, but not necessarily optimal, tour as quickly as possible. You can specify the following values:

TRUE uses a mixed integer linear programming solver.

FALSE does not use a mixed integer linear programming solver.

By default, MILP=TRUE.

OUT=*CAS-libref.data-table*

specifies the output data table to contain the solution to the traveling salesman problem. *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 output data table. For more information about this two-level name, see the section "Using CAS Sessions and CAS Engine Librefs" on page 6 in Chapter 2, "Introduction."

RELOBJGAP=*number*

RELATIVEOBJECTIVEGAP=*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

$$|\text{BestInteger} - \text{BestBound}| / (1\text{E}{-10} + |\text{BestBound}|)$$

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.

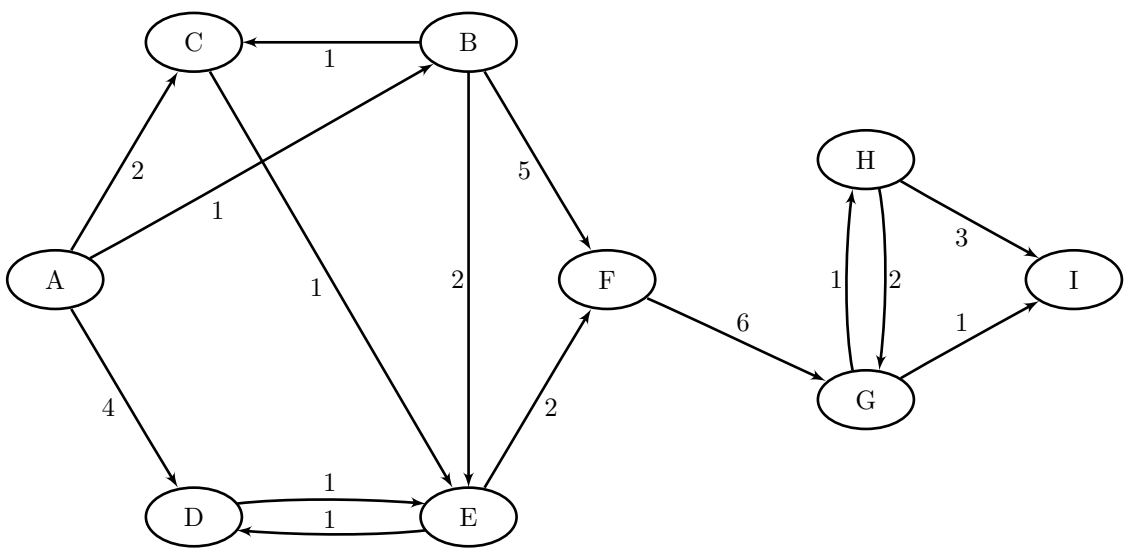
TARGET=*number*
specifies a stopping criterion for minimization problems. If the best integer objective is less than or equal to *number*, the solver stops. The default is the largest (in magnitude) negative number that can be represented by a double.

Details: OPTNETWORK Procedure

Graph Input Data

This section describes how to input a graph for analysis by PROC OPTNETWORK. Let $G = (N, A)$ define a graph that contains a set N of nodes and a set A of links. Consider the directed graph shown in Figure 3.5.

Figure 3.5 A Simple Directed Graph



Notice that each node and link has associated attributes: a node label and a link weight.

Links Input Data

The LINKS= option in the PROC OPTNETWORK statement defines the data table that contains the list of links in the graph. A link is represented as a pair of nodes, which are defined by using either numeric or character labels. The links data table is expected to contain some combination of the following possible variables:

- auxweight: the auxiliary link weight (must be numeric)
- from: the *from* node (can be numeric or character)
- lower: the link lower bound (must be numeric)
- to: the *to* node (can be numeric or character)

- upper: the link upper bound (must be numeric)
- weight: the link weight (must be numeric)

As described for the DIRECTION= option, if the graph is undirected, the *from* and *to* labels are interchangeable. If the weights are not given for algorithms that call for link weights, they are all assumed to be 1.

The data variable names can have any values that you want. If you use nonstandard names, you must identify the variables by using the LINKSVAR statement, as described in the section "LINKSVAR Statement" on page 31.

For example, the following two data tables identify the same graph:

```
data mycas.LinkSetInA;
   input from $ to $ weight;
   datalines;
A B 1
A C 2
A D 4
;

data mycas.LinkSetInB;
   input source_node $ sink_node $ value;
   datalines;
A B 1
A C 2
A D 4
;
```

You can present these data tables to PROC OPTNETWORK by using the following equivalent statements:

```
proc optnetwork
   links = mycas.LinkSetInA;
run;

proc optnetwork
   links     = mycas.LinkSetInB;
   linksVar
      from   = source_node
      to     = sink_node
      weight = value;
run;
```

The directed graph G shown in Figure 3.5 can be represented by the following links data table, mycas.LinkSetIn:

```
data mycas.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, declare it as a directed graph, and output the resulting links and nodes data tables. These statements do not run any algorithms, so the resulting output contains only the input graph.

```
proc optnetwork
   direction = directed
   links     = mycas.LinkSetIn
   outNodes  = mycas.NodeSetOut
   outLinks  = mycas.LinkSetOut;
run;
```

The output data table mycas.NodeSetOut, shown in Figure 3.6, now contains the nodes that are read from the input links data table. The variable node shows the label associated with each node.

Figure 3.6 Nodes Data Table of a Simple Directed Graph

node
A
B
C
D
E
F
G
H
I

The output data table mycas.LinkSetOut, shown in Figure 3.7, contains the links that were read from the input links data table. The variables from and to show the associated node labels.

Figure 3.7 Links Data Table of a Simple Directed Graph

Obs	from	to	weight
1	A	B	1
2	A	C	2
3	A	D	4
4	B	C	1
5	B	E	2
6	B	F	5
7	C	E	1
8	D	E	1
9	E	D	1
10	E	F	2
11	F	G	6
12	G	H	1
13	H	G	2
14	G	I	1
15	H	I	3

If you define this graph as undirected, then reciprocal links (for example, $D \rightarrow E$ and $D \leftarrow E$) are treated as the same link, and duplicates are removed. PROC OPTNETWORK aggregates the attributes of each

duplicate link by taking the minimum value (for each attribute). By default, DIRECTION=UNDIRECTED, so you can just remove this option to declare the graph as undirected.

The following statements read in this graph, declare it as an undirected graph, and output the resulting links and nodes data tables:

```
proc optnetwork
   links    = mycas.LinkSetIn
   outNodes = mycas.NodeSetOut
   outLinks = mycas.LinkSetOut;
run;
```

The progress of the procedure is shown in Figure 3.8. The log now shows the number of links that were declared as duplicates and aggregated.

Figure 3.8 PROC OPTNETWORK Log: Links Data Table of a Simple Undirected Graph

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: The graph contains 2 duplicate links that are ignored.
NOTE: The number of nodes in the input graph is 9.
NOTE: The number of links in the input graph is 13.
NOTE: The Cloud Analytic Services server processed the request in 0.045168 seconds.
NOTE: The data set MYCAS.NODESETOUT has 9 observations and 1 variables.
NOTE: The data set MYCAS.LINKSETOUT has 13 observations and 3 variables.
```

The output data table mycas.NodeSetOut is equivalent to the one shown in Figure 3.6. However, the new links data table mycas.LinkSetOut, shown in Figure 3.9, contains two fewer links than before, because duplicates are aggregated.

Figure 3.9 Links Data Table of a Simple Undirected Graph

Obs	from	to	weight
1	A	B	1
2	A	C	2
3	A	D	4
4	B	C	1
5	B	E	2
6	B	F	5
7	C	E	1
8	D	E	1
9	E	F	2
10	F	G	6
11	G	H	1
12	G	I	1
13	H	I	3

Thin Internal Format

Certain algorithms can perform more efficiently when you specify INTERNALFORMAT=THIN in their respective algorithm statement. However, when you specify this option, PROC OPTNETWORK does not remove duplicate links. Instead, you should use the appropriate DATA steps to clean your data before calling PROC OPTNETWORK.

Nodes Input Data

The NODES= option in the PROC OPTNETWORK statement defines the data table that contains the list of nodes in the graph. This data table is used to assign node attributes.

The nodes data table is expected to contain some combination of the following possible variables:

- lower: the node lower bound (must be numeric)
- node: the node label (can be numeric or character)
- upper: the node upper bound (must be numeric)
- weight: the node weight (must be numeric)

You can specify any value that you want for the data table variable name. If you use a nonstandard name, you must identify the variable by using the NODESVAR statement, as described in the section "NODESVAR Statement" on page 34.

The data table that you specify in the LINKS= option defines the set of nodes that are incident to some link. If the graph contains a node that has no links (called a *singleton* node), then you must define this node in the NODES= data table. The following statements produce a graph that has three links but four nodes, including the singleton node D:

```
data mycas.NodeSetIn;
   input node $ @@;
   datalines;
A B C D
;

data mycas.LinkSetInS;
   input from $ to $ weight;
   datalines;
A B 1
A C 2
B C 1
;
```

If you specify duplicate entries in the nodes data table, PROC OPTNETWORK issues an error message and stops.

Nodes Subset Input Data

For some algorithms, you might want to process only a subset of the nodes that appear in the input graph. You can accomplish this by using the NODESSUBSET= option in the PROC OPTNETWORK statement. You can use the nodes subset data table in conjunction with the PATH or SHORTESTPATH statement. (See

the sections "Path Enumeration" on page 81 and "Shortest Path" on page 85, respectively.) The nodes subset data table is expected to contain some combination of the following variables:

- node: the node label (can be numeric or character)
- source: whether to process this node as a source node in (shortest) path algorithms (must be numeric)
- sink: whether to process this node as a sink node in (shortest) path algorithms (must be numeric)

The values in the nodes subset data table determine how to process nodes when the PATH or SHORTESTPATH statement is processed. A value of 1 for the source variable designates that the node is to be processed as a source; a value of 0 designates that the node is not to be processed as a source. You can use the same values for the sink variable to designate whether the node is to be processed as a sink. You can also use the missing indicator (.) in place of 0 to designate that a node is not to be processed.

The following example of a nodes subset data table might be used with the graph in Figure 3.5:

```
data mycas.NodeSubSetIn;
   input node $ source sink;
   datalines;
A 1 .
F . 1
E 1 .
;
```

The data table mycas.NodeSubSetIn indicates that you want to process the (shortest) paths for the source-sink pairs in $\{A, E\} \times \{F\}$ (the crossproduct of subsets $\{A, E\}$ and $\{F\}$).

Standardized Labels

For large-scale graphs, the processing stage that reads the nodes and links into memory can be time-consuming. Under the following assumptions, you can use the STANDARDIZEDLABELS option in the PROC OPTNETWORK statement to speed up this stage:

1. The links data table variables from and to are numeric.
2. The node and nodes subset data table variable node is numeric.
3. The node labels start from 0 and are consecutive nonnegative integers.

Consider the following links data table that uses numeric labels:

```
data mycas.LinkSetIn;
   input from to weight;
   datalines;
0 1 1
3 0 2
1 5 1
;
```

Using default settings, the following statements echo link and nodes data tables that contain three links and four nodes, respectively:

```
proc optnetwork
   links    = mycas.LinkSetIn
   outNodes = mycas.NodeSetOut
   outLinks = mycas.LinkSetOut;
run;
```

The log is shown in Figure 3.10.

Figure 3.10 PROC OPTNETWORK Log: A Simple Undirected Graph

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: The number of nodes in the input graph is 4.
NOTE: The number of links in the input graph is 3.
NOTE: The Cloud Analytic Services server processed the request in 0.03505 seconds.
NOTE: The data set MYCAS.NODESETOUT has 4 observations and 1 variables.
NOTE: The data set MYCAS.LINKSETOUT has 3 observations and 3 variables.
```

The output data table mycas.NodeSetOut, shown in Figure 3.11, contains the unique numeric node labels, $\{0, 1, 3, 5\}$.

Figure 3.11 Nodes Data Table of a Simple Directed Graph

Obs	node
1	0
2	1
3	3
4	5

Using standardized labels, the same input data table defines a graph that has six (not four) nodes:

```
proc optnetwork
   standardizedLabels
   links    = mycas.LinkSetIn
   outNodes = mycas.NodeSetOut
   outLinks = mycas.LinkSetOut;
run;
```

The log that results from using standardized labels is shown in Figure 3.12.

Figure 3.12 PROC OPTNETWORK Log: A Simple Undirected Graph Using Standardized Labels

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: The number of nodes in the input graph is 6.
NOTE: The number of links in the input graph is 3.
NOTE: The Cloud Analytic Services server processed the request in 0.034344 seconds.
NOTE: The data set MYCAS.NODESETOUT has 6 observations and 1 variables.
NOTE: The data set MYCAS.LINKSETOUT has 3 observations and 3 variables.
```

The output data table mycas.NodeSetOut, shown in Figure 3.13, now contains all node labels from 0 to 5, based on the assumptions when you use the STANDARDIZEDLABELS option.

Figure 3.13 Nodes Data Table of a Simple Directed Graph

Obs	node
1	0
2	1
3	2
4	3
5	4
6	5

Execution Modes and Data Movement

When you run PROC OPTNETWORK, the algorithmic execution mode and the underlying data movement that is implemented (required) to support that execution mode depend on which algorithm you select.

For a single-machine CAS server, there is no data movement. The algorithm runs on the same machine where the data reside. For a multiple-machine CAS server, the assumption is that the data reside in parts on one or more of the machines in the server.

Certain algorithms run only in single-machine mode. In such cases, one particular machine (chosen randomly) is given the role of the processing machine, and the data from all the other machines are moved over to this processing machine.

Some algorithms use multiple machines, but each machine requires a global view of the input data. In such cases, each data part is repeated on all machines. Each machine processes a portion of the work across the entire graph. The resulting output tables are distributed tables.

Other algorithms use multiple machines and require only a portion of the data. However, because the original data are usually randomly distributed, the first step is to shuffle data between machines such that the data are appropriately aligned for the particular algorithm's needs. When the data are aligned correctly, each machine processes a part of the data and then iteratively merges results across the grid to obtain the final result. Again, the resulting output tables are distributed tables.

In addition, on each machine, some of these algorithms (as well as the input phase) take advantage of multicore chip technology by executing on multiple threads simultaneously. You can use the NTHREADS= option in the PROC OPTNETWORK statement to specify the number of threads to use. 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). Setting this option to a number greater than the number of available cores might hurt performance. Specifying a high *number* 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 some circumstances, the OPTNETWORK procedure might use fewer threads than the specified *number* because the procedure's internal algorithms have determined that a smaller number is preferable.

In the case of BY-group processing, the data must first be partitioned such that each observation within a BY group resides on the same machine. If the data are not already partitioned, PROC OPTNETWORK shuffles the data appropriately as a first step. When the data are partitioned, the selected algorithm runs against the groups (on each machine) by using multiple threads (one group per thread). If the algorithm itself is a multithreaded algorithm, then it uses multiple threads (on each group) if and only if the value of the NTHREADS= option exceeds the number of groups assigned to the processing machine. In this case, the resulting output tables are distributed tables (partitioned by group). You can prepartition your input data by using the PARTITION= option in a DATA step. Prepartitioning avoids the need for PROC OPTNETWORK to shuffle the data. This option is described in *SAS Cloud Analytic Services: Language Reference*.

The data movement and execution modes for each algorithm are listed in Table 3.5. The table uses the abbreviations SM (single machine), MM (multiple machines), and MT (multithreaded execution).

Table 3.5 Execution Modes and Data Movement

Statement (and Options)	Data Movement	Processing Mode
BICONNECTEDCOMPONENTS	Moved to SM	SM
CLIQUE	Moved to SM	SM
CONNECTEDCOMPONENTS ALGORITHM=		
DFS, UNIONFIND	Moved to SM	SM
PARALLEL	Shuffled across MM	MM
CYCLE ALGORITHM=		
BACKTRACK	Moved to SM	SM
BUILD	Moved to SM	SM (MT)
LINEARASSIGNMENT	Moved to SM	SM
MINCOSTFLOW	Moved to SM	SM
MINCUT	Moved to SM	SM
MINSPANTREE	Moved to SM	SM
PATH	Repeated on MM	MM (MT)
SHORTESTPATH	Repeated on MM	MM (MT)
SUMMARY (other than shortest path)	Moved to SM	SM
SHORTESTPATH=	Moved to SM	SM (MT)
TRANSITIVECLOSURE	Moved to SM	SM
TSP	Moved to SM	SM

Because of communication costs, increasing the number of machines does not guarantee faster execution, especially when you are dealing with small graphs. 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*.

Numeric Limitations

Extremely large or extremely small numerical values might cause computational difficulties for some of the algorithms in PROC OPTNETWORK. 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, PROC OPTNETWORK 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 3.6, where M is defined as the largest number that can be represented by a double.

Table 3.6 Threshold Limits by Statement

	Graph Links				Graph Nodes		
Statement	weight	auxweight	lower	upper	weight	lower	upper
CYCLE	$\sqrt{M}$				$\sqrt{M}$		
LINEARASSIGNMENT	$\sqrt{M}$						
MINCOSTFLOW	1e15		1e15	1e15		1e15	1e15
MINCUT	$\sqrt{M}$						
MINSPANTREE	$\sqrt{M}$						
PATH	$\sqrt{M}$	$\sqrt{M}$	$\sqrt{M}$				
SHORTESTPATH	$\sqrt{M}$	$\sqrt{M}$					
SUMMARY							
DIAMETERAPPROX=	$\sqrt{M}$						
SHORTESTPATH=	$\sqrt{M}$						
TSP	1e20						

To obtain these limits, use the SAS function **constant**. For example, the following DATA step assigns $\sqrt{M}$ to a variable x and prints that value to the log:

```
data _null_;
   x = constant('SQRTBIG');
   put x=;
run;
```

Missing Values

For most of the algorithms in PROC OPTNETWORK, there is no valid interpretation for a missing value. If the user data contain a missing value, PROC OPTNETWORK issues an error message. One exception is for the minimum-cost network flow solver when you are setting the link or node bounds. In this case, a missing value is interpreted as the default bound value, as described in the section "Minimum-Cost Network Flow" on page 68.

Negative Link Weights

For certain algorithms in PROC OPTNETWORK, a negative link weight is not allowed. The following algorithm issues an error message if you provide a negative link weight:

- MINCUT

Size Limitations

PROC OPTNETWORK can handle any graph whose numbers of nodes and links are each less than or equal to 2,147,483,647 (the maximum that a 32-bit integer can represent). This maximum also applies to 64-bit systems. For graphs that contain two billion nodes (or links), memory restrictions also become a limiting factor.

If the data from your problem require a graph that contains more than two billion nodes (or links), there is usually a heuristic way to break the network into smaller networks based on problem-specific attributes. Then, using DATA steps (or a BY statement), you can process each of the smaller networks iteratively through repeated calls to PROC OPTNETWORK. By using DATA steps (or a BY statement), you can also often work around memory limitations, because the full graph never resides in memory.

An exception to this limitation is the parallel union-find algorithm for finding connected components. This algorithm is limited to 2,147,483,647 links *per machine* in your session configuration (rather than total links). This algorithm is still limited to 2,147,483,647 total nodes.

Biconnected Components and Articulation Points

A *biconnected component* of a graph $G = (N, A)$ is a connected subgraph that you cannot break into disconnected pieces by deleting any single node (and its incident links). An *articulation point* of a graph is a node 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 $i \in N$ that, if removed, breaks 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 where articulation points are important include airline hubs, electric circuits, network wires, protein bonds, traffic routers, and many other industrial applications.

In PROC OPTNETWORK, you can find biconnected components and articulation points of an input graph by using the BICONNECTEDCOMPONENTS statement. This algorithm works only with undirected graphs.

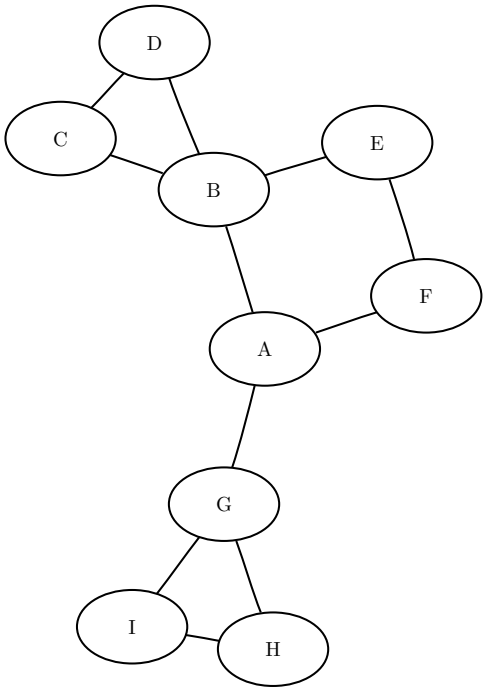
The results of the biconnected components algorithm are written to the output links data table that you specify in the OUTLINKS= option in the PROC OPTNETWORK statement. For each link in the links data table, the variable biconcomp identifies its component. The component identifiers are numbered sequentially, starting from the value of the INDEXOFFSET= option in the PROC OPTNETWORK statement. The results of the articulation points are written to the output nodes data table that you specify in the OUTNODES= option in the PROC OPTNETWORK statement. For each node in the nodes data table, the variable artpoint is either 1 (if the node is an articulation point) or 0 (otherwise).

The algorithm that PROC OPTNETWORK uses to compute biconnected components is a variant of the depth-first search algorithm (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 shown in Figure 3.14.

Figure 3.14 A Simple Undirected Graph G



The undirected graph G can be represented by the following links data table, mycas.LinkSetInBiCC:

```
data mycas.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 for G and output the results in the data tables mycas.LinkSetOut and mycas.NodeSetOut:

```
proc optnetwork
   links    = mycas.LinkSetInBiCC
   outLinks = mycas.LinkSetOut
   outNodes = mycas.NodeSetOut;
   biconnectedComponents;
run;
```

The output data table mycas.LinkSetOut contains the biconnected components of the input graph, as shown in Figure 3.15.

Figure 3.15 Biconnected Components of a Simple Undirected Graph

from	to	biconcomp
A	B	1
A	F	1
B	E	1
E	F	1
A	G	2
B	D	3
B	C	3
C	D	3
G	I	4
G	H	4
H	I	4

The output data table mycas.NodeSetOut contains the articulation points of the input graph, as shown in Figure 3.16.

Figure 3.16 Articulation Points of a Simple Undirected Graph

node	artpoint
A	1
B	1
D	0
G	1
I	0
F	0
E	0
H	0
C	0

The biconnected components are shown graphically in Figure 3.17 and Figure 3.18.

Figure 3.17 Biconnected Components C^1 and C^2

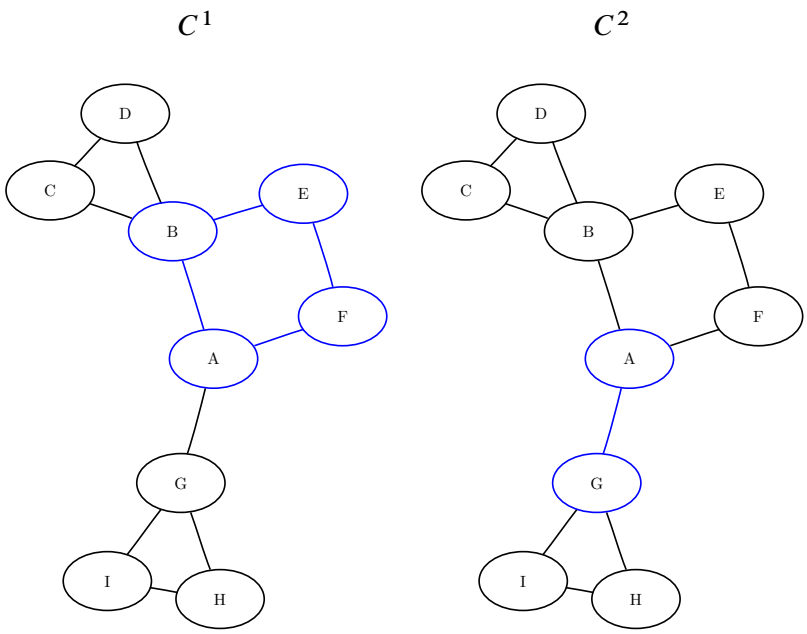
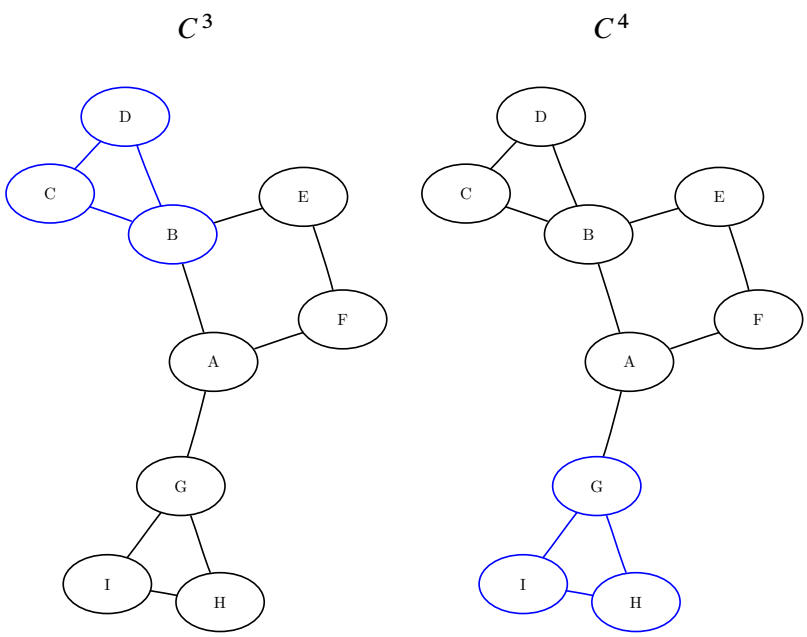


Figure 3.18 Biconnected Components C^3 and C^4



For a more detailed example, see "Example 3.1: Articulation Points in a Terrorist Network" on page 117.

Clique Enumeration

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 that is added. Finding cliques in graphs has applications in many industries, including bioinformatics, social networks, electrical engineering, and chemistry.

You can find the maximal cliques of an input graph by using the CLIQUE statement. The options for this statement are described in the section "CLIQUE Statement" on page 26. The clique algorithm works only with undirected graphs.

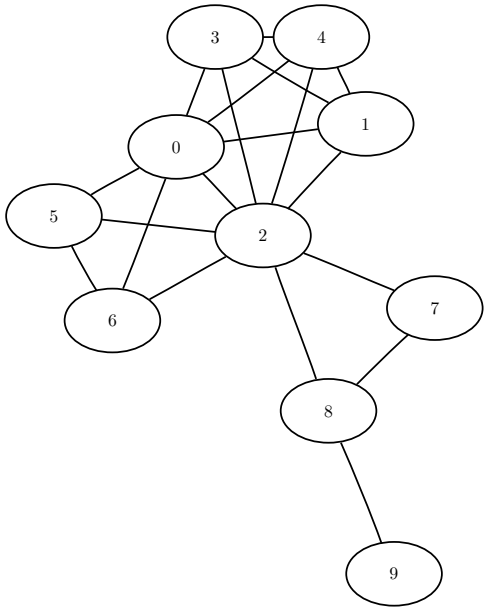
The results of the clique algorithm are written to the output data table that you specify in the OUT= option in the CLIQUE statement. Each node of each clique is listed in the output data table along with the variable clique to identify the clique to which it belongs. The clique identifiers are numbered sequentially, starting from the value of the INDEXOFFSET= option in the PROC OPTNETWORK statement. A node can appear multiple times in this data table if it belongs to multiple cliques.

The algorithm that PROC OPTNETWORK 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 usually 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 shown in Figure 3.19.

Figure 3.19 A Simple Undirected Graph G



The undirected graph G can be represented by the following links data table, mycas.LinkSetIn:

```
data mycas.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 table mycas.Cliques, and use the SQL procedure as a convenient way to create a local data set (CliqueSizes) of clique sizes:

```
proc optnetwork
   links         = mycas.LinkSetIn;
   clique
      out        = mycas.Cliques
      maxCliques = all;
run;

proc sql;
   create table CliqueSizes as
   select clique, count(*) as size
   from mycas.Cliques
   group by clique
   order by size desc;
quit;
```

The output data table mycas.Cliques now contains the maximal cliques of the input graph, as shown in Figure 3.20.

Figure 3.20 Maximal Cliques of a Simple Undirected Graph

clique	node
1	0
1	2
1	3
1	4
1	1
2	0
2	2
2	5
2	6
3	2
3	8
3	7
4	8
4	9

In addition, the output data table mycas.CliqueSizes contains the number of nodes in each clique, as shown in Figure 3.21.

Figure 3.21 Sizes of Maximal Cliques of a Simple Undirected Graph

clique	size
1	5
2	4
3	3
4	2

The maximal cliques are shown graphically in Figure 3.22 and Figure 3.23.

Figure 3.22 Maximal Cliques C^1 and C^2

$C^1 = \{0, 1, 2, 3, 4\}$ $C^2 = \{0, 2, 5, 6\}$

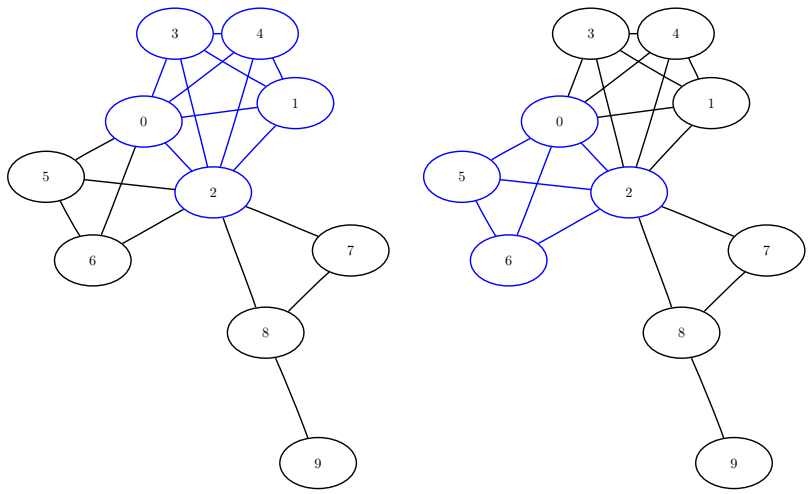
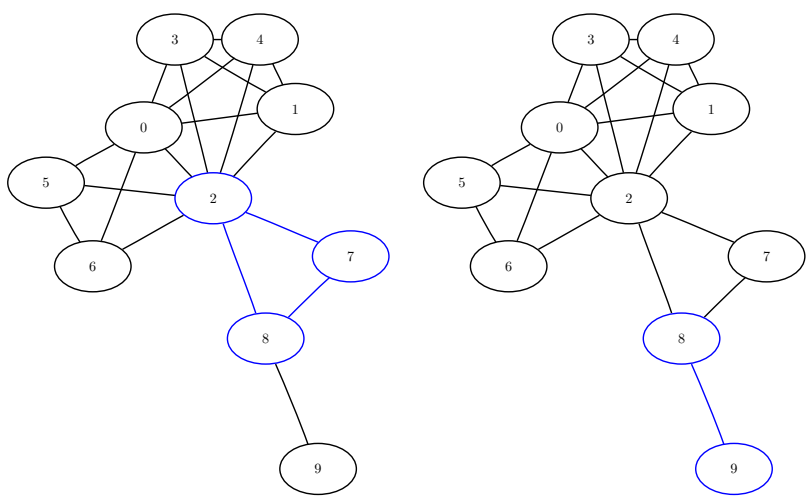


Figure 3.23 Maximal Cliques C^3 and C^4

$C^3 = \{2, 7, 8\}$ $C^4 = \{8, 9\}$



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 is 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 PROC OPTNETWORK, you can invoke connected components by using the CONNECTEDCOMPONENTS statement. The options for this statement are described in the section "CONNECTEDCOMPONENTS Statement" on page 27.

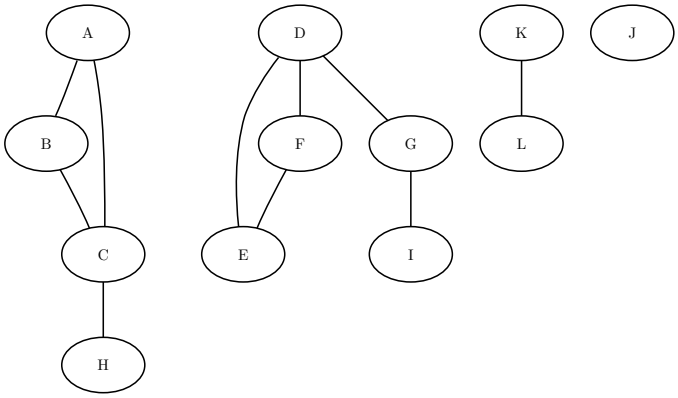
There are three algorithms for finding connected components in an undirected graph: a depth-first search algorithm (ALGORITHM=DFS), a union-find algorithm (ALGORITHM=UNIONFIND), and a distributed parallel union-find algorithm (ALGORITHM=PARALLEL). For a graph $G = (N, A)$, each algorithm runs in time $O(|N| + |A|)$ and can usually scale to very large graphs. The default is the sequential union-find algorithm (ALGORITHM=UNIONFIND). For directed graphs, only the depth-first search algorithm (ALGORITHM=DFS) is available.

The results of the connected components algorithm are written to the output nodes data table that you specify in the OUTNODES= option in the PROC OPTNETWORK statement and the output links data table that you specify in the OUTLINKS= option in the PROC OPTNETWORK statement. For each node in the nodes data table (or link in the links data table), the variable concomp identifies its component. The component identifiers are numbered sequentially, starting from the value of the INDEXOFFSET= option in the PROC OPTNETWORK statement.

Connected Components of a Simple Undirected Graph

This section illustrates the use of the connected components algorithm on the simple undirected graph G shown in Figure 3.24.

Figure 3.24 A Simple Undirected Graph G



The undirected graph G can be represented by the following links data table, mycas.LinkSetIn:

```
data mycas.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 find the connected components and output the results in the data table mycas.NodeSetOut:

```
proc optnetwork
   links    = mycas.LinkSetIn
   outNodes = mycas.NodeSetOut;
   connectedComponents;
run;
```

The output data table mycas.NodeSetOut contains the connected components of the input graph, as shown in Figure 3.25.

Figure 3.25 Connected Components of a Simple Undirected Graph

node	concomp
B	1
C	1
A	1
H	1
D	2
G	2
E	2
I	2
F	2
K	3
L	3

Notice that you define the graph by using only the links data table. As seen in Figure 3.24, this graph also contains a singleton node labeled J, which has no associated links. By definition, this node defines its own component. But because you define the input graph by using only the links data table, it does not show up in the results data table. To define a graph by using nodes that have no associated links, you should also define the input nodes data table. In this case, define the nodes data table mycas.NodeSetIn as follows:

```
data mycas.NodeSetIn;
   input node $ @@;
   datalines;
A  B  C  D  E  F  G  H  I  J  K  L
;
```

Now, when you find the connected components, you define the input graph by using both the nodes input data table and the links input data table:

```
proc optnetwork
   nodes    = mycas.NodeSetIn
   links    = mycas.LinkSetIn
   outNodes = mycas.NodeSetOut;
   connectedComponents;
run;
```

The resulting data table, mycas.NodeSetOut, includes the singleton node J as its own component, as shown in Figure 3.26.

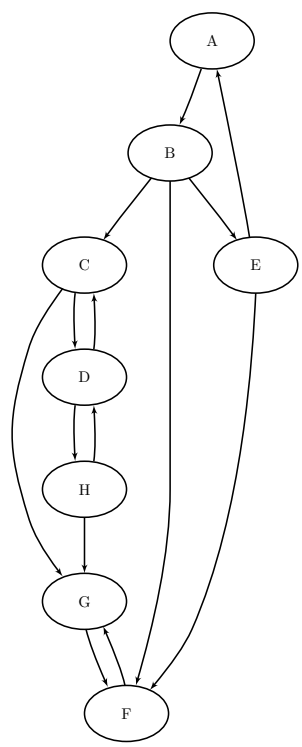
Figure 3.26 Connected Components of a Simple Undirected Graph

node	concomp
H	1
A	1
B	1
C	1
D	2
E	2
I	2
F	2
G	2
J	3
L	4
K	4

Connected Components of a Simple Directed Graph

This section illustrates the use of the connected components algorithm on the simple directed graph G shown in Figure 3.27.

Figure 3.27 A Simple Directed Graph G



The directed graph G can be represented by the following links data table, mycas.LinkSetIn:

```
data mycas.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 find the connected components and output the results in the data table mycas.NodeSetOut:

```
proc optnetwork
   direction = directed
   links     = mycas.LinkSetIn
   outNodes  = mycas.NodeSetOut;
   connectedComponents;
run;
```

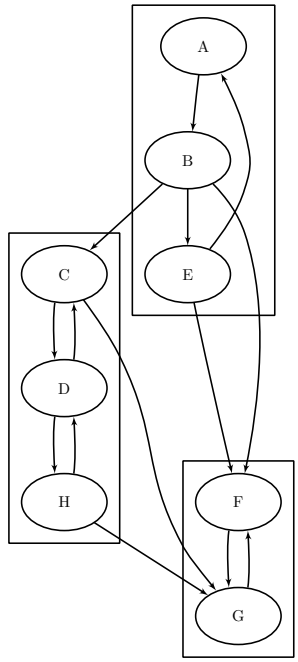
The output data table mycas.NodeSetOut, shown in Figure 3.28, now contains the connected components of the input graph.

Figure 3.28 Connected Components of a Simple Directed Graph

node	concomp
B	1
E	1
A	1
D	2
C	2
H	2
F	3
G	3

The connected components are represented graphically in Figure 3.29.

Figure 3.29 Strongly Connected Components of Graph G



Cycle Enumeration

A *path* in a graph is a sequence of links such that the *to* node of each link is the *from* node of the next link. An *elementary cycle* is a path in which the starting node and the ending node are the same and no node appears more than twice in the sequence.

In PROC OPTNETWORK, you can find (or just count) the elementary cycles of an input graph by specifying the CYCLE statement. The options for this statement are described in the section "CYCLE Statement" on page 28. To find the cycles and report them in an output data table, use the OUT= option. To simply count the cycles, omit the OUT= option.

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 ($A \rightarrow B \rightarrow C \rightarrow A$ and $A \rightarrow C \rightarrow B \rightarrow A$).

The results of the cycle enumeration algorithm are written to the output data table that you specify in the OUT= option in the CYCLE statement. Each node of each cycle is listed in the OUT= data table along with the variable cycle to identify the cycle to which it belongs. The variable order defines the order (sequence) of the node in the cycle. The cycle identifiers are numbered sequentially, starting from the value of the INDEXOFFSET= option in the PROC OPTNETWORK statement.

The default algorithm that PROC OPTNETWORK uses to compute 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 $O((|N| + |A|)(c + 1))$, 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)

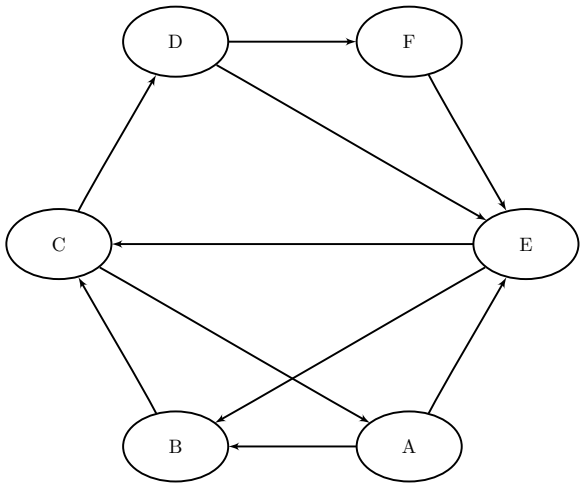
(ALGORITHM=BUILD). This algorithm is usually much faster than the backtracking algorithm when the length of the cycles is sufficiently restricted.

If MAXCYCLES=ALL and there are many cycles, the OUT= data table can become very large. It might be beneficial to check the number of cycles before you try to create the OUT= data table. By default (MAXCYCLES=1), 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= options might increase the computation time. For more information about these options, see the section "CYCLE Statement" on page 28.

Cycle Enumeration of a Simple Directed Graph

This section provides a simple example of using the cycle enumeration algorithm on the simple directed graph *G* shown in Figure 3.30. For a more detailed example involving both cycle enumeration and transitive closure, see "Example 3.5: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System" on page 129.

Figure 3.30 A Simple Directed Graph *G*



The directed graph *G* can be represented by the following links data table, mycas.LinkSetIn:

```
data mycas.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 count the number of cycles in the graph (without storing them):

```
proc optnetwork
   direction    = directed
   links        = mycas.LinkSetIn;
   cycle
      maxCycles = all;
run;
%put &_OROPTNETWORK_;
```

The result is written to the log of the OPTNETWORK procedure, as shown in Figure 3.31.

Figure 3.31 PROC OPTNETWORK Log: Count the Number of Cycles in a Simple Directed Graph

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: The number of nodes in the input graph is 6.
NOTE: The number of links in the input graph is 10.
NOTE: Processing cycle detection.
NOTE: The algorithm found 7 cycles.
NOTE: Processing cycle detection used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.024881 seconds.
STATUS=OK  PROBLEM_TYPE=CYCLE  SOLUTION_STATUS=OK  NUM_CYCLES=7  CPU_TIME=0.08  REAL_TIME=0.02
```

The following statements return the first cycle found in the graph:

```
proc optnetwork
   direction = directed
   links     = mycas.LinkSetIn;
   cycle
      out    = mycas.Cycles;
run;
```

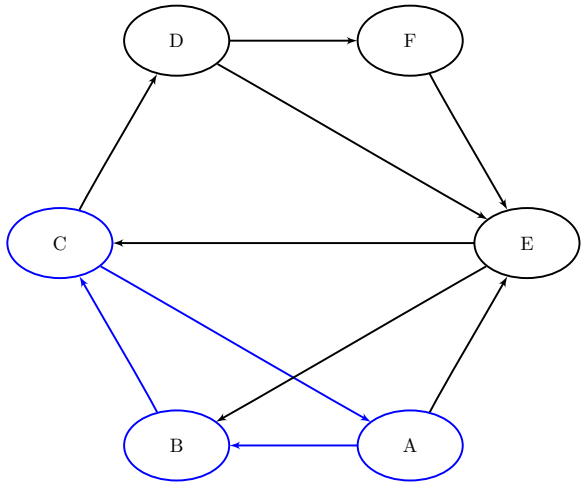
The output data table mycas.Cycles now contains the first cycle found in the input graph, as shown in Figure 3.32.

Figure 3.32 First Cycle Found in a Simple Directed Graph

cycle	order	node
1	1	A
1	2	B
1	3	C
1	4	A

The first cycle that is found in the input graph is shown graphically in Figure 3.33.

Figure 3.33 $A \rightarrow B \rightarrow C \rightarrow A$



The following statements return all the cycles in the graph:

```
proc optnetwork
   direction    = directed
   links        = mycas.LinkSetIn;
   cycle
      out       = mycas.Cycles
      maxCycles = all;
run;
```

The output data table mycas.Cycles now contains all the cycles in the input graph, as shown in Figure 3.34.

Figure 3.34 All Cycles in a Simple Directed Graph

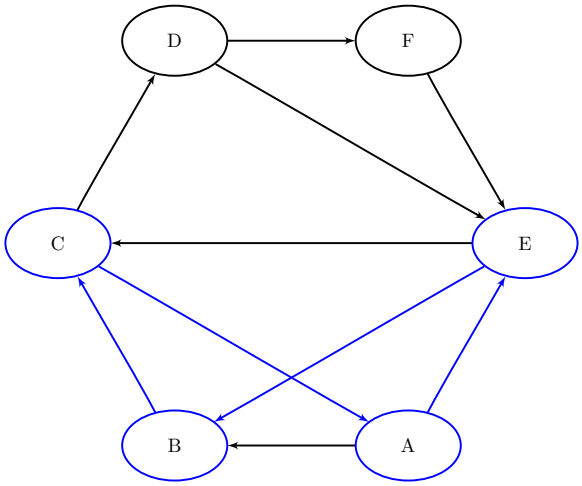
cycle	order	node
1	1	A
1	2	B
1	3	C
1	4	A
2	1	A
2	2	E
2	3	B
2	4	C
2	5	A
3	1	A
3	2	E
3	3	C
3	4	A
4	1	B
4	2	C
4	3	D
4	4	E

cycle	order	node
4	5	B
5	1	B
5	2	C
5	3	D
5	4	F
5	5	E
5	6	B
6	1	E
6	2	C
6	3	D
6	4	E
7	1	E
7	2	C
7	3	D
7	4	F
7	5	E

The six additional cycles are shown graphically in Figure 3.35 through Figure 3.37.

Figure 3.35 Cycles

$A \rightarrow E \rightarrow B \rightarrow C \rightarrow A$



$A \rightarrow E \rightarrow C \rightarrow A$

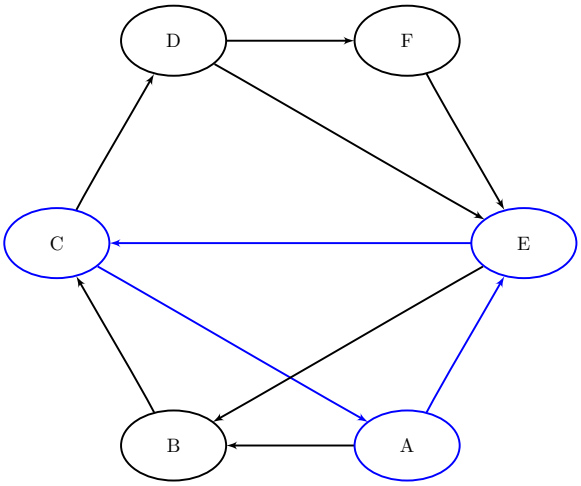
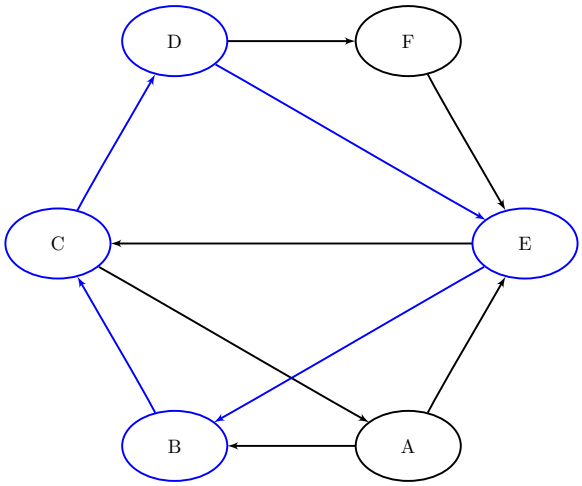


Figure 3.36 Cycles

$B \rightarrow C \rightarrow D \rightarrow E \rightarrow B$



$B \rightarrow C \rightarrow D \rightarrow F \rightarrow E \rightarrow B$

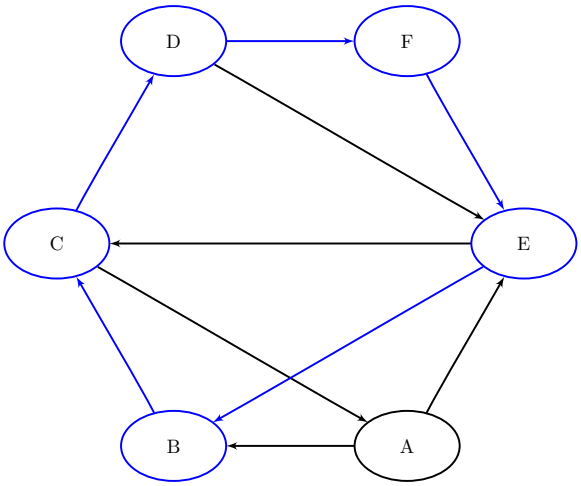
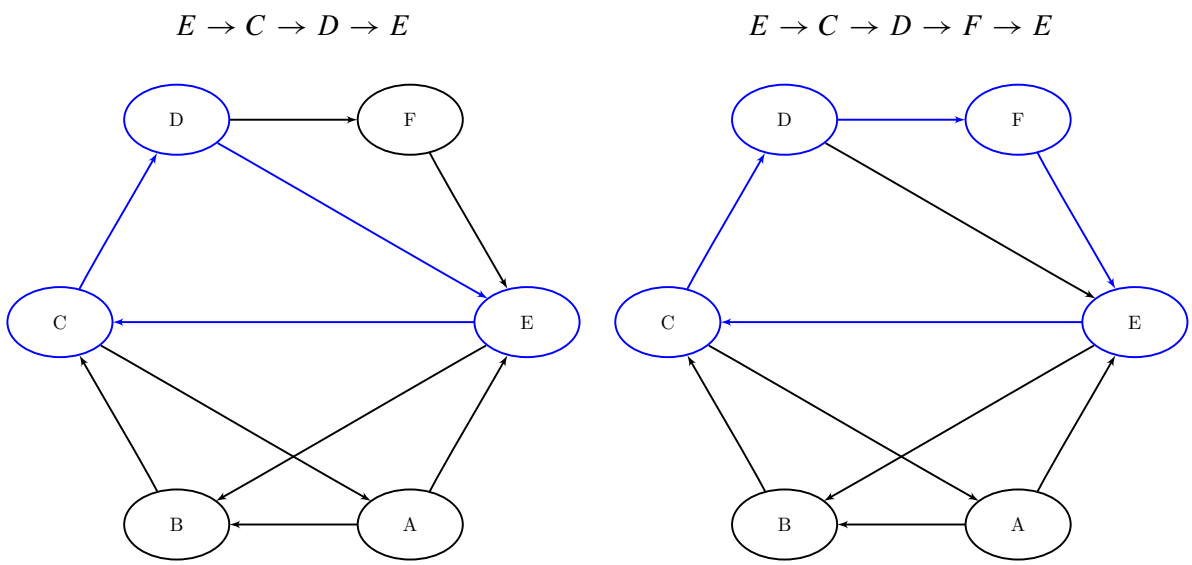


Figure 3.37 Cycles



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, 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 in one group to an abstract object in 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:

$$\begin{aligned}
\text{minimize} \quad & \sum_{(i,j)\in A} c_{ij} x_{ij} \\
\text{subject to} \quad & \sum_{(i,j)\in A} x_{ij} \leq 1 \quad i \in S \\
& \sum_{(i,j)\in A} x_{ij} = 1 \quad j \in T \\
& x_{ij} \in \{0,1\} \qquad (i,j) \in A
\end{aligned}$$

This model allows for some elements of set S (workers) to go unassigned (if $|S| > |T|$). However, if

$|S| < |T|$, then the following optimization problem is solved:

$$\begin{aligned}
\text{minimize} \quad & \sum_{(i,j)\in A} c_{ij} x_{ij} \\
\text{subject to} \quad & \sum_{(i,j)\in A} x_{ij} = 1 \quad i \in S \\
& \sum_{(i,j)\in A} x_{ij} \le 1 \quad j \in T \\
& x_{ij} \in \{0,1\} \qquad (i,j) \in A
\end{aligned}$$

This model allows for some elements of set T (tasks) to go unassigned.

In PROC OPTNETWORK, you can invoke the linear assignment problem solver by using the LINEARASSIGNMENT statement. The options for this statement are described in the section "LINEARASSIGNMENT Statement" on page 31. The algorithm that the PROC OPTNETWORK uses for solving a LAP is based on augmentation of shortest paths (Jonker and Volgenant 1987). This algorithm can be applied only to bipartite graphs. The resulting assignment (or matching) is contained in the output data table that you specify in the OUT= option in the LINEARASSIGNMENT statement.

For a detailed example, see "Example 3.3: Linear Assignment Problem for Minimizing Relay Times" on page 125.

Minimum-Cost Network Flow

The *minimum-cost network flow* (MCF) problem 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 as c_{ij}. The demand (or supply) at each node $i \in N$ is designated as b_i, where $b_i \ge 0$ denotes a supply node and $b_i < 0$ denotes a demand node. These values must be within $[b_i^l, b_i^u]$. 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:

$$\begin{aligned}
\text{minimize} \quad & \sum_{(i,j)\in A} c_{ij} x_{ij} \\
\text{subject to} \quad & b_i^l \le \sum_{(i,j)\in A} x_{ij} - \sum_{(j,i)\in A} x_{ji} \le b_i^u \quad i \in N \\
& l_{ij} \le x_{ij} \le u_{ij} \qquad (i,j) \in A
\end{aligned}$$

When $b_i = b_i^l = b_i^u$ 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 PROC OPTNETWORK, you can invoke the minimum-cost network flow solver by using the MINCOSTFLOW statement. The options for this statement are described in the section "MINCOSTFLOW Statement" on page 32.

The algorithm that PROC OPTNETWORK uses to solve the MCF problem 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 "Graph Input Data" on page 41. The links data table, which you specify in the LINKS= option in the PROC OPTNETWORK statement, contains the following columns:

- weight, which defines the link cost c_{ij}
- lower, which defines the link lower bound l_{ij}. The default is 0.
- upper, which defines the link upper bound u_{ij}. The default is ∞.

The nodes data table, which is specified in the NODES= option in the PROC OPTNETWORK statement, can contain the following columns:

- lower, which defines the node supply lower bound b_i^l. The default is 0.
- upper, which defines the node supply upper bound b_i^u. The default is ∞.

To define a standard network flow problem in which the node supply must be met exactly, use the lower variable only. You do not need to specify all the node supply bounds. For any missing node, the solver uses a lower and upper bound of 0.

To explicitly define an upper bound of ∞, use the special missing value (.I). To explicitly define a lower bound of $-\infty$, use the special missing value (.M). Infinite bounds are restricted as follows:

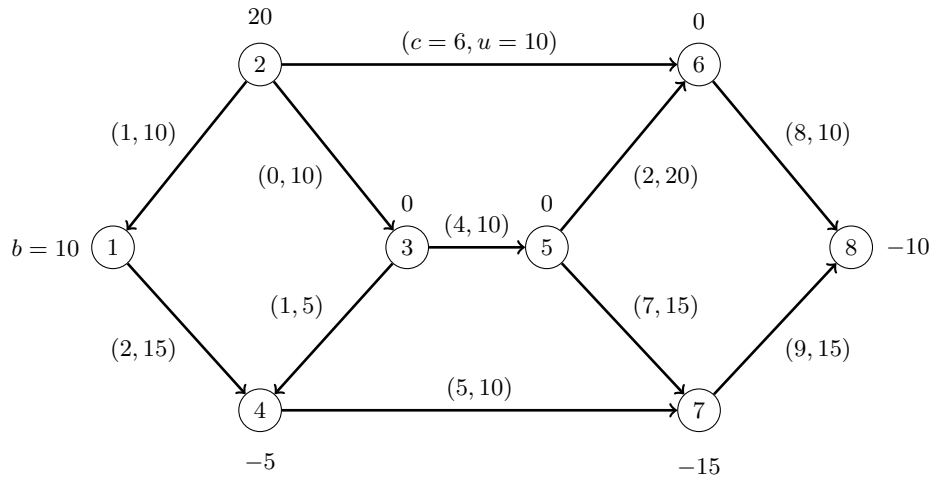
- The flow on a link must be bounded from below ($l_{ij} = -\infty$ is not allowed).
- Flow balance constraints cannot be *free* ($b_i^l = -\infty$ and $b_i^u = \infty$ is not allowed).

The resulting optimal flow through the network is written to the links output data table, which you specify in the OUTLINKS= option in the PROC OPTNETWORK statement.

Minimum-Cost Network Flow for a Simple Directed Graph

This 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 3.38, which appears in Ahuja, Magnanti, and Orlin (1993).

Figure 3.38 Minimum-Cost Network Flow Problem: Data



The directed graph G can be represented by the following links data table mycas.LinkSetIn and nodes data table mycas.NodeSetIn:

```
data mycas.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 mycas.NodeSetIn;
   input node lower;
   datalines;
1  10
2  20
4  -5
7 -15
8 -10
;
```

You can use the following call to PROC OPTNETWORK to find a minimum-cost flow:

```
proc optnetwork
   logLevel    = moderate
   direction   = directed
   links       = mycas.LinkSetIn
```

```
      nodes       = mycas.NodeSetIn
      outLinks    = mycas.LinkSetOut;
      minCostFlow
         logFreq  = 1;
   run;
   %put &_OROPTNETWORK_;
```

The progress of the procedure is shown in Figure 3.39.

Figure 3.39 PROC OPTNETWORK Log for Minimum-Cost Network Flow

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Reading the nodes data.
NOTE: Reading the links data.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: Building the input (full) graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 11.
NOTE: Processing the minimum-cost network flow problem.
NOTE: The network has 1 connected component.
                       Primal         Primal           Dual
      Iteration      Objective  Infeasibility  Infeasibility     Time
              1   0.000000E+00   2.000000E+01   8.900000E+01     0.00
              2   0.000000E+00   2.000000E+01   8.900000E+01     0.00
              3   5.000000E+00   1.500000E+01   8.400000E+01     0.00
              4   5.000000E+00   1.500000E+01   8.300000E+01     0.00
              5   5.000000E+00   1.500000E+01   8.300000E+01     0.00
              6   7.500000E+01   1.500000E+01   7.900000E+01     0.00
              7   1.300000E+02   1.000000E+01   7.600000E+01     0.00
              8   2.700000E+02   0.000000E+00   0.000000E+00     0.00
NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: Objective = 270.
NOTE: Processing the minimum-cost network flow problem used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.069513 seconds.
NOTE: The data set MYCAS.LINKSETOUT has 11 observations and 5 variables.
STATUS=OK  PROBLEM_TYPE=MINCOSTFLOW  SOLUTION_STATUS=OPTIMAL  OBJECTIVE=270  CPU_TIME=0.11
REAL_TIME=0.07
```

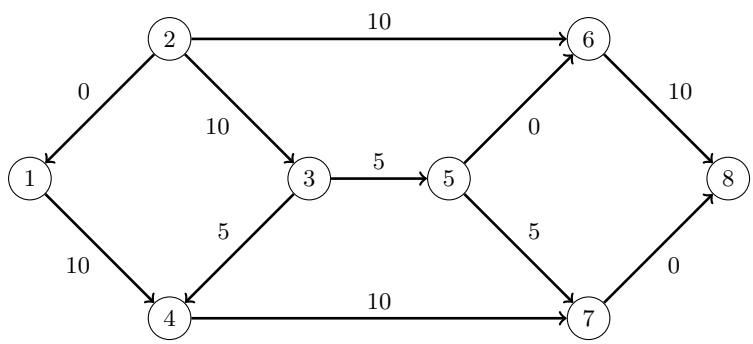
The optimal solution is displayed in Figure 3.40.

Figure 3.40 Minimum-Cost Network Flow Problem: Optimal Solution

Obs	from	to	weight	upper	mcf_flow
1	2	1	1	10	0
2	2	3	0	10	10
3	2	6	6	10	10
4	4	7	5	10	10
5	1	4	2	15	10
6	3	4	1	5	5
7	7	8	9	15	0
8	5	7	7	15	5
9	6	8	8	10	10
10	3	5	4	10	5
11	5	6	2	20	0

The optimal solution is represented graphically in Figure 3.41.

Figure 3.41 Minimum-Cost Network Flow Problem: Optimal Solution



Minimum-Cost Network Flow with Flexible Supply and Demand

Using the same directed graph shown in Figure 3.38, 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 table:

```
data mycas.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 OPTNETWORK to find a minimum-cost flow:

```
proc optnetwork
   logLevel    = moderate
   direction   = directed
   links       = mycas.LinkSetIn
   nodes       = mycas.NodeSetIn
   outLinks    = mycas.LinkSetOut;
   minCostFlow
      logFreq  = 1;
run;
%put &_OROPTNETWORK_;
```

The progress of the procedure is shown in Figure 3.42.

Figure 3.42 PROC OPTNETWORK Log for Minimum-Cost Network Flow

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Reading the nodes data.
NOTE: Reading the links data.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: Building the input (full) graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 11.
NOTE: Processing the minimum-cost network flow problem.
NOTE: The network has 1 connected component.
                        Primal         Primal           Dual
      Iteration      Objective  Infeasibility  Infeasibility     Time
              1   0.000000E+00   2.000000E+01   1.790000E+02     0.00
              2   1.000000E+01   2.000000E+01   1.730000E+02     0.00
              3   4.500000E+01   2.000000E+01   1.740000E+02     0.00
              4   4.500000E+01   2.000000E+01   1.740000E+02     0.00
              5   7.500000E+01   1.500000E+01   1.690000E+02     0.00
              6   7.500000E+01   1.500000E+01   1.690000E+02     0.00
              7   1.300000E+02   1.000000E+01   1.600000E+02     0.00
              8   1.300000E+02   1.000000E+01   8.700000E+01     0.00
              9   1.450000E+02   6.000000E+00   8.400000E+01     0.00
             10   1.450000E+02   6.000000E+00   1.660000E+02     0.00
             11   2.150000E+02   1.000000E+00   7.900000E+01     0.00
             12   2.150000E+02   1.000000E+00   7.900000E+01     0.00
             13   2.150000E+02   1.000000E+00   7.900000E+01     0.00
             14   2.340000E+02   0.000000E+00   8.000000E+00     0.00
             15   2.260000E+02   0.000000E+00   6.000000E+00     0.00
             16   2.260000E+02   0.000000E+00   0.000000E+00     0.00
             17   2.260000E+02   0.000000E+00   0.000000E+00     0.00
             18   2.260000E+02   0.000000E+00   0.000000E+00     0.00
NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: Objective = 226.
NOTE: Processing the minimum-cost network flow problem used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.038218 seconds.
NOTE: The data set MYCAS.LINKSETOUT has 11 observations and 5 variables.
STATUS=OK  PROBLEM_TYPE=MINCOSTFLOW  SOLUTION_STATUS=OPTIMAL  OBJECTIVE=226  CPU_TIME=0.11
REAL_TIME=0.04
```

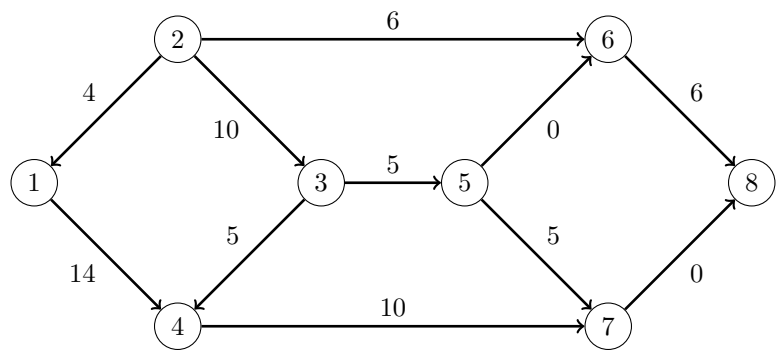
The optimal solution is displayed in Figure 3.43.

Figure 3.43 Minimum-Cost Network Flow Problem: Optimal Solution

Obs	from	to	weight	upper	mcf_flow
1	2	1	1	10	4
2	2	3	0	10	10
3	2	6	6	10	6
4	4	7	5	10	10
5	1	4	2	15	14
6	3	4	1	5	5
7	7	8	9	15	0
8	5	7	7	15	5
9	6	8	8	10	6
10	3	5	4	10	5
11	5	6	2	20	0

The optimal solution is represented graphically in Figure 3.44.

Figure 3.44 Minimum-Cost Network Flow Problem: Optimal Solution



Minimum Cut

A *cut* is a partition of the nodes of a graph into two disjoint subsets. A *cut set* is the set of links whose 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 PROC OPTNETWORK, you can invoke the minimum-cut algorithm by using the MINCUT statement. The options for this statement are described in the section "MINCUT Statement" on page 33. You can use this algorithm only on undirected graphs.

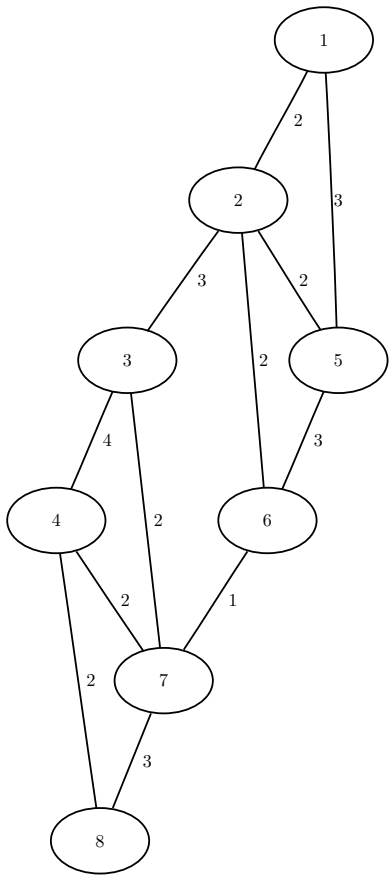
If the value of the MAXCUTS= option is greater than 1, then the algorithm can return more than one set of cuts. The resulting cuts can be described in terms of partitions of the nodes of the graph or the links in the cut sets. The node partition is specified by the partition variable, for each cut *i*, in the data table that you specify in the OUTPARTITIONS= option in the MINCUT statement. Each node is assigned the value 0 or 1, which defines the side of the partition to which it belongs. The cut set is defined in the output data table that you specify in the OUTCUTSETS= option in the MINCUT statement. This data table lists the links and their weights for each cut.

PROC OPTNETWORK 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 3.45.

Figure 3.45 A Simple Undirected Graph



The links data table can be represented as follows:

```
data mycas.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
;
```

The following statements calculate minimum cuts in the graph and output the results in the data tables mycas.CutSets and mycas.Partitions:

```
proc optnetwork
   logLevel        = moderate
   links           = mycas.LinkSetIn;
   minCut
     outCutSets    = mycas.CutSets
     outPartitions = mycas.Partitions
     maxCuts       = 3;
run;
%put &_OROPTNETWORK_;
```

The progress of the procedure is shown in Figure 3.46.

Figure 3.46 PROC OPTNETWORK Log for Minimum Cut

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Reading the links data.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: Building the input (full) graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 12.
NOTE: Processing the minimum-cut problem.
NOTE: The minimum-cut algorithm found 3 cuts.
NOTE: Objective = 4.
NOTE: Processing the minimum-cut problem used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.034648 seconds.
NOTE: The data set MYCAS.CUTSETS has 6 observations and 4 variables.
NOTE: The data set MYCAS.PARTITIONS has 24 observations and 3 variables.
STATUS=OK  PROBLEM_TYPE=MINCUT  SOLUTION_STATUS=OPTIMAL  OBJECTIVE=4  CPU_TIME=0.10
REAL_TIME=0.03
```

The output data table mycas.Partitions now contains the partition of the nodes and is shown by cut in Figure 3.47.

Figure 3.47 Minimum Cut Node Partition

cut=1

node	partition
2	1
6	1
1	1
8	0
5	1
4	0
7	0
3	0

Figure 3.47 *continued*

cut=2

node	partition
7	1
4	1
6	1
5	1
3	1
8	1
2	1
1	0

cut=3

node	partition
1	1
7	1
4	1
6	1
5	1
8	0
3	1
2	1

The output data table mycas.CutSets contains the links in the cut sets for each cut. This data table is shown by cut in Figure 3.48.

Figure 3.48 Minimum Cut Sets

cut=1

from	to	weight
6	7	1
2	3	3
cut		**4**

cut=2

from	to	weight
1	2	2
1	5	3
cut		**5**

cut=3

from	to	weight
7	8	3
4	8	2
cut		**5**
		14

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 PROC OPTNETWORK, you can invoke the minimum spanning tree algorithm by using the MINSPANTREE statement. The options for this statement are described in the section "MINSPANTREE Statement" on page 33. You can use this algorithm only on undirected graphs.

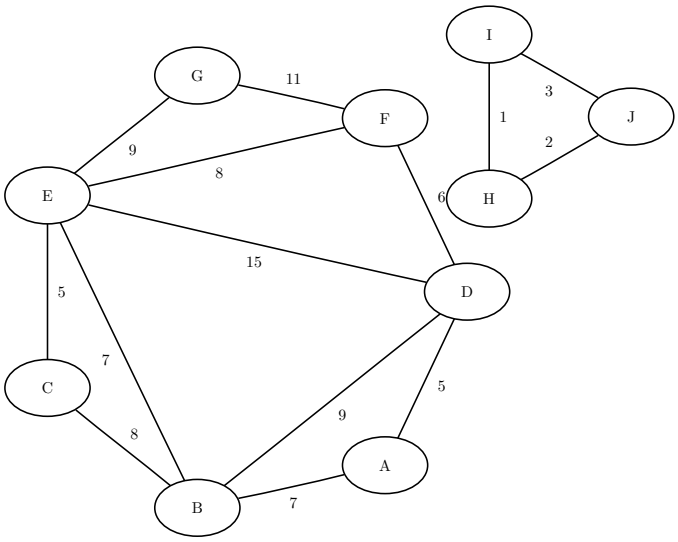
The resulting minimum spanning tree is contained in the output data table that you specify in the OUT= option in the MINSPANTREE statement.

PROC OPTNETWORK 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 3.49.

Figure 3.49 A Simple Undirected Graph



The links data table can be represented as follows:

```
data mycas.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 table mycas.MinSpanForest:

```
proc optnetwork
   links      = mycas.LinkSetIn;
   minSpanTree
      out     = mycas.MinSpanForest;
run;
```

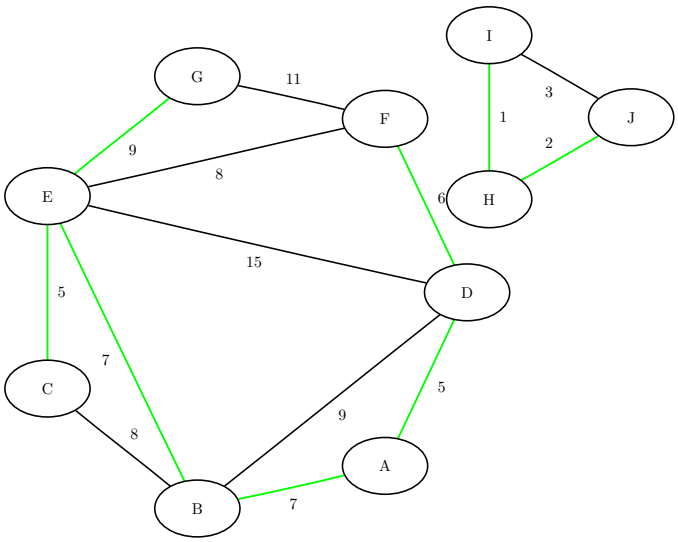
The output data table mycas.MinSpanForest now contains the links that belong to a minimum spanning forest, which is shown in Figure 3.50.

Figure 3.50 Minimum Spanning Forest

from	to	weight
H	I	1
H	J	2
A	D	5
C	E	5
D	F	6
B	E	7
A	B	7
E	G	9
		42

The minimal cost links are shown in green in Figure 3.51.

Figure 3.51 Minimum Spanning Forest



For a more detailed example, see "Example 3.4: Minimum Spanning Tree for Computer Network Design" on page 127.

Path Enumeration

A *path* in a graph is a sequence of links such that the *to* node of each link is the *from* node of the next link. An *elementary path* is a path in which no node appears more than twice in the sequence. A path between two nodes, *u* and *v*, in a graph is a path that starts at *u* and ends at *v*. The starting node is called the *source node*, and the ending node is called the *sink node*.

In PROC OPTNETWORK, you can find the elementary paths of an input graph by specifying the PATH statement. The options for this statement are described in the section "PATH Statement" on page 35.

By default, PROC OPTNETWORK finds paths for all pairs of nodes in the input graph. That is, it finds all paths for each possible combination of source nodes and sink nodes. Alternatively, you can use the SOURCE= option to fix a particular source node and find all paths from the fixed source node to all possible sink nodes. Conversely, by using the SINK= option, you can fix a sink node and find all paths from all possible source nodes to the fixed sink node. By using both options together, you can request all paths for a specific source-sink pair. In addition, you can use the NODESSUBSET= option to define a list of source-sink pairs to process, as described in the section "Nodes Subset Input Data" on page 45. The following section provides an example of how to use one of these options. Additional examples that show how to define the source-sink pairs of interest are found in the section "Shortest Path" on page 85.

For weighted graphs, the algorithm uses the weight variable that is defined in the links (nodes) data table to evaluate a path's total link (node) weight. You can also use the AUXWEIGHT= option in the LINKSVAR statement to define an auxiliary link weight.

Output Data Tables

The path enumeration algorithm produces up to two output data tables. The output data table that you specify in the OUTPATHSLINKS= option contains the links of the paths for each source-sink pair. The output data table that you specify in the OUTPATHSNODES= option contains the nodes of the paths for each source-sink pair.

OUTPATHSLINKS= Data Table

The OUTPATHSLINKS= data table contains the links present in each path. For large graphs and a large requested number of source-sink pairs, this output data table can be extremely large. Generating the output can sometimes take longer than computing the paths. This output data table is a distributed table when you are running on multiple machines. The only restriction is the total available cache disk space enabled by your configuration, as described in *SAS Cloud Analytic Services: Language Reference*.

The OUTPATHSLINKS= data table contains the following columns:

- source: the source node label of this path
- sink: the sink node label of this path
- path: for this source-sink pair, the path identifier of this path
- order: for this source-sink pair, the order of this link in this path
- from: the *from* node label of this link in this path
- to: the *to* node label of this link in this path

- weight: the weight of this link in this path
- *column*: the auxiliary weight of this link (if the AUXWEIGHT=*column* is defined in the LINKSVAR statement)

OUTPATHSNODES= Data Table

The OUTPATHSNODES= data table contains the nodes present in each path. This output data table can also be extremely large. This output data table is a distributed table when you are running on multiple machines.

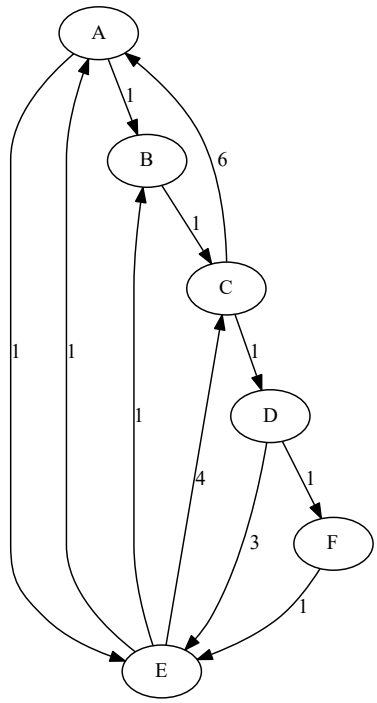
The OUTPATHSNODES= data table contains the following columns:

- source: the source node label of this path
- sink: the sink node label of this path
- path: for this source-sink pair, the path identifier of this path
- order: for this source-sink pair, the order of this node in this path
- node: the node label of this node in this path
- weight: the weight of this node in this path

Path Enumeration for One Source-Sink Pair

This section provides a simple example of using the path enumeration algorithm on the simple directed graph G shown in Figure 3.52 to find all paths between one source-sink pair by using the SOURCE= and SINK= options.

Figure 3.52 A Simple Directed Graph G



The directed graph G can be represented by the following links data table, mycas.LinkSetIn:

```
data mycas.LinkSetIn;
   input from $ to $ weight @@;
   datalines;
A B 1 A E 1 B C 1 C A 6 C D 1
D E 3 D F 1 E B 1 E C 4 F E 1
E A 1
;
```

The following statements find all paths between node D and node A whose path link weight is less than or equal to 10:

```
proc optnetwork
   direction         = directed
   links             = mycas.LinkSetIn;
   path
      source         = D
      sink           = A
      maxLinkWeight  = 10
      outPathsLinks  = mycas.PathLinks
      outPathsNodes  = mycas.PathNodes;
run;
```

The output data table mycas.PathLinks contains the links of the three paths from D to A whose path link weight is less than or equal to 10, as shown in Figure 3.53.

Figure 3.53 Links for All (Short) Paths in a Simple Directed Graph

source	sink	path	order	from	to	weight
D	A	1	1	D	E	3
D	A	1	2	E	A	1
D	A	2	1	D	F	1
D	A	2	2	F	E	1
D	A	2	3	E	A	1
D	A	3	1	D	F	1
D	A	3	2	F	E	1
D	A	3	3	E	B	1
D	A	3	4	B	C	1
D	A	3	5	C	A	6

The output data table mycas.PathNodes contains the nodes of the three paths, as shown in Figure 3.54.

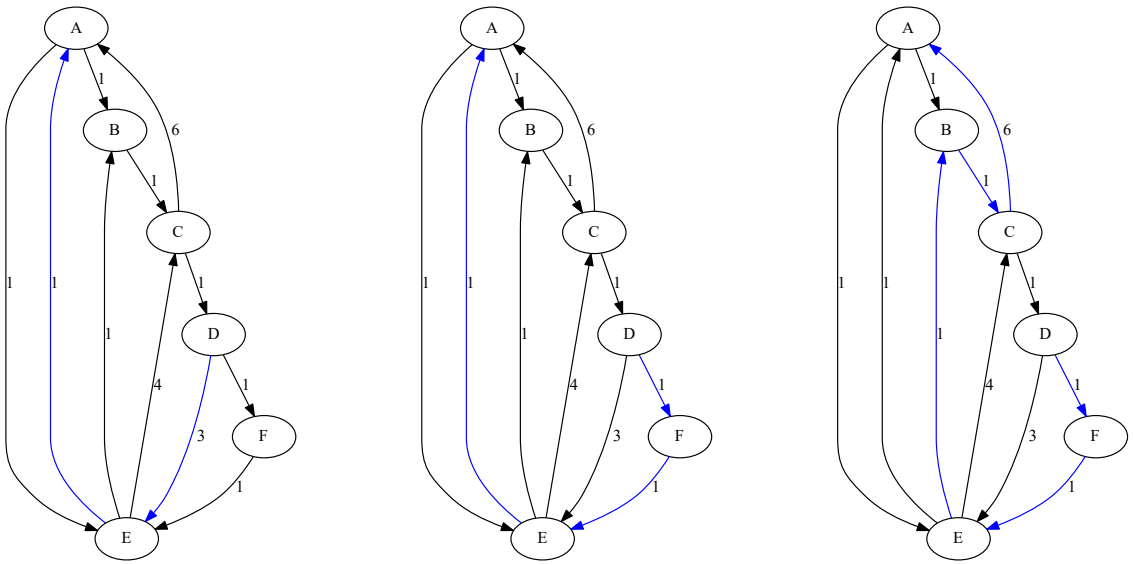
Figure 3.54 Nodes for All (Short) Paths in a Simple Directed Graph

source	sink	path	order	node
D	A	1	1	D
D	A	1	2	E
D	A	1	3	A
D	A	2	1	D
D	A	2	2	F
D	A	2	3	E
D	A	2	4	A
D	A	3	1	D
D	A	3	2	F
D	A	3	3	E
D	A	3	4	B
D	A	3	5	C
D	A	3	6	A

The three (short) paths are shown graphically in Figure 3.55.

Figure 3.55 Paths

$D \rightarrow E \rightarrow A$ $\quad$ $D \rightarrow F \rightarrow E \rightarrow A$ $\quad$ $D \rightarrow F \rightarrow E \rightarrow B \rightarrow C \rightarrow A$



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 PROC OPTNETWORK, you can find shortest paths by using the SHORTESTPATH statement. The options for this statement are described in the section "SHORTESTPATH Statement" on page 36.

By default, PROC OPTNETWORK finds shortest paths for all pairs of nodes in the input graph. That is, it finds a shortest path for each possible combination of source nodes and sink nodes. Alternatively, you can use the SOURCE= option 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= option, you can fix a sink node and find shortest paths from all possible source nodes to the fixed sink node. By using both options together, you can request one particular shortest path for a specific source-sink pair. In addition, you can use the NODESSUBSET= option to define a list of source-sink pairs to process, as described in the section "Nodes Subset Input Data" on page 45. The following sections show examples of how to use these options.

Which algorithm PROC OPTNETWORK uses to find shortest paths depends on the data. The algorithm and run-time complexity for each link type are shown in Table 3.7.

Table 3.7 Algorithms for Shortest Paths

Link Type	Algorithm	Complexity (per Source Node)
Unweighted	Breadth-first search	$O(\|N\| + \|A\|)$
Weighted (nonnegative)	Dijkstra's algorithm	$O(\|N\| \log \|N\| + \|A\|)$
Weighted (positive and negative allowed)	Bellman-Ford algorithm	$O(\|N\|\|A\|)$

You can find details for each algorithm in Ahuja, Magnanti, and Orlin (1993).

For weighted graphs, the algorithm uses the weight variable that is defined in the links data table to evaluate a path's total weight (cost). You can also use the AUXWEIGHT= option in the LINKSVAR statement to define an auxiliary weight. The auxiliary weight is not used in the algorithm to evaluate a path's total weight. It is calculated only for the sake of reporting the total auxiliary weight for each shortest path.

Output Data Tables

The shortest path algorithm produces up to two output data tables. The output data table that you specify in the OUTPATHS= option contains the links of a shortest path for each source-sink pair. The output data table that you specify in the OUTWEIGHTS= option contains the total weight for the shortest path for each source-sink pair.

OUTPATHS= Data Table

The OUTPATHS= data table contains the links present in each shortest path. For large graphs and a large requested number of source-sink pairs, this output data table can be extremely large. 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 data table of 11 GB. Finding shortest paths for all pairs from all nodes would produce an enormous output data table. This output data table is a distributed table when you are running on multiple machines. The only restriction is the total available cache disk space enabled by your configuration, as described in *SAS Cloud Analytic Services: Language Reference*. An example of finding the all-pairs shortest path for this road network is shown in "Example 3.8: Shortest Paths of the New York Road Network" on page 140.

The OUTPATHS= data table contains the following columns:

- source: the source node label of this shortest path
- sink: the sink node label of this shortest path
- order: for this source-sink pair, the order of this link in a shortest path
- from: the *from* node label of this link in a shortest path
- to: the *to* node label of this link in a shortest path
- weight: the weight of this link in a shortest path
- *column*: the auxiliary weight of this link (if the AUXWEIGHT=*column* is defined in the LINKSVAR statement)

OUTWEIGHTS= Data Table

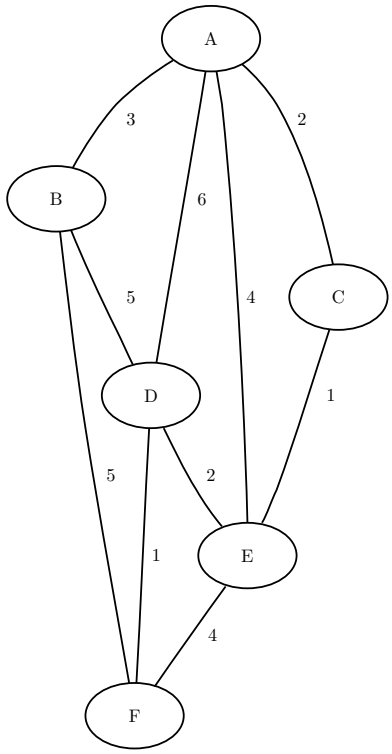
The OUTWEIGHTS= data table contains the total weight (and total auxiliary weight) of each shortest path. This data table contains the following columns:

- source: the source node label of this shortest path
- sink: the sink node label of this shortest path
- path_weight: the total weight of the shortest path for this source-sink pair
- path_auxweight: the total auxiliary weight of the shortest path for this source-sink pair (if you specify the AUXWEIGHT= option in the LINKSVAR statement)

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* shown in Figure 3.56.

Figure 3.56 A Simple Undirected Graph *G*



The undirected graph *G* can be represented by the following links data table, mycas.LinkSetIn:

```
data mycas.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 find shortest paths for all source-sink pairs:

```
proc optnetwork
   links         = mycas.LinkSetIn;
   shortestPath
      outWeights = mycas.ShortPathW
      outPaths   = mycas.ShortPathP;
run;
```

The output data table mycas.ShortPathP contains the shortest paths, as shown in Figure 3.57.

Figure 3.57 All-Pairs Shortest Paths

source	sink	order	from	to	weight
A	B	1	A	B	3
A	C	1	A	C	2
A	D	1	A	C	2
A	D	2	C	E	1
A	D	3	E	D	2
A	E	1	A	C	2
A	E	2	C	E	1
A	F	1	A	C	2
A	F	2	C	E	1
A	F	3	E	D	2
A	F	4	D	F	1
B	A	1	B	A	3
B	C	1	B	A	3
B	C	2	A	C	2
B	D	1	B	D	5
B	E	1	B	A	3
B	E	2	A	C	2
B	E	3	C	E	1
B	F	1	B	F	5
C	A	1	C	A	2
C	B	1	C	A	2
C	B	2	A	B	3
C	D	1	C	E	1
C	D	2	E	D	2
C	E	1	C	E	1
C	F	1	C	E	1
C	F	2	E	D	2
C	F	3	D	F	1
D	A	1	D	E	2
D	A	2	E	C	1
D	A	3	C	A	2
D	B	1	D	B	5
D	C	1	D	E	2
D	C	2	E	C	1
D	E	1	D	E	2
D	F	1	D	F	1
E	A	1	E	C	1
E	A	2	C	A	2
E	B	1	E	C	1
E	B	2	C	A	2
E	B	3	A	B	3
E	C	1	E	C	1
E	D	1	E	D	2
E	F	1	E	D	2
E	F	2	D	F	1
F	A	1	F	D	1
F	A	2	D	E	2
F	A	3	E	C	1
F	A	4	C	A	2
F	B	1	F	B	5
F	C	1	F	D	1
F	C	2	D	E	2
F	C	3	E	C	1
F	D	1	F	D	1
F	E	1	F	D	1
F	E	2	D	E	2

The output data table mycas.ShortPathW contains the path weights of the shortest paths of each source-sink pair, as shown in Figure 3.58.

Figure 3.58 All-Pairs Shortest Paths Summary

source	sink	path_weight
C	A	2
C	B	5
C	D	3
C	E	1
C	F	4
A	B	3
A	C	2
A	D	5
A	E	3
A	F	6
E	A	3
E	B	6
E	C	1
E	D	2
E	F	3

source	sink	path_weight
D	A	5
D	B	5
D	C	3
D	E	2
D	F	1
B	A	3
B	C	5
B	D	5
B	E	6
B	F	5
F	A	6
F	B	5
F	C	4
F	D	1
F	E	3

Shortest Paths for a Subset of Source-Sink Pairs

This section illustrates the use of a nodes subset data table, the NODESSUBSET= option, and the shortest path algorithm to find shortest paths for a subset of source-sink pairs. The data table variables source and sink are used as indicators to specify which pairs to process. The marked source nodes define a set S, and the marked sink nodes define a set T. PROC OPTNETWORK then calculates all the source-sink pairs in the crossproduct of these two sets.

For example, the following DATA step tells PROC OPTNETWORK to calculate the pairs in $S \times T = \{A, C\} \times \{B, F\}$:

```
data mycas.NodeSubSetIn;
   input node $ source sink;
   datalines;
A 1 0
C 1 0
B 0 1
F 0 1
;
```

The following statements find a shortest path for the four combinations of source-sink pairs:

```
proc optnetwork
   nodesSubset  = mycas.NodeSubSetIn
   links        = mycas.LinkSetIn;
   shortestPath
      outPaths  = mycas.ShortPath;
run;
```

The output data table mycas.ShortPath contains the shortest paths, as shown in Figure 3.59.

Figure 3.59 Shortest Paths for a Subset of Source-Sink Pairs

source	sink	order	from	to	weight
A	B	1	A	B	3
A	F	1	A	C	2
A	F	2	C	E	1
A	F	3	E	D	2
A	F	4	D	F	1
C	B	1	C	A	2
C	B	2	A	B	3
C	F	1	C	E	1
C	F	2	E	D	2
C	F	3	D	F	1

Shortest Paths for a Subset of Source or Sink Pairs

This section illustrates the use of the shortest path algorithm to find the shortest paths between a subset of source (or sink) nodes and all the other sink (or source) nodes.

In this case, you designate the subset of source (or sink) nodes in the nodes subset data table by specifying the source (or sink) variable. By specifying only one of the variables, you indicate that you want PROC OPTNETWORK to calculate all source-sink pairs from a subset of source nodes (or to calculate all source-sink pairs to a subset of sink nodes).

For example, the following DATA step designates nodes *B* and *E* as source nodes:

```
data mycas.NodeSubSetIn;
   input node $ source;
   datalines;
B 1
E 1
;
```

You can use the same PROC OPTNETWORK call that is used in the section "Shortest Paths for a Subset of Source-Sink Pairs" on page 89 to find all the shortest paths from nodes *B* and *E*. The output data table mycas.ShortPath contains the shortest paths, as shown in Figure 3.60.

Figure 3.60 Shortest Paths for a Subset of Source Pairs

source	sink	order	from	to	weight
B	A	1	B	A	3
B	C	1	B	A	3
B	C	2	A	C	2
B	D	1	B	D	5
B	E	1	B	A	3
B	E	2	A	C	2
B	E	3	C	E	1
B	F	1	B	F	5
E	A	1	E	C	1
E	A	2	C	A	2
E	B	1	E	C	1
E	B	2	C	A	2
E	B	3	A	B	3
E	C	1	E	C	1
E	D	1	E	D	2
E	F	1	E	D	2
E	F	2	D	F	1

Conversely, the following DATA step designates nodes *B* and *E* as sink nodes:

```
data mycas.NodeSubSetIn;
   input node $ sink;
   datalines;
B 1
E 1
;
```

You can use the same PROC OPTNETWORK call again to find all the shortest paths to nodes *B* and *E*. The output data table mycas.ShortPath contains the shortest paths, as shown in Figure 3.61.

Figure 3.61 Shortest Paths for a Subset of Sink Pairs

source	sink	order	from	to	weight
C	B	1	C	A	2
C	B	2	A	B	3
C	E	1	C	E	1
A	B	1	A	B	3
A	E	1	A	C	2
A	E	2	C	E	1
E	B	1	E	C	1
E	B	2	C	A	2
E	B	3	A	B	3
D	B	1	D	B	5
D	E	1	D	E	2
B	E	1	B	A	3
B	E	2	A	C	2
B	E	3	C	E	1
F	B	1	F	B	5
F	E	1	F	D	1
F	E	2	D	E	2

Shortest Paths for One Source-Sink Pair

This section illustrates the use of the shortest path algorithm to find the shortest paths between one source-sink pair by using the SOURCE= and SINK= options.

The following statements find a shortest path between node *C* and node *F*:

```
proc optnetwork
   links        = mycas.LinkSetIn;
   shortestPath
      source    = C
      sink      = F
      outPaths  = mycas.ShortPath;
run;
```

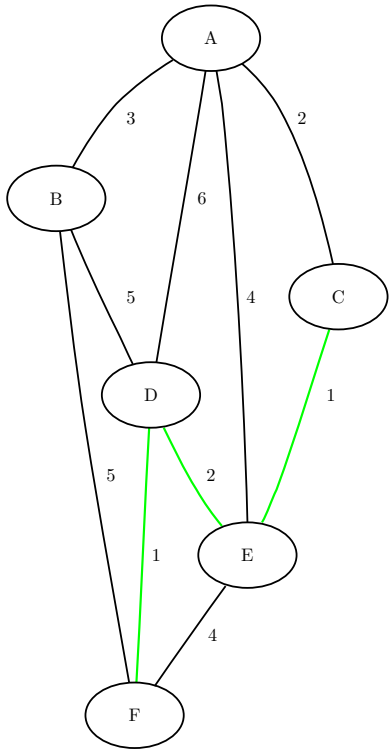
The output data table mycas.ShortPath contains this shortest path, as shown in Figure 3.62.

Figure 3.62 Shortest Paths for One Source-Sink Pair

source	sink	order	from	to	weight
C	F	1	C	E	1
C	F	2	E	D	2
C	F	3	D	F	1

The shortest path is shown graphically in Figure 3.63.

Figure 3.63 Shortest Path between Nodes C and F



Shortest Paths with Auxiliary Weight Calculation

This section illustrates the use of the shortest path algorithm with auxiliary weights to find the shortest paths between all source-sink pairs.

Consider a links data table in which the auxiliary weight is a counter for each link:

```
data mycas.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 find the shortest paths for all source-sink pairs:

```
proc optnetwork
   links         = mycas.LinkSetIn;
   linksVar
      auxWeight  = count;
   shortestPath
      outWeights = mycas.ShortPathW;
run;
```

The output data table mycas.ShortPathW contains the total path weight of shortest paths in each source-sink pair, as shown in Figure 3.64. Because the variable count in mycas.LinkSetIn has a value of 1 for all links, the value in the output data table variable path_auxweight contains the number of links in each shortest path.

Figure 3.64 Shortest Paths Including Auxiliary Weights in Calculation

source	sink	path_weight	path_auxweight
C	A	2	1
C	B	5	2
C	D	3	2
C	E	1	1
C	F	4	3
A	B	3	1
A	C	2	1
A	D	5	3
A	E	3	2
A	F	6	4
E	A	3	2
E	B	6	3
E	C	1	1
E	D	2	1
E	F	3	2

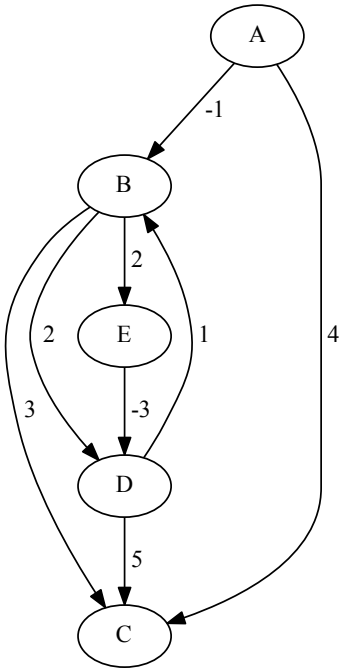
source	sink	path_weight	path_auxweight
D	A	5	3
D	B	5	1
D	C	3	2
D	E	2	1
D	F	1	1
B	A	3	1
B	C	5	2
B	D	5	1
B	E	6	3
B	F	5	1
F	A	6	4
F	B	5	1
F	C	4	3
F	D	1	1
F	E	3	2

The section "Road Network Shortest Path" on page 13 shows an example of using the shortest path algorithm to minimize travel time 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 3.65.

Figure 3.65 A Simple Directed Graph *G* with Negative Link Weights



You can represent the directed graph *G* by using the following links data table, mycas.LinkSetIn:

```
data mycas.LinkSetIn;
   input from $ to $ weight @@;
   datalines;
A B -1  A C  4  B C  3  B D  2  B E  2
D B  1  D C  5  E D -3
;
```

The following statements find a shortest path between the source node *E* and the sink node *B*:

```
proc optnetwork
   direction    = directed
   links        = mycas.LinkSetIn;
   shortestPath
      source    = E
      sink      = B
      outPaths  = mycas.ShortPathP;
run;
```

The output data table mycas.ShortPathP contains a shortest path from node *E* to node *B*, as shown in Figure 3.66.

Figure 3.66 Shortest Paths with Negative Link Weights

source	sink	order	from	to	weight
E	B	1	E	D	-3
E	B	2	D	B	1

Now, consider the following adjustment to the weight of link (B, E):

```
data mycas.LinkSetIn;
   set mycas.LinkSetIn;
   if(from="B" and to="E") then
      weight=1;
run;
```

In this case, there is a negative weight cycle $(E \rightarrow D \rightarrow B \rightarrow E)$. The Bellman-Ford algorithm catches the cycle and produces an error message, as shown in Figure 3.67.

Figure 3.67 PROC OPTNETWORK Log: Negative Weight Cycle

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 8.
NOTE: Processing the shortest paths problem using 32 threads on each of 4 machines.
NOTE: Processing the shortest paths problem between 1 source nodes and 1 sink nodes.
ERROR: The graph contains a negative weight cycle.
NOTE: Processing the shortest paths problem used 0.00 (cpu: 0.00) seconds.
ERROR: The action stopped due to errors.
NOTE: The Cloud Analytic Services server processed the request in 0.038334 seconds.
NOTE: The SAS System stopped processing this step because of errors.
STATUS=ERROR  PROBLEM_TYPE=SHORTESTPATH  CPU_TIME=0.11  REAL_TIME=0.04
```

Summary Statistics

In PROC OPTNETWORK, you can calculate various summary statistics for a graph and its nodes by using the SUMMARY statement. The options for this statement are described in the section "SUMMARY Statement" on page 37.

Output Data Tables

The summary statistics that PROC OPTNETWORK produces are divided into two categories: statistics on the entire graph and statistics on the nodes and links of the graph. The latter statistics are appended to the output nodes and links data tables that you specify in the OUTNODES= and OUTLINKS= option in the PROC OPTNETWORK statement. The former statistics are contained in the data table that you specify in the OUT= option in the SUMMARY statement.

Let $\delta(u)$ represent the list of nodes that are connected to node u in an undirected graph. In a directed graph, $\delta^{\text{out}}(u)$ represents the list of nodes that are connected *from* node u (out-links), and $\delta^{\text{in}}(u)$ represents the list of nodes that are connected *to* node u (in-links).

OUT= Data Table

By default, the summary output data table that you specify in the OUT= option in the SUMMARY statement contains the following columns:

- nodes: the number of nodes in the graph ($|N|$)
- links: the number of links in the graph ($|A|$)
- avg_links_per_node: the average number of links per node
- density: the number of links in the graph divided by the number of links in a complete graph $\left(\frac{|A|}{|K(N)|}\right)$
- self_links_ignored: the number of self-links that are ignored
- dup_links_ignored: the number of duplicate links that are ignored
- leaf_nodes: the number of leaf nodes
  - undirected graph: $u \in N$ such that $\delta(u) = 1$
  - directed graph: $u \in N$ such that $\delta^{\text{out}}(u) = 0$ and $\delta^{\text{in}}(u) > 0$
- singleton_nodes: the number of singleton nodes
  - undirected graph: $u \in N$ such that $\delta(u) = 0$
  - directed graph: $u \in N$ such that $\delta^{\text{out}}(u) + \delta^{\text{in}}(u) = 0$

You can produce statistics about the connectedness of the graph by using the CONNECTEDCOMPONENTS and BICONNECTEDCOMPONENTS options. For more information about connected components and biconnected components, see the sections "Connected Components" on page 58 and "Biconnected Components and Articulation Points" on page 51, respectively. If you use the CONNECTEDCOMPONENTS or BICONNECTEDCOMPONENTS option, the following columns might also appear in the summary output data table for undirected graphs:

- concomp: the number of connected components in the graph
- biconcomp: the number of biconnected components in the graph
- artpoints: the number of articulation points in the graph
- isolated_pairs: the number of isolated pairs of nodes (a connected component of size 2)
- isolated_stars: the number of isolated stars (a connected component, C, of size greater than 2 with):
  - one node i with $\delta(i) = |C| - 1$ and all other nodes $u \in C \setminus \{i\}$ with $\delta(u) = 1$

The following columns appear for directed graphs:

- concomp: the number of strongly connected components in the graph
- isolated_pairs: the number of isolated pairs of nodes (a weakly connected component of size 2)

- isolated_stars_out: the number of isolated outward stars (a weakly connected component, C, of size greater than 2 with):
  - one node i with $\delta^{\text{out}}(i) = |C| - 1$ and all other nodes $u \in C \setminus \{i\}$ with $\delta^{\text{in}}(u) = 1$
- isolated_stars_in: the number of isolated inward stars (a weakly connected component, C, of size greater than 2 with):
  - one node i with $\delta^{\text{in}}(i) = |C| - 1$ and all other nodes $u \in C \setminus \{i\}$ with $\delta^{\text{out}}(u) = 1$

You can produce statistics about the shortest paths in the graph by using the SHORTESTPATH= option. The *diameter* of a graph is the longest possible shortest path distance of all source-sink pairs that the graph can contain. For more information about shortest paths, see the section "Shortest Path" on page 85. If you use the SHORTESTPATH= option, the following columns also appear in the summary output data table:

- diameter_wt: the longest weighted shortest path in the graph
- diameter_unwt: the longest unweighted shortest path in the graph
- avg_shortpath_wt: the average weighted shortest path in the graph
- avg_shortpath_unwt: the average unweighted shortest path in the graph

Calculating the diameter of a graph is computationally expensive, because it involves calculating shortest paths for all pairs. For undirected graphs, an approximate method is available based on Boitmanis et al. (2006). You can invoke the algorithm by using the DIAMETERAPPROX= option. The exact method runs in time $O(|N| \times (|N| \log |N| + |A|))$; the approximate method runs in time $O(|A|\sqrt{|N|})$ with an additive error of $O(\sqrt{|N|})$. If you use the DIAMETERAPPROX= option, the following columns also appear in the summary output data table:

- diameter_approx_wt: the approximate longest weighted shortest path in the graph
- diameter_approx_unwt: the approximate longest unweighted shortest path in the graph

OUTNODES= Data Table

In addition, you can produce summary statistics about the nodes of the graph. By default, the following columns are appended to the data table that you specify in the OUTNODES= option in the PROC OPTNETWORK statement:

- sum_in_and_out_wt: the sum of the link weights from and to the node
- leaf_node: 1, if the node is a leaf node; otherwise, 0
- singleton_node: 1, if the node is a singleton node; otherwise, 0
- isolated_pair: the identifier, if the node is in an isolated pair; otherwise, missing (.)
- neighbor_leaf_nodes: the number of leaf nodes connected to the node

You can produce statistics about the connectedness of the graph by using the CONNECTEDCOMPONENTS and BICONNECTEDCOMPONENTS options. If you use these options, the following column also appears in the nodes output data table for undirected graphs:

- isolated_star: the identifier, if the node is in an isolated star; otherwise, missing (.)

The following columns also appear for directed graphs:

- isolated_star_out: the identifier, if the node is in an isolated outward star; otherwise, missing (.)
- isolated_star_in: the identifier, if the node is in an isolated inward star; otherwise, missing (.)

You can produce statistics about the shortest path distances to and from nodes in the graph by using the SHORTESTPATH= option. The *eccentricity* of a node u is the longest of all possible shortest path distances between u and any other node. If you use the SHORTESTPATH= option, the following columns also appear in the nodes output data table for undirected graphs:

- eccentr_out_wt: the longest weighted shortest path distance from the node
- eccentr_out_unwt: the longest unweighted shortest path distance from the node

The following columns also appear for directed graphs:

- eccentr_in_wt: the longest weighted shortest path distance to the node
- eccentr_in_unwt: the longest unweighted shortest path distance to the node

OUTLINKS= Data Table

In addition, you can produce summary statistics about the connectedness of the links of the graph. If you use the CONNECTEDCOMPONENTS or BICONNECTEDCOMPONENTS option, the following columns are appended to the data table that you specify in the OUTLINKS= option in the PROC OPTNETWORK statement for undirected graphs:

- isolated_pair: the identifier, if the link is in an isolated pair; otherwise, missing (.)
- isolated_star: the identifier, if the link is in an isolated star; otherwise, missing (.)

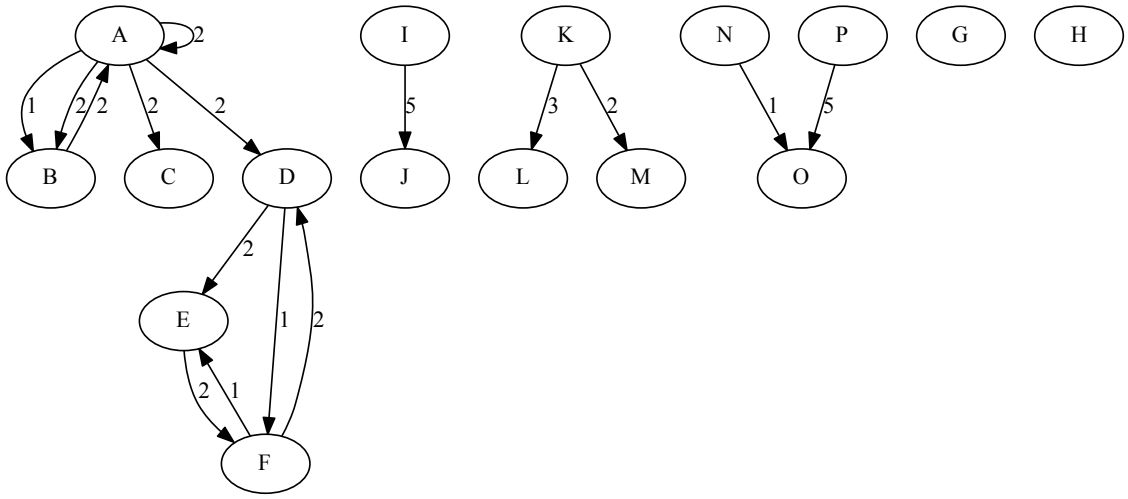
The following columns are appended for directed graphs:

- isolated_star_out: the identifier, if the link is in an isolated outward star; otherwise, missing (.)
- isolated_star_in: the identifier, if the link is in an isolated inward star; otherwise, missing (.)

Summary Statistics of a Simple Directed Graph

This section illustrates the calculation of summary statistics on the simple directed graph *G* shown in Figure 3.68.

Figure 3.68 A Simple Directed Graph *G*



You can represent the directed graph *G* by using the following nodes data table, mycas.NodeSetIn, and links data table, mycas.LinkSetIn:

```
data mycas.NodeSetIn;
  input node $ @@;
  datalines;
A B C D E F G H I J K L M N O P
;
data mycas.LinkSetIn;
   input from $ to $ weight @@;
   datalines;
A B 1 A C 2 A D 2 B A 2 D E 2
D F 1 E F 2 F D 2 F E 1 A A 2
A B 2 I J 5 K L 3 K M 2 N O 1
P O 5
;
```

The following statements calculate the default summary statistics and output the results in the data table mycas.Summary:

```
proc optnetwork
   direction = directed
   nodes     = mycas.NodeSetIn
   links     = mycas.LinkSetIn;
   summary
      out    = mycas.Summary;
run;
```

The output data table mycas.Summary contains the default summary statistics of the input graph, as shown in Figure 3.69.

Figure 3.69 Graph Summary Statistics of a Simple Directed Graph

nodes	links	avg_links_per_node	density	self_links_ignored	dup_links_ignored	leaf_nodes	singleton_nodes
16	14	0.875	0.058333	1	1	5	2

The following statements calculate the default summary statistics and produce information about the connectedness of the graph. They output the results in the data table mycas.Summary.

```
proc optnetwork
   direction  = directed
   nodes      = mycas.NodeSetIn
   links      = mycas.LinkSetIn;
   summary
      connectedComponents
      out     = mycas.Summary;
run;
```

The output data table mycas.Summary contains the summary statistics of the input graph, as shown in Figure 3.70.

Figure 3.70 Graph Summary and Connectedness Statistics of a Simple Directed Graph

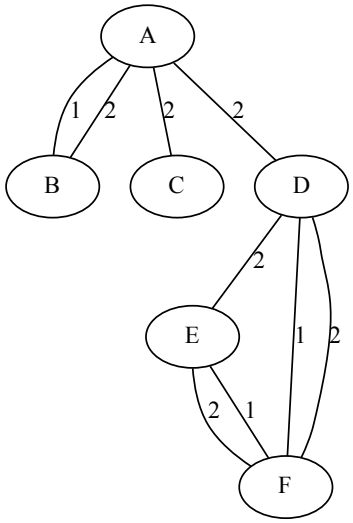
nodes	links	avg_links_per_node	density	self_links_ignored	dup_links_ignored	leaf_nodes	singleton_nodes
16	14	0.875	0.058333	1	1	5	2

concomp	isolated_pairs	isolated_stars_out	isolated_stars_in
13	1	1	1

Summary Statistics of a Simple Undirected Graph

This section illustrates the calculation of summary and shortest path statistics on the simple undirected graph *G* shown in Figure 3.71.

Figure 3.71 A Simple Undirected Graph G



You can represent the undirected graph G by using the following links data table, mycas.LinkSetIn:

```
data mycas.LinkSetIn;
   input from $ to $ weight @@;
   datalines;
A B 1 A C 2 A D 2 B A 2 D E 2
D F 1 E F 2 F D 2 F E 1
;
```

The following statements calculate the default summary statistics and produce information about the shortest path distances of the graph. They output the results in the data table mycas.Summary. In addition, node statistics are produced and output in the data table mycas.NodeSetOut.

```
proc optnetwork
   links           = mycas.LinkSetIn
   outNodes        = mycas.NodeSetOut;
   summary
      out          = mycas.Summary
      shortestPath = weight;
run;
```

The output data tables mycas.Summary and mycas.NodeSetOut now contain the summary statistics of the input graph, as shown in Figure 3.72.

Figure 3.72 Graph Summary and Shortest Path Statistics of a Simple Undirected Graph

nodes	links	avg_links_per_node	density	self_links_ignored	dup_links_ignored	leaf_nodes	singleton_nodes	diameter_wt	avg_shortpath_wt
6	6	1	0.4	0	3	2	0	6	3.06667

Figure 3.72 *continued*

node	leaf_node	singleton_node	neighbor_leaf_nodes	sum_in_and_out_wt	eccentr_wt_out
A	0	0	2	5	4
B	1	0	0	1	5
C	1	0	0	2	6
D	0	0	0	5	4
E	0	0	0	3	6
F	0	0	0	2	5

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 is a path from i to j in G.

The transitive closure of a graph can help efficiently answer questions about reachability. Suppose you want to find out 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). Transitive closure has many applications, including speeding up the processing of structured query languages, which are often used in databases.

In PROC OPTNETWORK, you can invoke the transitive closure algorithm by using the TRANSITIVECLOSURE statement. The options for this statement are described in the section "TRANSITIVECLOSURE Statement" on page 38.

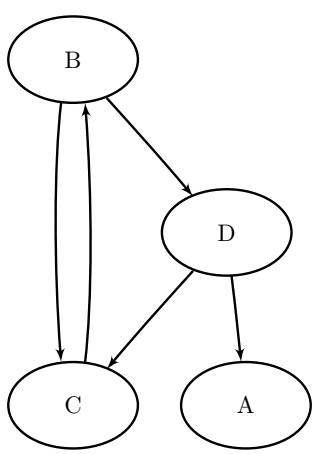
The links that define the transitive closure of the input graph are written to the output data table that you specify in the OUT= option in the TRANSITIVECLOSURE statement.

The algorithm that PROC OPTNETWORK 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 shown in Figure 3.73.

Figure 3.73 A Simple Directed Graph G



The directed graph G can be represented by the following links data table, mycas.LinkSetIn:

```
data mycas.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 table mycas.TransClosure:

```
proc optnetwork
   direction       = directed
   links           = mycas.LinkSetIn;
   transitiveClosure
      out          = mycas.TransClosure;
run;
```

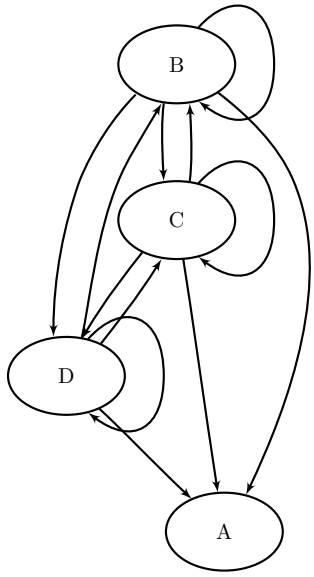
The output data table mycas.TransClosure contains the transitive closure of G, as shown in Figure 3.74.

Figure 3.74 Transitive Closure of a Simple Directed Graph

from	to
B	C
C	B
B	D
D	C
D	A
B	B
D	B
C	C
C	D
D	D
B	A
C	A

The transitive closure of G is shown graphically in Figure 3.75.

Figure 3.75 Transitive Closure of G



For a more detailed example, see "Example 3.5: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System" on page 129.

Traveling Salesman Problem

The *traveling salesman problem* (TSP) finds a minimum-cost tour in a 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 starting node and ending node are the same and 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:

$$
\begin{array}{llcll}
\text{minimize} & \displaystyle\sum_{(i,j)\in A} c_{ij} x_{ij} & & & \\
\text{subject to} & \displaystyle\sum_{(i,j)\in\delta(i)} x_{i,j} & = & 2 \quad i \in N & \text{(two\_match)} \\
 & \displaystyle\sum_{(i,j)\in\delta(S)} x_{ij} & \geq & 2 \quad S \subset N,\ 2 \leq |S| \leq |N| - 1 & \text{(subtour\_elim)} \\
 & x_{ij} \in \{0, 1\} & & (i, j) \in A &
\end{array}
$$

The equations (two_match) are the *matching constraints*, which ensure that each node has degree 2 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 are used on an expanded graph G', as described in Kumar and Li (1994). PROC OPTNETWORK 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. The TSP has applications in planning, logistics, manufacturing, genomics, and many other areas.

In PROC OPTNETWORK, you can invoke the traveling salesman problem solver by using the TSP statement. The options for this statement are described in the section "TSP Statement" on page 38.

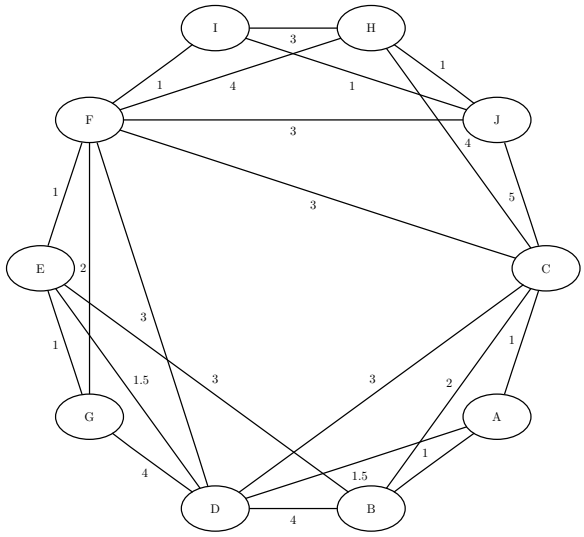
The algorithm that PROC OPTNETWORK uses for solving the TSP is based on a variant of the branch-and-cut process described in Applegate et al. (2006).

The resulting tour is represented in two ways: in the data table that you specify in the OUTNODES= option in the PROC OPTNETWORK statement, the tour is specified as a sequence of nodes; in the data table that you specify in the OUT= option in the TSP statement, the tour is specified as a sequence 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 3.76.

Figure 3.76 A Simple Undirected Graph



You can represent the links data table as follows:

```
data mycas.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 tables mycas.TSPTour and mycas.NodeSetOut:

```
proc optnetwork
   logLevel = moderate
   links    = mycas.LinkSetIn
   outNodes = mycas.NodeSetOut;
   tsp
      out   = mycas.TSPTour;
run;
%put &_OROPTNETWORK_;
```

The progress of the OPTNETWORK procedure is shown in Figure 3.77.

Figure 3.77 PROC OPTNETWORK Log: Optimal Traveling Salesman Tour of a Simple Undirected Graph

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Reading the links data.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: Building the input (full) graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 22.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 16 using 0.04 (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.
          Node   Active   Sols    BestInteger      BestBound      Gap    Time
             0        1      1     16.0000000     15.5005000    3.22%       0
             0        0      1     16.0000000     16.0000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 16.
NOTE: Processing the traveling salesman problem used 0.06 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.141246 seconds.
NOTE: The data set MYCAS.NODESETOUT has 10 observations and 2 variables.
NOTE: The data set MYCAS.TSPTOUR has 10 observations and 4 variables.
STATUS=OK  PROBLEM_TYPE=TSP  SOLUTION_STATUS=OPTIMAL  NUM_SOLUTIONS=1  OBJECTIVE=16
RELATIVE_GAP=0  ABSOLUTE_GAP=0  PRIMAL_INFEASIBILITY=0  BOUND_INFEASIBILITY=0
INTEGER_INFEASIBILITY=0  BEST_BOUND=16  NODES=1  ITERATIONS=14  CPU_TIME=0.12  REAL_TIME=0.14
```

The output data table mycas.NodeSetOut now contains a sequence of nodes in the optimal tour and is shown in Figure 3.78.

Figure 3.78 Nodes in the Optimal Traveling Salesman Tour

node	tsp_order
A	1
B	2
C	3
H	4
J	5
I	6
F	7
G	8
E	9
D	10

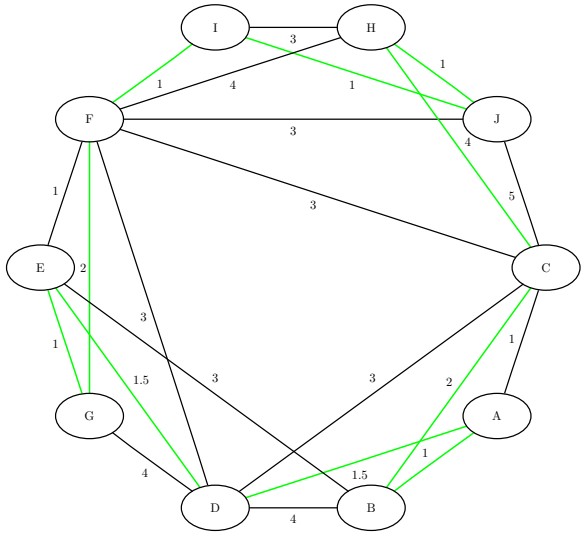
The output data table mycas.TSPTour now contains a sequence of links in the optimal tour and is shown in Figure 3.79.

Figure 3.79 Links in the Optimal Traveling Salesman Tour

tsp_order	from	to	weight
1	A	B	1.0
2	B	C	2.0
3	C	H	4.0
4	H	J	1.0
5	I	J	1.0
6	F	I	1.0
7	F	G	2.0
8	E	G	1.0
9	D	E	1.5
10	A	D	1.5
			16.0

The minimum-cost links are shown in green in Figure 3.80.

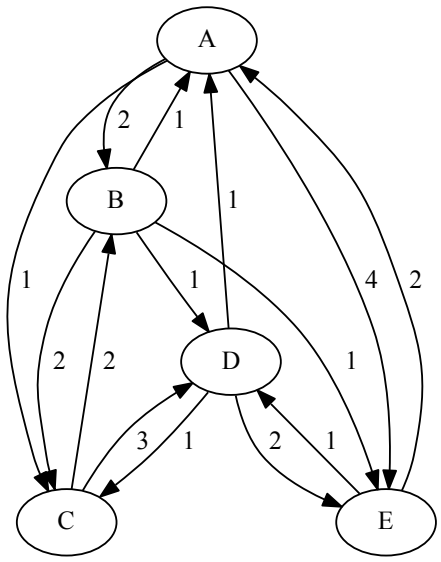
Figure 3.80 Optimal Traveling Salesman Tour



Traveling Salesman Problem Applied to a Simple Directed Graph

As another simple example, consider the weighted directed graph in Figure 3.81. 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.

Figure 3.81 A Simple Directed Graph



You can represent the links data table as follows:

```
data mycas.LinkSetIn;
   input from $ to $ weight @@;
   datalines;
A B 2 A C 1 A E 4 B A 1 B C 2
B D 1 B E 1 C B 2 C D 3 D A 1
D C 1 D E 2 E A 2 E D 1
;
```

The following statements calculate an optimal traveling salesman tour (on a directed graph) and output the results in the data tables mycas.TSPTour and mycas.NodeSetOut:

```
proc optnetwork
   direction = directed
   logLevel  = moderate
   links     = mycas.LinkSetIn
   outNodes  = mycas.NodeSetOut;
   tsp
      out    = mycas.TSPTour;
run;
%put &_OROPTNETWORK_;
```

The progress of the OPTNETWORK procedure is shown in Figure 3.82.

Figure 3.82 PROC OPTNETWORK Log: Optimal Traveling Salesman Tour of a Simple Directed Graph

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Reading the links data.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: Building the input (full) graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 14.
NOTE: The TSP solver is starting using an augmented symmetric graph with 10 nodes and 19 links.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 6 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.
          Node   Active   Sols    BestInteger      BestBound      Gap    Time
             0        1      1      6.0000000      5.9001000    1.69%       0
             0        0      1      6.0000000      6.0000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 6.
NOTE: Processing the traveling salesman problem used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.039407 seconds.
NOTE: The data set MYCAS.NODESETOUT has 5 observations and 2 variables.
NOTE: The data set MYCAS.TSPTOUR has 5 observations and 4 variables.
STATUS=OK  PROBLEM_TYPE=TSP  SOLUTION_STATUS=OPTIMAL  NUM_SOLUTIONS=1  OBJECTIVE=6
RELATIVE_GAP=0  ABSOLUTE_GAP=0  PRIMAL_INFEASIBILITY=0  BOUND_INFEASIBILITY=0
INTEGER_INFEASIBILITY=0  BEST_BOUND=6  NODES=1  ITERATIONS=7  CPU_TIME=0.11  REAL_TIME=0.04
```

The output data table mycas.NodeSetOut now contains a sequence of nodes in the optimal tour and is shown in Figure 3.83.

Figure 3.83 Nodes in the Optimal Traveling Salesman Tour

node	tsp_order
A	1
C	2
B	3
E	4
D	5

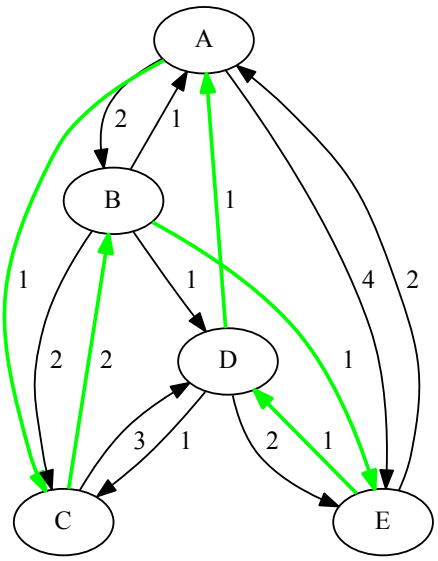
The output data table mycas.TSPTour now contains a sequence of links in the optimal tour and is shown in Figure 3.84.

Figure 3.84 Links in the Optimal Traveling Salesman Tour

tsp_order	from	to	weight
1	A	C	1
2	C	B	2
3	B	E	1
4	E	D	1
5	D	A	1
			6

The minimum-cost links are shown in green in Figure 3.85.

Figure 3.85 Optimal Traveling Salesman Tour



Macro Variable _OROPTNETWORK_

The OPTNETWORK procedure defines a macro variable named _OROPTNETWORK_. This variable contains a character string that indicates the status of PROC OPTNETWORK upon termination and details about the selected algorithm. The various terms of the variable are interpreted as follows:

STATUS

indicates the status of the procedure at termination. The STATUS term can take one of the following values:

OK	The procedure terminated normally.
OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.

INTERRUPTED	The procedure was interrupted by the user.
ERROR	The procedure encountered an error.

PROBLEM_TYPE

indicates the selected problem type (algorithm class). The PROBLEM_TYPE term can take one of the following values:

BICONNECTEDCOMPONENTS	Biconnected components
CLIQUE	Clique enumeration
CONNECTEDCOMPONENTS	Connected components
CYCLE	Cycle enumeration
LINEARASSIGNMENT	Weighted matching
MINCOSTFLOW	Minimum-cost network flow
MINCUT	Minimum cut
MINSPANTREE	Minimum spanning tree
PATH	Path enumeration
SHORTESTPATH	Shortest path
SUMMARY	Graph summary
TRANSITIVECLOSURE	Transitive closure
TSP	Traveling salesman

SOLUTION_STATUS

indicates the solution status of the selected problem type (algorithm class). The SOLUTION_STATUS term can take one of the following values:

OK	The algorithm terminated normally.
ERROR	The algorithm encountered an error.
OPTIMAL	The solution is optimal.
OPTIMAL_AGAP	The solution is optimal within the absolute gap that you specified in the ABSOBJGAP= option.
OPTIMAL_RGAP	The solution is optimal within the relative gap that you specified in the RELOBJGAP= option.
OPTIMAL_COND	The solution is optimal, but some infeasibilities (primal, bound, or integer) exceed tolerances because of scaling.
TARGET	The solution is not worse than the target that you specified in the TARGET= option.
INFEASIBLE	The problem is found to be 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 that you specified in the MAXSOLS= option.
NODE_LIM_SOL	The solver reached the maximum number of nodes that you specified in the MAXNODES= option and found a solution.
NODE_LIM_NOSOL	The solver reached the maximum number of nodes that you specified in the MAXNODES= option and did not find a solution.
TIMELIMIT	The algorithm reached the execution time limit that you specified in the MAXTIME= option.
TIME_LIM_SOL	The solver reached the execution time limit that you specified in the MAXTIME= option and found a solution.
TIME_LIM_NOSOL	The solver reached the execution time limit that you specified in the MAXTIME= option and did not find a solution.
HEURISTIC_SOL	The solver used only heuristics and found a solution.
HEURISTIC_NOSOL	The solver used only heuristics and did not find a solution.
INTERRUPTED	The algorithm was interrupted by the user.
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.

CPU_TIME
indicates the total CPU time (in seconds) that PROC OPTNETWORK used.

REAL_TIME
indicates the elapsed time (in seconds) that PROC OPTNETWORK used.

In addition, each algorithm might report some additional details. The following section provides more information about these details.

Macro Variable _OROPTNETWORK_ Details

The BICONNECTEDCOMPONENTS algorithm provides the following additional information:

NUM_COMPONENTS
indicates the number of biconnected components that the algorithm found.

NUM_ARTICULATION_POINTS
indicates the number of articulation points that the algorithm found.

The CLIQUE algorithm provides the following additional information:

NUM_CLIQUES
indicates the number of cliques that the algorithm found.

The CONNECTEDCOMPONENTS algorithm provides the following additional information:

NUM_COMPONENTS
indicates the number of connected components that the algorithm found.

The CYCLE algorithm provides the following additional information:

NUM_CYCLES
indicates the number of cycles that the algorithm found.

The LINEARASSIGNMENT algorithm provides the following additional information:

OBJECTIVE
indicates the total weight of the minimum linear assignment.

The MINCOSTFLOW algorithm provides the following additional information:

OBJECTIVE
indicates the total link weight of the minimum-cost network flow.

The MINCUT algorithm provides the following additional information:

OBJECTIVE
indicates the total link weight of the minimum cut.

The MINSPANTREE algorithm provides the following additional information:

OBJECTIVE
indicates the total link weight of the minimum spanning tree.

The PATH algorithm provides the following additional information:

NUM_PATHS
indicates the number of paths that the algorithm found.

The SHORTESTPATH algorithm provides the following additional information:

NUM_PATHS
indicates the number of shortest paths that the algorithm found.

The TSP algorithm provides the following additional information:

OBJECTIVE
indicates the objective value that the solver obtains at termination.

RELATIVE_GAP
indicates the relative gap between the best integer objective (BestInteger) and the objective of the best remaining node (BestBound) upon termination of the solver. The relative gap is equal to

$$|\text{BestInteger} - \text{BestBound}| / (1\text{E}{-10} + |\text{BestBound}|)$$

ABSOLUTE_GAP
indicates the absolute gap between the best integer objective (BestInteger) and the objective of the best remaining node (BestBound) upon termination of the 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 that the solver returned.

BEST_BOUND
indicates the best linear programming objective value of all unprocessed nodes in the branch-and-bound tree at the end of execution. A missing value indicates that the solver has processed either all or none of the nodes in the branch-and-bound tree.

NODES
indicates the number of nodes that the solver enumerated by using the branch-and-bound algorithm.

ITERATIONS
indicates the number of simplex iterations that the solver used to solve the problem.

ODS Table Names

For general information about ODS tables, see *SAS Output Delivery System: Procedures Guide*. Each ODS table that the OPTNETWORK procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. These names are listed in Table 3.8.

Table 3.8 ODS Tables Produced by PROC OPTNETWORK

Table Name	Description
ProblemSummary	Summary of the graph
SolutionSummary	Summary of the solution status, timing, and results

The following statements use the example in the section "Shortest Paths for All Pairs" on page 87 and find all-pairs shortest paths for a small undirected graph. By default, this code produces the two ODS output tables listed in Table 3.8.

```
data mycas.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
;

proc optnetwork
   links         = mycas.LinkSetIn;
   shortestPath
      outWeights = mycas.ShortPathW
      outPaths   = mycas.ShortPathP;
run;
```

The problem summary table in Figure 3.86 provides a basic summary of the graph input.

Figure 3.86 Problem Summary Table

The OPTNETWORK Procedure

Problem Summary	
Number of Nodes	6
Number of Links	10
Graph Direction	Undirected

The solution summary table in Figure 3.87 provides a basic solution summary for the algorithm that is processed. The information in this table is similar to the information that is provided in the macro variable _OROPTNETWORK_, described in the section "Macro Variable _OROPTNETWORK_" on page 112. The timing information in this table (and in the log) represents the time spent running the algorithm, excluding the time spent in input, graph building, and output. In the case of a distributed algorithm, which uses multiple machines, the real time represents the maximum amount of time that an individual machine used to run the algorithm, and the CPU time represents the total amount of time across all active machines in your configured session.

Figure 3.87 Solution Summary Table

Solution Summary	
Problem Type	Shortest Path
Solution Status	OK
Number of Paths	30
CPU Time	0.01
Real Time	0.01

Examples: OPTNETWORK Procedure

Example 3.1: 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 3.88 shows this network, which was constructed after the attacks, based on collected intelligence information.

Figure 3.88 Terrorist Communications Network from 9/11

The full network data include 153 links. The following statements show a small subset to illustrate the use of the BICONNECTEDCOMPONENTS statement in this context:

```
data mycas.LinkSetInTerror911;
   input from & $32. to & $32.;
   datalines;
Abu Zubeida                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
Jerome Courtaillier        Zacarias Moussaoui
```

```
Jerome Courtaillier        David Courtaillier
Zacarias Moussaoui         David Courtaillier
Zacarias Moussaoui         Ahmed Ressam
Zacarias Moussaoui         Abu Qatada
Zacarias Moussaoui         Ramzi Bin al-Shibh
Zacarias Moussaoui         Mahamed Atta
Ahmed Ressam               Haydar Abu Doha
Mehdi Khammoun             Haydar Abu Doha
Essid Sami Ben Khemais     Haydar Abu Doha
Mehdi Khammoun             Essid Sami Ben Khemais
Mehdi Khammoun             Mohamed Bensakhria
...
;
```

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:

```
proc optnetwork
   links    = mycas.LinkSetInTerror911
   outNodes = mycas.NodeSetOut;
   biconnectedComponents;
run;

data mycas.ArtPoints;
   set   mycas.NodeSetOut;
   where artpoint=1;
run;
```

The output data table mycas.ArtPoints contains members of the network who are articulation points. By focusing on cutting off these particular members, investigators could have significantly disrupted the terrorists' ability to communicate when planning the attack.

Output 3.1.1 Articulation Points of Terrorist Communications Network from 9/11

node	artpoint
Djamal Beghal	1
Mohamed Atta	1
Zacarias Moussaoui	1
Nawaf Alhazmi	1
Essid Sami Ben Khemais	1
Mamoun Darkazanli	1

Example 3.2: Cycle Enumeration for Kidney Donor Exchange

This example looks at an application of cycle enumeration 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 3.89. More generally, an n-way swap can be performed involving n donors and n recipients (Willingham 2009).

Figure 3.89 Kidney Donor Exchange Two-Way Swap

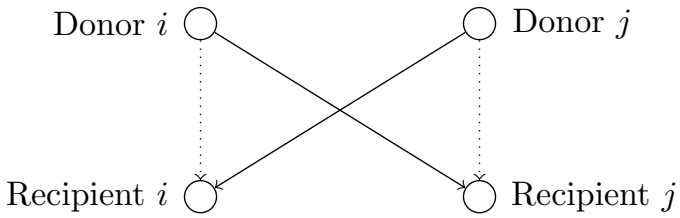
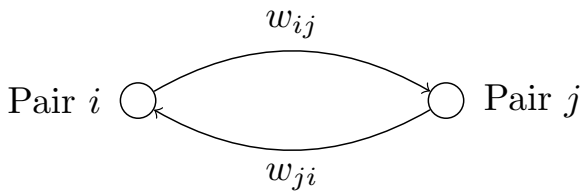


Figure 3.90 Kidney Donor Exchange Network



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 3.90. 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 67. 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.

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 OPTNETWORK to generate the cycles and then PROC OPTMODEL to read the PROC OPTNETWORK output, formulate the set packing problem, call the mixed integer linear programming solver, and output the optimal solution. See Chapter 10, "The OPTMODEL Procedure" (*SAS Optimization: Mathematical Optimization Procedures*).

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:

```
/* create random graph on n nodes with arc probability p
   and uniform(0,1) weight */
%let n = 100;
%let p = 0.02;
data mycas.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;
run;
```

The following statements use PROC OPTNETWORK to generate all cycles whose length is greater than or equal to 2 and less than or equal to 10:

```
/* generate all cycles with 2 <= length <= max_length */
%let max_length = 10;
proc optnetwork
   logLevel     = moderate
   direction    = directed
   links        = mycas.LinkSetIn;
   cycle
      minLength = 2
      maxLength = &max_length
      out       = mycas.Cycles
      maxCycles = all;
run;
%put &_OROPTNETWORK_;
```

PROC OPTNETWORK finds 395 cycles of the appropriate length, as shown in Output 3.2.1.

Output 3.2.1 Cycles for Kidney Donor Exchange PROC OPTNETWORK Log

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Reading the links data.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: Building the input (full) graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 98.
NOTE: The number of links in the input graph is 208.
NOTE: Processing cycle detection using 32 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.
NOTE: The Cloud Analytic Services server processed the request in 0.036714 seconds.
NOTE: The data set MYCAS.CYCLES has 3826 observations and 3 variables.
STATUS=OK  PROBLEM_TYPE=CYCLE  SOLUTION_STATUS=OK  NUM_CYCLES=395  CPU_TIME=0.12  REAL_TIME=0.04
```

From the resulting data table mycas.Cycles, use the following DATA step to convert the cycles into one observation per arc:

```
/* convert cycles into one observation per arc */
data Cycles;
   set mycas.Cycles;
run;
proc sort data=Cycles;
   by cycle order;
run;
data Cycles0(keep=c i j);
   set Cycles;
   retain last;
   c    = cycle;
   i    = last;
   j    = node;
   last = j;
   if order ne 1 then output;
run;
```

For this set of cycles, you can now formulate a mixed integer linear program (MILP) to maximize the total cycle weight. Let C define the set of cycles of appropriate length, N_c define the set of nodes in cycle c, A_c define the set of links in cycle c, and w_{ij} denote 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:

$$\begin{array}{lll} \text{maximize} & \displaystyle\sum_{c \in C} \left(\sum_{(i,j) \in A_c} w_{ij} \right) x_c & \\ \text{subject to} & \displaystyle\sum_{c \in C : i \in N_c} x_c \leq 1 & i \in N \qquad \text{(incomp\_pair)} \\ & x_c \in \{0, 1\} & c \in C \end{array}$$

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:

```
/* solve set packing problem to find maximum-weight node-disjoint union
   of short directed cycles */
proc optmodel;
   /* declare index sets and parameters, and read data */
   set <num,num> ARCS;
   num weight {ARCS};
   read data mycas.LinkSetIn into ARCS=[from to] weight;
   set <num,num,num> TRIPLES;
   read data Cycles0 into TRIPLES=[c i j];
   set CYCLES = setof {<c,i,j> in TRIPLES} c;
   set ARCS_c {c in CYCLES} = setof {<(c),i,j> in TRIPLES} <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];
   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;
%put &_OROPTMODEL_;
```

PROC OPTMODEL solves the problem by using the mixed integer linear programming solver.

Output 3.2.2 Cycles for Kidney Donor Exchange PROC OPTMODEL Log

```
NOTE: There were 208 observations read from the data set MYCAS.LINKSETIN.
NOTE: There were 3431 observations read from the data set WORK.CYCLES0.
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 initial MILP heuristics are applied.
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.
          Node   Active   Sols    BestInteger      BestBound      Gap    Time
             0        1      3     20.6710373   1147.4221881   98.20%       0
             0        1      3     20.6710373     25.4194215   18.68%       0
             0        1      3     20.6710373     25.1227054   17.72%       0
             0        1      4     20.8736621     25.0541979   16.69%       0
             0        1      4     20.8736621     24.9277715   16.26%       0
             0        1      5     24.8508554     24.8508554    0.00%       0
             0        0      5     24.8508554     24.8508554    0.00%       0
NOTE: The MILP solver added 10 cuts with 955 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 24.850855395.
NOTE: The data set WORK.SOLUTION has 7 observations and 2 variables.
STATUS=OK ALGORITHM=BAC SOLUTION_STATUS=OPTIMAL OBJECTIVE=24.850855395 RELATIVE_GAP=0
ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=2.220446E-16 BOUND_INFEASIBILITY=2.220446E-16
INTEGER_INFEASIBILITY=4.107825E-15 BEST_BOUND=24.850855395 NODES=1 SOLUTIONS_FOUND=6
ITERATIONS=113 PRESOLVE_TIME=0.06 SOLUTION_TIME=0.15
```

The output data table mycas.Solution, shown in Output 3.2.3, now contains the cycles that define the best exchange and their associated weight (quality).

Output 3.2.3 Maximum-Quality Solution for Kidney Donor Exchange

c	cycle_weight
29	4.3542
117	4.9748
200	4.3403
243	5.0843
272	1.7253
277	2.4295
385	1.9424
	24.8509

Example 3.3: Linear Assignment Problem for Minimizing Relay Times

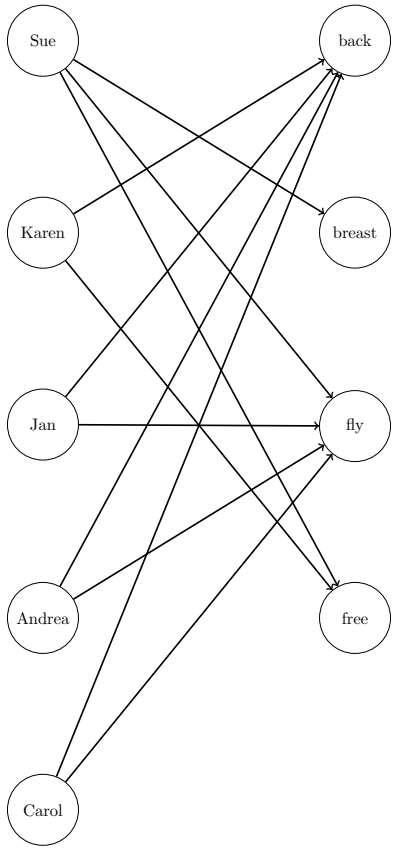
A swimming coach needs to assign a swimmer to each leg of a medley relay team; each swimmer in the relay uses a different stroke (backstroke, breaststroke, butterfly, or freestyle). The swimmers' best times for each stroke are stored in a SAS data set. The LINEARASSIGNMENT statement evaluates the times 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). Certain swimmers are not eligible to perform certain strokes in the relay because they do not excel at these strokes. A missing (.) value in the data matrix identifies an ineligible assignment. For example:

```
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 be interpreted as the minimum-weight matching in a bipartite graph. The eligible assignments define links between the rows (swimmers) and the columns (strokes), as in Figure 3.91.

Figure 3.91 Bipartite Graph for Linear Assignment Problem



You can transform the matrix data format into a links data table as follows:

```
data mycas.RelayTimesLinks(keep=name attr cost);
   set RelayTimesMatrix;
   length attr $ 8;
   array stroke[4] back breast fly free;
   do s = 1 to dim(stroke);
      if stroke[s] ne . then do;
         attr = vname(stroke[s]);
         cost = stroke[s];
         output;
      end;
   end;
run;
```

This graph must be bipartite (such that S and T are disjoint). If it is not, PROC OPTNETWORK returns an error.

The following statements find the optimal minimum-weight matching:

```
proc optnetwork
   direction       = directed
   links           = mycas.RelayTimesLinks;
   linksVar
      from         = name
      to           = attr
      weight       = cost;
   linearAssignment
      out          = mycas.LinearAssignLinks;
run;
```

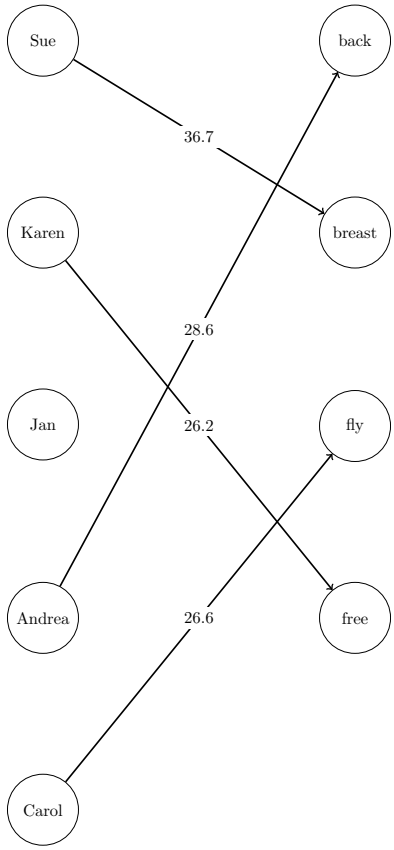
The output data table mycas.LinearAssignLinks contains the optimal assignment, as shown in Output 3.3.1.

Output 3.3.1 Optimal Assignments for Swim Times

name	attr	cost
Andrea	back	28.6
Carol	fly	26.6
Karen	free	26.2
Sue	breast	36.7
		118.1

The optimal assignments are shown graphically in Figure 3.92.

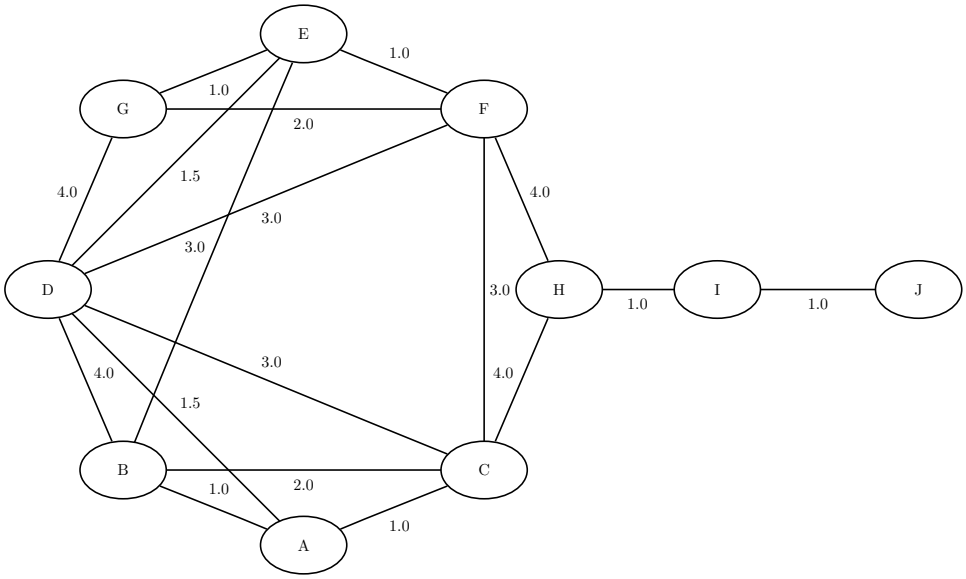
Figure 3.92 Optimal Assignments for Swim Times



Example 3.4: Minimum Spanning Tree for Computer Network Design

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 3.93. 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 for the network of possible links.

Figure 3.93 Potential Office Computer Network



Define the links data table as follows:

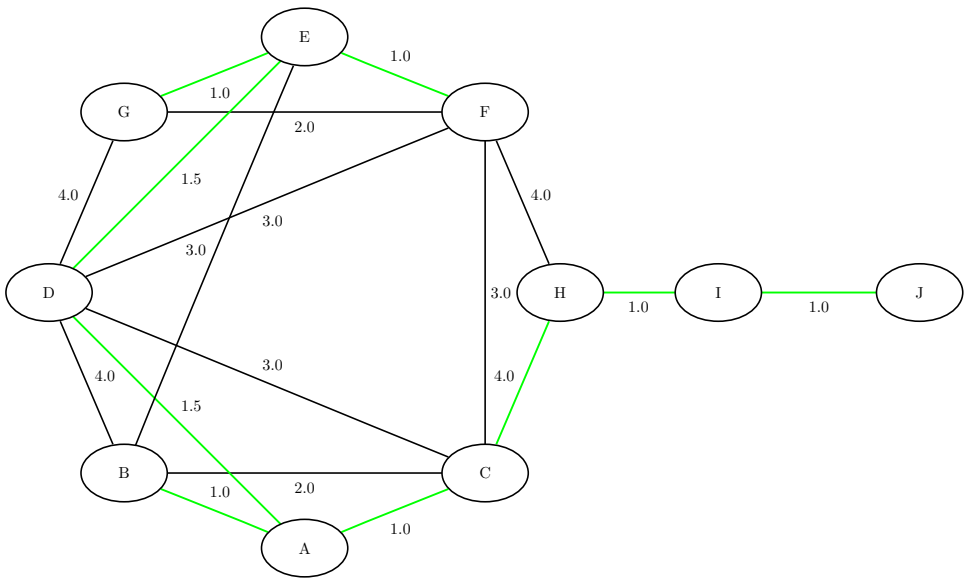
```
data mycas.LinkSetInCompNet;
   input from $ to $ weight @@;
   datalines;
A B 1.0  A C 1.0  A D 1.5  B C 2.0  B D 4.0
B E 3.0  C D 3.0  C F 3.0  C H 4.0  D E 1.5
D F 3.0  D G 4.0  E F 1.0  E G 1.0  F G 2.0
F H 4.0  H I 1.0  I J 1.0
;
```

The following statements find a minimum spanning tree:

```
proc optnetwork
   links      = mycas.LinkSetInCompNet;
   minSpanTree
      out     = mycas.MinSpanTree;
run;
```

Output 3.4.1 shows the resulting data table mycas.MinSpanTree, which is displayed graphically in Figure 3.94, with the minimal cost links shown in green.

Figure 3.94 Minimum Spanning Tree for Office Computer Network



Output 3.4.1 Minimum Spanning Tree for a Computer Network Design

from	to	weight
I	J	1.0
A	C	1.0
E	F	1.0
E	G	1.0
H	I	1.0
A	B	1.0
D	E	1.5
A	D	1.5
C	H	4.0
		13.0

Example 3.5: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System

Most systems that track software errors, or bugs, 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 dependency*. 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. 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 dependency, and no primary bug is defined for the development team to focus on. You can easily see this circular dependency in the graph representation, because $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$. You can find such circular dependencies by using cycle

enumeration, which is described in the section "Cycle Enumeration" on page 62. The second situation that needs to be checked is more general. If one 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. You can identify the existence of these chains 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 some chain that is present in G.

Consider the following data, which define some duplicated bugs (called *defects*) in a small sample of the bug tracking system:

```
data mycas.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 mycas.DefectLinks. The output data table mycas.Cycles contains any circular dependencies, and the data table mycas.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 optnetwork
   logLevel        = moderate
   direction       = directed
   links           = mycas.DefectLinks;
   linksVar
      from         = defectId
      to           = linkedDefect;
   cycle
      out          = mycas.Cycles
      maxCycles    = all;
run;
%put &_OROPTNETWORK_;

proc optnetwork
   logLevel        = moderate
   direction       = directed
   links           = mycas.DefectLinks;
   linksVar
      from         = defectId
```

```
      to           = linkedDefect;
   transitiveClosure
      out          = mycas.TransClosure;
run;
%put &_OROPTNETWORK_;

proc sql;
   create table Chains as
   select defectId, linkedDefect
      from mycas.TransClosure(where=(defectId ne linkedDefect)) except
   select defectId, linkedDefect
      from mycas.DefectLinks;
quit;
```

The progress of the procedure is shown in Output 3.5.1.

Output 3.5.1 PROC OPTNETWORK Log: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Reading the links data.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: Building the input (full) graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 16.
NOTE: The number of links in the input graph is 12.
NOTE: Processing cycle detection.
NOTE: Processing cycle detection using the backtrack algorithm.
NOTE: The algorithm found 1 cycles.
NOTE: Processing cycle detection used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.033453 seconds.
NOTE: The data set MYCAS.CYCLES has 4 observations and 3 variables.
STATUS=OK  PROBLEM_TYPE=CYCLE  SOLUTION_STATUS=OK  NUM_CYCLES=1  CPU_TIME=0.10  REAL_TIME=0.03
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Reading the links data.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: Building the input (full) graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 16.
NOTE: The number of links in the input graph is 12.
NOTE: Processing the transitive closure.
NOTE: Processing the transitive closure used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.033135 seconds.
NOTE: The data set MYCAS.TRANSCLOSURE has 20 observations and 2 variables.
STATUS=OK  PROBLEM_TYPE=TRANSITIVECLOSURE  SOLUTION_STATUS=OK  CPU_TIME=0.10  REAL_TIME=0.03
NOTE: Table WORK.CHAINS created, with 5 rows and 2 columns.
```

Output 3.5.2 displays the output data table mycas.Cycles, which contains one case of a circular dependency in which the DUPs start and end at S0152674.

Output 3.5.2 Cycle in Bug Tracking System

cycle	order	node
1	1	S0152674
1	2	S0153280
1	3	S0153307
1	4	S0152674

Output 3.5.3 displays the local data set Chains, which contains the chains in the bug tracking system that come from the links in G^T that are not in G.

Output 3.5.3 Chains in Bug Tracking System

defectId	linkedDefect
S0152674	S0153307
S0153280	S0152674
S0153307	S0153280
S0162973	S0165405
S0325026	S0643230

Example 3.6: 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 105.

The following data set provides a list of the capital cities and their latitude and longitude:

```
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
Austin, TX          30.274722222 -97.74055556
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    -119.7662917
Charleston, WV      38.336388889 -81.61222222
Cheyenne, WY        41.140277778 -104.8197222
Columbia, SC        34.000433333 -81.03314722
Columbus, OH        39.961388889 -82.99888889
Concord, NH         43.206747222 -71.53812778
Denver, CO          39.739094444 -104.9848972
Des Moines, IA      41.591177778 -93.60386944
```

```
Dover, DE           39.157305556 -75.51972222
Frankfort, KY       38.186777778 -84.87533333
Harrisburg, PA      40.264444444 -76.86666667
Hartford, CT        41.764136111 -72.68277778
Helena, MT          46.5857      -112.0184
Indianapolis, IN    39.768611111 -86.1625
Jackson, MS         32.303888889 -90.18222222
Jefferson City, MO  38.579119444 -92.17299167
Lansing, MI         42.733727778 -84.55558889
Lincoln, NE         40.808088889 -96.69958611
Little Rock, AR     34.746758333 -92.28876111
Madison, WI         43.074444444 -89.38472222
Montgomery, AL      32.377447222 -86.30094167
Montpelier, VT      44.262222222 -72.58083333
Nashville, TN       36.165833333 -86.78416667
Oklahoma City, OK   35.492280556 -97.50337222
Olympia, WA         47.035277778 -122.9063889
Phoenix, AZ         33.448097222 -112.0970944
Pierre, SD          44.367166667 -100.3463528
Providence, RI      41.830833333 -71.415
Raleigh, NC         35.780277778 -78.63916667
Richmond, VA        37.538758333 -77.43359444
Sacramento, CA      38.576572222 -121.4934111
Saint Paul, MN      44.955147222 -93.10223611
Salem, OR           44.938730556 -123.0300972
Salt Lake City, UT  40.777222222 -111.8880556
Santa Fe, NM        35.682280556 -105.9396583
Springfield, IL     39.798516667 -89.65488889
Tallahassee, FL     30.438111111 -84.2816
Topeka, KS          39.048008333 -95.67815556
Trenton, NJ         40.220436111 -74.76990278
Washington, DC      38.889802778 -77.00911389
;
```

From this list, you can generate a links data table, mycas.CitiesDist, that contains the distances (in miles) between each pair of cities. The distances are calculated by using the SAS function GEODIST.

```
/* create a list of all the possible pairs of cities */
proc sql;
   create table mycas.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 OPTNETWORK statements find an optimal tour:

```
/* find optimal tour using OPTNETWORK */
proc optnetwork
   logLevel   = moderate
   links      = mycas.CitiesDist
   outNodes   = mycas.TSPTourNodes;
```

```
      linksVar
         from    = city1
         to      = city2
         weight  = distance;
      tsp
         out     = mycas.TSPTourLinks;
   run;
   %put &_OROPTNETWORK_;
```

The progress of the procedure is shown in Output 3.6.1. The total mileage that is needed to optimally visit the capital cities is 10,637.36 miles.

Output 3.6.1 PROC OPTNETWORK Log: Traveling Salesman Tour of US Capital Cities

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Reading the links data.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: Building the input (full) graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 49.
NOTE: The number of links in the input graph is 1176.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 10637.362018 using 0.10 (cpu: 0.07)
      seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Branch and Cut algorithm is used.
          Node   Active   Sols    BestInteger      BestBound      Gap    Time
             0        1      1  10637.3620180  10056.3886552    5.78%       0
             0        1      2  10637.3620180  10263.0069519    3.65%       0
             0        1      3  10637.3620180  10283.8406458    3.44%       0
             0        1      3  10637.3620180  10285.8416375    3.42%       0
             0        1      3  10637.3620180  10315.5163280    3.12%       0
             0        1      3  10637.3620180  10408.7862167    2.20%       0
             0        1      3  10637.3620180  10488.6539926    1.42%       0
             0        1      3  10637.3620180  10498.4713072    1.32%       0
             0        1      3  10637.3620180  10549.6939727    0.83%       0
             0        1      3  10637.3620180  10581.1239040    0.53%       0
             0        1      3  10637.3620180  10602.7858111    0.33%       0
             0        1      3  10637.3620180  10629.8536061    0.07%       0
             0        0      3  10637.3620180  10637.3620180    0.00%       0
NOTE: The MILP solver added 14 cuts with 4094 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 10637.362018.
NOTE: Processing the traveling salesman problem used 0.13 (cpu: 0.09) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.199135 seconds.
NOTE: The data set MYCAS.TSPTOURNODES has 49 observations and 2 variables.
NOTE: The data set MYCAS.TSPTOURLINKS has 49 observations and 4 variables.
STATUS=OK  PROBLEM_TYPE=TSP  SOLUTION_STATUS=OPTIMAL  NUM_SOLUTIONS=3  OBJECTIVE=10637.362018
RELATIVE_GAP=0  ABSOLUTE_GAP=0  PRIMAL_INFEASIBILITY=0  BOUND_INFEASIBILITY=0
INTEGER_INFEASIBILITY=0  BEST_BOUND=10637.362018  NODES=1  ITERATIONS=188  CPU_TIME=0.22
REAL_TIME=0.20
```

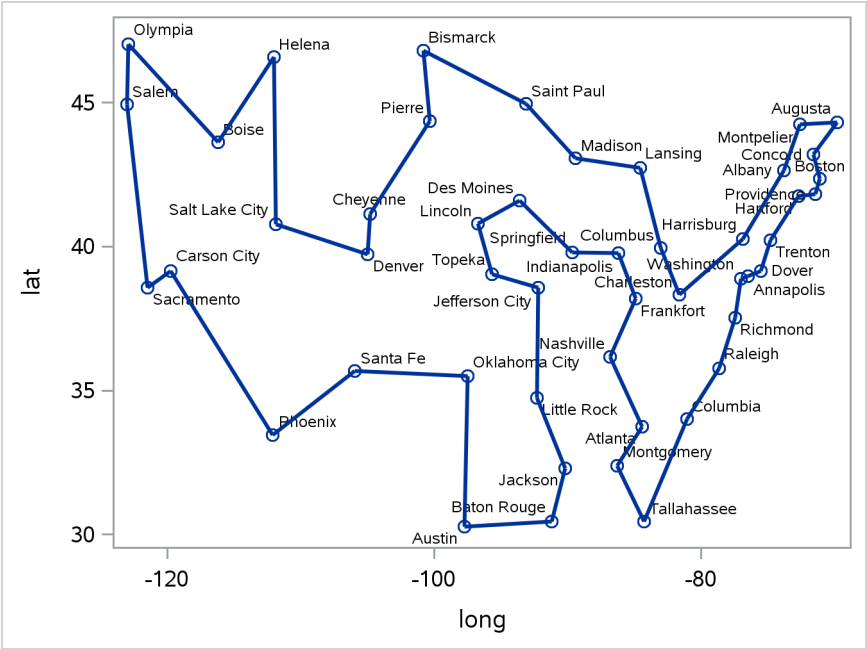
The following statements produce a graphical display of the solution:

```
/* merge latitude and longitude */
data TSPTourLinks;
   set mycas.TSPTourLinks;
run;
```

```
proc sort data=TSPTourLinks;
   by tsp_order;
run;
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 mycas.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;

data sganno;
   set TSPTourLinksAnno2(rename=(long1=x1 lat1=y1 long2=x2 lat2=y2));
   drawspace = 'datavalue';
   function  = 'line';
run;

data Cities2;
   set Cities;
   label = scan(city,1,',');
run;

proc sgplot data=Cities2 sganno=sganno;
   scatter y=lat x=long / datalabel=label;
   yaxis offsetmax=0.05;
run;
```

The minimal-cost tour of the capital cities is shown in Figure 3.6.2.

Output 3.6.2 Optimal Traveling Salesman Tour of US Capital Cities



The output data set TSPTourLinks contains the sequence of 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:

```
/* 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 output data set TSPTourLinksDirected is shown in Figure 3.95.

Figure 3.95 Links in the Optimal Traveling Salesman Tour

city1	city2	distance
Albany, NY	Harrisburg, PA	230.71
Harrisburg, PA	Charleston, WV	287.04
Charleston, WV	Columbus, OH	134.61
Columbus, OH	Lansing, MI	207.78
Lansing, MI	Madison, WI	246.20
Madison, WI	Saint Paul, MN	226.22
Saint Paul, MN	Bismarck, ND	392.09
Bismarck, ND	Pierre, SD	170.80
Pierre, SD	Cheyenne, WY	318.40
Cheyenne, WY	Denver, CO	97.09
Denver, CO	Salt Lake City, UT	371.84
Salt Lake City, UT	Helena, MT	401.15
Helena, MT	Boise, ID	289.50
Boise, ID	Olympia, WA	402.89
Olympia, WA	Salem, OR	144.95
Salem, OR	Sacramento, CA	446.28
Sacramento, CA	Carson City, NV	101.59
Carson City, NV	Phoenix, AZ	581.52
Phoenix, AZ	Santa Fe, NM	383.38
Santa Fe, NM	Oklahoma City, OK	475.23
Oklahoma City, OK	Austin, TX	359.89
Austin, TX	Baton Rouge, LA	391.67
Baton Rouge, LA	Jackson, MS	140.45
Jackson, MS	Little Rock, AR	207.71
Little Rock, AR	Jefferson City, MO	264.39
Jefferson City, MO	Topeka, KS	191.94
Topeka, KS	Lincoln, NE	133.03
Lincoln, NE	Des Moines, IA	170.19
Des Moines, IA	Springfield, IL	241.50
Springfield, IL	Indianapolis, IN	185.93
Indianapolis, IN	Frankfort, KY	129.29
Frankfort, KY	Nashville, TN	174.72
Nashville, TN	Atlanta, GA	215.06
Atlanta, GA	Montgomery, AL	145.82
Montgomery, AL	Tallahassee, FL	179.17
Tallahassee, FL	Columbia, SC	310.59
Columbia, SC	Raleigh, NC	183.17
Raleigh, NC	Richmond, VA	138.55
Richmond, VA	Washington, DC	96.03
Washington, DC	Annapolis, MD	28.58
Annapolis, MD	Dover, DE	53.68
Dover, DE	Trenton, NJ	83.54
Trenton, NJ	Hartford, CT	152.52
Hartford, CT	Providence, RI	65.65
Providence, RI	Boston, MA	40.61
Boston, MA	Concord, NH	63.39
Concord, NH	Augusta, ME	116.20
Augusta, ME	Montpelier, VT	138.89
Montpelier, VT	Albany, NY	125.92
		10,637.36

Example 3.7: Connected Components for US Patent Citations

This example looks at the structural relationship of US patent citations by using a large data set that is maintained by the Stanford Network Analysis Project (SNAP) (Leskovec 2014). The citation graph includes over 16 million citations made to patents between 1975 and 1999.

The following statements construct the links data table mycas.Patents from a local copy of the raw patent citation data:

```
filename in 'cit-Patents.txt';
data mycas.Patents;
   infile in firstobs=5 dlm='09'X;
   input from to;
run;
```

The following statements find the connected components of the citation graph by using a distributed union-find algorithm. This algorithm takes advantage of all the machines in your configured session.

```
proc optnetwork
   links        = mycas.Patents
   outNodes     = mycas.OutNodes;
   connectedComponents
```

```
      algorithm = parallel;
run;
%put &_OROPTNETWORK_;
```

The progress of the procedure is shown in Output 3.7.1.

Output 3.7.1 PROC OPTNETWORK Log: Connected Components for US Patent Citations

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: The graph contains 1 self links that are ignored.
NOTE: The number of nodes in the input graph is 3774768.
NOTE: The number of links in the input graph is 16518947.
NOTE: Processing connected components using 4 machines.
NOTE: The graph has 3627 connected components.
NOTE: Processing connected components used 0.85 (cpu: 1.90) seconds.
NOTE: The Cloud Analytic Services server processed the request in 6.049694 seconds.
NOTE: The data set MYCAS.OUTNODES has 3774768 observations and 2 variables.
STATUS=OK  PROBLEM_TYPE=CONNECTEDCOMPONENTS  SOLUTION_STATUS=OK  NUM_COMPONENTS=3627
CPU_TIME=36.63  REAL_TIME=6.05
```

The following statements use PROC SQL to calculate the size of each component:

```
proc sql;
   create table FreqCount as
   select concomp, count(*) as count
   from mycas.OutNodes
   group by concomp
   order by count descending;
quit;
```

The 10 biggest components are shown in Output 3.7.2. It is interesting to note that the vast majority of patents (over 99%) are all contained in the same component. This is not too surprising, because many of the seminal patent claims are required in order to understand subsequent inventions.

Output 3.7.2 Ten Largest Components for US Patent Citations

Obs	concomp	count
1	1	3764117
2	446	19
3	242	16
4	25	15
5	345	14
6	169	14
7	263	14
8	1431	14
9	239	13
10	158	13

Example 3.8: Shortest Paths of the New York Road Network

This example looks at the road networks in the state of New York (NY). The distance graph raw data are maintained at the DIMACS challenge website (Demetrescu 2010). The NY road network includes 264,346 intersections (nodes) and 365,050 roads (links). Although the input data table is not large, the computing power that you need to find all-pairs shortest paths is enormous. In addition, the storage space that you need to handle the results data can easily overwhelm the capacity of a single machine. In this example, a session of 130 machines (each with 32 cores) was configured to process this graph.

The following statements construct the links data table mycas.RoadNY from a local copy of the raw distance graph data:

```
filename in 'USA-road-d.NY.gr';
data mycas.RoadNY (drop=a);
   infile in firstobs=8;
   input a $ from $ to $ weight;
run;
```

The following statements find the all-pairs shortest paths of the NY road network (that have a total path weight of less than 20,000) by using a distributed algorithm. This algorithm takes advantage of all the machines and cores in your configured session.

```
proc optnetwork
   logFreqTime      = 10
   logLevel         = aggressive
   links            = mycas.RoadNY;
   shortestPath
      maxPathWeight = 20000
      outWeights    = mycas.shortPathSummary
      outPaths      = mycas.shortPathPaths;
run;
%put &_OROPTNETWORK_;
```

The progress of the procedure is shown in Output 3.8.1.

Output 3.8.1 PROC OPTNETWORK Log: Shortest Paths of the NY Road Network

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Reading the links data.
NOTE: Data input used 1.29 (cpu: 59.69) seconds.
NOTE: The graph contains 368796 duplicate links that are ignored.
NOTE: Building the input (full) graph storage used 1.68 (cpu: 108.79) seconds.
NOTE: The number of nodes in the input graph is 264346.
NOTE: The number of links in the input graph is 365050.
NOTE: Processing the shortest paths problem using 32 threads on each of 130 machines.
NOTE: Processing the shortest paths problem between 264346 source nodes and 264346 sink nodes.
                                                   Real
      Algorithm              Sources   Complete     Time
      shortestPath               320         0%    11.52
      shortestPath             41920        15%    21.16
      shortestPath             83675        31%    31.13
      shortestPath            124715        47%    41.03
      shortestPath            165060        62%    51.19
      shortestPath            205160        77%    61.00
      shortestPath            245556        92%    71.14
      shortestPath            264346       100%    84.30
NOTE: Processing the shortest paths problem used 81.21 (cpu: 231662.52) seconds.
NOTE: The Cloud Analytic Services server processed the request in 90.427072 seconds.
NOTE: The data set MYCAS.SHORTPATHSUMMARY has 104263396 observations and 3 variables.
NOTE: The data set MYCAS.SHORTPATHPATHS has 1419295895 observations and 6 variables.
STATUS=OK  PROBLEM_TYPE=SHORTESTPATH  SOLUTION_STATUS=OK  NUM_PATHS=104263396
CPU_TIME=233171.83  REAL_TIME=90.43
```

Notice that the resulting output data tables, mycas.shortPathSummary and mycas.shortPathPaths, are large distributed data tables.

Example 3.9: Shortest Path in a Road Network by Date and Time

This example reconsiders the road network between a SAS employee's home in Raleigh, North Carolina, and SAS headquarters nearby in Cary introduced in the section "Road Network Shortest Path" on page 13. The following data provide a snapshot of the road network and travel times observed at three different times:

```
data mycas.LinkSetInRoadNC;
   input start_inter $1-20 end_inter $21-40 miles miles_per_hour
      date date11. time time10.;
   format date date11. time time10.;
   time_to_travel = miles * 1/miles_per_hour * 60;
   datalines;
614CapitalBlvd      Capital/WadeAve      0.6  25 15-APR-2013 10:30 am
614CapitalBlvd      Capital/US70W        0.6  25 15-APR-2013 10:30 am
```

```
614CapitalBlvd      Capital/US440W       3.0  45 15-APR-2013 10:30 am
Capital/WadeAve     WadeAve/RaleighExpy  3.0  40 15-APR-2013 10:30 am
Capital/US70W       US70W/US440W         3.2  60 15-APR-2013 10:30 am
US70W/US440W        US440W/RaleighExpy   2.7  60 15-APR-2013 10:30 am
Capital/US440W      US440W/RaleighExpy   6.7  60 15-APR-2013 10:30 am
US440W/RaleighExpy  RaleighExpy/US40W    3.0  60 15-APR-2013 10:30 am
WadeAve/RaleighExpy RaleighExpy/US40W    3.0  60 15-APR-2013 10:30 am
RaleighExpy/US40W   US40W/HarrisonAve    1.3  55 15-APR-2013 10:30 am
US40W/HarrisonAve   SASCampusDrive       0.5  25 15-APR-2013 10:30 am
614CapitalBlvd      Capital/WadeAve      0.6  25 16-APR-2013  9:30 am
614CapitalBlvd      Capital/US70W        0.6  25 16-APR-2013  9:30 am
614CapitalBlvd      Capital/US440W       3.0  45 16-APR-2013  9:30 am
Capital/WadeAve     WadeAve/RaleighExpy  3.0  25 16-APR-2013  9:30 am
Capital/US70W       US70W/US440W         3.2  60 16-APR-2013  9:30 am
US70W/US440W        US440W/RaleighExpy   2.7  60 16-APR-2013  9:30 am
Capital/US440W      US440W/RaleighExpy   6.7  60 16-APR-2013  9:30 am
US440W/RaleighExpy  RaleighExpy/US40W    3.0  60 16-APR-2013  9:30 am
WadeAve/RaleighExpy RaleighExpy/US40W    3.0  60 16-APR-2013  9:30 am
RaleighExpy/US40W   US40W/HarrisonAve    1.3  55 16-APR-2013  9:30 am
US40W/HarrisonAve   SASCampusDrive       0.5  25 16-APR-2013  9:30 am
614CapitalBlvd      Capital/WadeAve      0.6  25 18-APR-2013  8:30 am
614CapitalBlvd      Capital/US440W       3.0  45 18-APR-2013  8:30 am
Capital/WadeAve     WadeAve/RaleighExpy  3.0  25 18-APR-2013  8:30 am
Capital/US440W      US440W/RaleighExpy   6.7  60 18-APR-2013  8:30 am
US440W/RaleighExpy  RaleighExpy/US40W    3.0  60 18-APR-2013  8:30 am
WadeAve/RaleighExpy RaleighExpy/US40W    3.0  60 18-APR-2013  8:30 am
RaleighExpy/US40W   US40W/HarrisonAve    1.3  55 18-APR-2013  8:30 am
US40W/HarrisonAve   SASCampusDrive       0.5  25 18-APR-2013  8:30 am
;
```

The first snapshot (15-APR-2013 10:30 am) is a typical traffic pattern on a workday. The second snapshot (16-APR-2013 9:30 am) represents morning rush-hour traffic, and the third (18-APR-2013 8:30 am) represents rush-hour traffic where a major highway (US70W) has been closed for repairs.

The following statements find the route that yields the shortest path between home (614 Capital Boulevard) and SAS headquarters (SAS Campus Drive) for all three scenarios simultaneously by using the BY statement:

```
proc optnetwork
   links         = mycas.LinkSetInRoadNC;
   linksVar
      from       = start_inter
      to         = end_inter
      weight     = time_to_travel;
   shortestPath
      outPaths   = mycas.ShortPathP
      outWeights = mycas.ShortPathW
      source     = "614CapitalBlvd"
      sink       = "SASCampusDrive";
   displayout
      ProblemSummary  = ProblemSummary
      SolutionSummary = SolutionSummary;
   by date time;
run;
%put &_OROPTNETWORK_;
```

Assuming that your grid has a total of at least three cores, all three graphs are processed simultaneously through one call to PROC OPTNETWORK. The progress of the procedure is shown in Output 3.9.1.

Output 3.9.1 PROC OPTNETWORK Log: Shortest Path in a Road Network by Date and Time

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 11.
NOTE: Processing the shortest paths problem using 32 threads.
NOTE: Processing the shortest paths problem between 1 source nodes and 1 sink nodes.
NOTE: Processing the shortest paths problem used 0.00 (cpu: 0.00) seconds.
NOTE: The above message was for the following BY group:
      date=15-APR-2013 time=10:30:00
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 11.
NOTE: Processing the shortest paths problem using 32 threads.
NOTE: Processing the shortest paths problem between 1 source nodes and 1 sink nodes.
NOTE: Processing the shortest paths problem used 0.00 (cpu: 0.01) seconds.
NOTE: The above message was for the following BY group:
      date=16-APR-2013 time=9:30:00
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 8.
NOTE: Processing the shortest paths problem using 32 threads.
NOTE: Processing the shortest paths problem between 1 source nodes and 1 sink nodes.
NOTE: Processing the shortest paths problem used 0.00 (cpu: 0.00) seconds.
NOTE: The above message was for the following BY group:
      date=18-APR-2013 time=8:30:00
NOTE: The CAS table 'PROBLEMSUMMARY' in caslib 'CASUSERHDFS(magala)' has 3 rows and 5 columns.
NOTE: The CAS table 'SOLUTIONSUMMARY' in caslib 'CASUSERHDFS(magala)' has 3 rows and 7 columns.
NOTE: The Cloud Analytic Services server processed the request in 0.104125 seconds.
NOTE: The data set MYCAS.SHORTPATHP has 16 observations and 8 variables.
NOTE: The data set MYCAS.SHORTPATHW has 3 observations and 5 variables.
STATUS=OK  PROBLEM_TYPE=SHORTESTPATH  CPU_TIME=0.33  REAL_TIME=0.10
```

Output 3.9.2 displays the output table mycas.ProblemSummary, which contains a summary of each graph that is processed by PROC OPTNETWORK.

Output 3.9.2 Problem Summary by Date and Time

date	time	numNodes	numLinks	graphDirection
15-APR-2013	10:30:00	10	11	Undirected
16-APR-2013	9:30:00	10	11	Undirected
18-APR-2013	8:30:00	8	8	Undirected

Output 3.9.3 displays the output table mycas.SolutionSummary, which contains a solution summary for the processing on each graph.

Output 3.9.3 Solution Summary by Date and Time

date	time	problemType	status	numPaths	cpuTime	realTime
15-APR-2013	10:30:00	Shortest Path	OK	1	0.00	0.00
16-APR-2013	9:30:00	Shortest Path	OK	1	0.01	0.00
18-APR-2013	8:30:00	Shortest Path	OK	1	0.00	0.00

Figure 3.96 displays the output data table mycas.ShortPathW, which shows the total time to travel on the best route for each time snapshot.

Figure 3.96 Shortest Path Summary for Road Network at Each Date and Time

date	time	source	sink	path_weight
15-APR-2013	10:30:00	614CapitalBlvd	SASCampusDrive	11.5582
16-APR-2013	9:30:00	614CapitalBlvd	SASCampusDrive	12.9582
18-APR-2013	8:30:00	614CapitalBlvd	SASCampusDrive	14.2582

Figure 3.97 displays the output data table mycas.ShortPathP, which shows (by date and time) the best route for each time snapshot.

Figure 3.97 Shortest Path for Road Network by Date and Time

date=15-APR-2013 time=10:30:00

order	start_inter	end_inter	time_to_travel
1	614CapitalBlvd	Capital/WadeAve	1.4400
2	Capital/WadeAve	WadeAve/RaleighExpy	4.5000
3	WadeAve/RaleighExpy	RaleighExpy/US40W	3.0000
4	RaleighExpy/US40W	US40W/HarrisonAve	1.4182
5	US40W/HarrisonAve	SASCampusDrive	1.2000
time			**11.5582**
date			**11.5582**

date=16-APR-2013 time=9:30:00

order	start_inter	end_inter	time_to_travel
1	614CapitalBlvd	Capital/US70W	1.4400
2	Capital/US70W	US70W/US440W	3.2000
3	US70W/US440W	US440W/RaleighExpy	2.7000
4	US440W/RaleighExpy	RaleighExpy/US40W	3.0000
5	RaleighExpy/US40W	US40W/HarrisonAve	1.4182
6	US40W/HarrisonAve	SASCampusDrive	1.2000
time			**12.9582**
date			**12.9582**

Figure 3.97 *continued*

date=18-APR-2013 time=8:30:00

order	start_inter	end_inter	time_to_travel
1	614CapitalBlvd	Capital/WadeAve	1.4400
2	Capital/WadeAve	WadeAve/RaleighExpy	7.2000
3	WadeAve/RaleighExpy	RaleighExpy/US40W	3.0000
4	RaleighExpy/US40W	US40W/HarrisonAve	1.4182
5	US40W/HarrisonAve	SASCampusDrive	1.2000
time			**14.2582**
date			**14.2582**
			38.7745

References

Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice-Hall.

Applegate, D. L., Bixby, R. E., Chvátal, V., and Cook, W. J. (2006). *The Traveling Salesman Problem: A Computational Study*. Princeton, NJ: Princeton University Press.

Boitmanis, K., Freivalds, K., Ledins, P., and Opmanis, R. (2006). "Fast and Simple Approximation of the Diameter and Radius of a Graph." In *Experimental Algorithms*, vol. 4007, edited by C. Alvarez and M. Serna, 98–108. Berlin: Springer-Verlag. http://dx.doi.org/10.1007/11764298_9.

Bron, C., and Kerbosch, J. (1973). "Algorithm 457: Finding All Cliques of an Undirected Graph." *Communications of the ACM* 16:48–50.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms*. Cambridge, MA, and New York: MIT Press and McGraw-Hill.

Demetrescu, C. (2010). "9th DIMACS Implementation Challenge—Shortest Paths." The data are available at http://www.dis.uniroma1.it/challenge9/download.shtml.

Google (2011). "Google Maps." Accessed March 16, 2011. http://maps.google.com.

Harley, E. R. (2003). "Graph Algorithms for Assembling Integrated Genome Maps." Ph.D. diss., University of Toronto.

Johnson, D. B. (1975). "Finding All the Elementary Circuits of a Directed Graph." *SIAM Journal on Computing* 4:77–84.

Jonker, R., and Volgenant, A. (1987). "A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems." *Computing* 38:325–340.

Krebs, V. (2002). "Uncloaking Terrorist Networks." *First Monday* 7. http://www.firstmonday.org/issues/issue7_4/krebs/.

Kruskal, J. B. (1956). "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem." *Proceedings of the American Mathematical Society* 7:48–50.

Kumar, R., and Li, H. (1994). "On Asymmetric TSP: Transformation to Symmetric TSP and Performance Bound." http://home.engineering.iastate.edu/~rkumar/PUBS/atsp.pdf.

Leskovec, J. (2014). "SNAP: Stanford Network Analysis Project." The data are available at https://snap.stanford.edu/data/index.html.

Liu, H., and Wang, J. (2006). "A New Way to Enumerate Cycles in Graph." In *Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services*, 57–59.

Stoer, M., and Wagner, F. (1997). "A Simple Min-Cut Algorithm." *Journal of the Association for Computing Machinery* 44:585–591.

Tarjan, R. E. (1972). "Depth-First Search and Linear Graph Algorithms." *SIAM Journal on Computing* 1:146–160.

Willingham, V. (2009). "Massive Transplant Effort Pairs 13 Kidneys to 13 Patients." CNN Health. Accessed March 16, 2011. http://www.cnn.com/2009/HEALTH/12/14/kidney.transplant/index.html.

Index

ABSOBJGAP= option
TSP statement, 39
ABSOLUTEOBJECTIVEGAP= option
TSP statement, 39
ALGORITHM= option
CONNECTEDCOMPONENTS statement, 27
CYCLE statement, 28
AUXWEIGHT= option
LINKSVAR statement, 31

BICONNECTEDCOMPONENTS option
SUMMARY statement, 37
BICONNECTEDCOMPONENTS statement
statement options, 26
BY statement
OPTNETWORK procedure, 26

CASESENSITIVE option
DISPLAY statement (OPTNETWORK), 30
CLIQUE statement
statement options, 26
CONNECTEDCOMPONENTS option
SUMMARY statement, 37
CONNECTEDCOMPONENTS statement
statement options, 27
CUTOFF= option
TSP statement, 39
CUTSTRATEGY= option
TSP statement, 39
CYCLE statement
statement options, 28

DIAMETERAPPROX= option
SUMMARY statement, 37
DIRECTION= option
PROC OPTNETWORK statement, 23
DISPLAY statement
OPTNETWORK procedure, 29
DISPLAYOUT statement
OPTNETWORK procedure, 30

EXCLUDE option
DISPLAY statement (OPTNETWORK), 30
EXCLUDEALL option
DISPLAY statement (OPTNETWORK), 30

FROM= option
LINKSVAR statement, 31

HEURISTICS= option
TSP statement, 39

INCLUDEALL option
DISPLAYOUT statement (OPTNETWORK), 30
INCLUDESELFLINK option
PROC OPTNETWORK statement, 24
INDEXOFFSET= option
PROC OPTNETWORK statement, 24
INTERNALFORMAT= option
CONNECTEDCOMPONENTS statement, 27
MINCOSTFLOW statement, 32
MINSPANTREE statement, 33
PATH statement, 35

LINEARASSIGNMENT statement
statement options, 31
LINKS= option
PROC OPTNETWORK statement, 24
LINKSVAR statement
statement options, 31
LOGFREQ= option
MINCOSTFLOW statement, 32
TSP statement, 40
LOGFREQTIME= option
PROC OPTNETWORK statement, 24
LOGFREQUENCY= option
MINCOSTFLOW statement, 32
TSP statement, 40
LOGFREQUENCYTIME= option
PROC OPTNETWORK statement, 24
LOGLEVEL= option
PROC OPTNETWORK statement, 24
LOWER= option
LINKSVAR statement, 31
NODESVAR statement, 34

MAXCLIQUES= option
CLIQUE statement, 26
MAXCUTS= option
MINCUT statement, 33
MAXCYCLES= option
CYCLE statement, 28
MAXLENGTH= option
CYCLE statement, 28
PATH statement, 35
MAXLINKWEIGHT= option
CYCLE statement, 28
PATH statement, 35
MAXNODES= option

TSP statement, 40
MAXNODEWEIGHT= option
CYCLE statement, 28
PATH statement, 35
MAXPATHWEIGHT= option
SHORTESTPATH statement, 37
MAXSOLS= option
TSP statement, 40
MAXTIME= option
CLIQUE statement, 27
CYCLE statement, 28
MINCOSTFLOW statement, 32
PATH statement, 35
TSP statement, 40
MAXWEIGHT= option
MINCUT statement, 33
MILP= option
TSP statement, 40
MINCOSTFLOW statement
statement options, 32
MINCUT statement
statement options, 33
MINLENGTH= option
CYCLE statement, 28
PATH statement, 36
MINLINKWEIGHT= option
CYCLE statement, 29
PATH statement, 36
MINNODEWEIGHT= option
CYCLE statement, 29
PATH statement, 36
MINSPANTREE statement
statement options, 33

NODE= option
NODESSUBSETVAR statement, 34
NODESVAR statement, 34
NODES= option
PROC OPTNETWORK statement, 25
NODESSUBSET= option
PROC OPTNETWORK statement, 25
NODESSUBSETVAR statement
statement options, 34
NODESVAR statement
statement options, 34
NOREPLACE option
DISPLAYOUT statement (OPTNETWORK), 30
NTHREADS= option
PROC OPTNETWORK statement, 25

OPTNETWORK procedure, 17
OPTNETWORK procedure, DISPLAY statement
CASESENSITIVE option, 30
EXCLUDE option, 30
EXCLUDEALL option, 30
TRACE option, 30
OPTNETWORK procedure, DISPLAYOUT statement
INCLUDEALL option, 30
NOREPLACE option, 30
REPEATED option, 30
OUT= option
CLIQUE statement, 27
CYCLE statement, 29
LINEARASSIGNMENT statement, 31
MINCUT statement, 33
MINSPANTREE statement, 34
SHORTESTPATH statement, 37
SUMMARY statement, 38
TRANSITIVECLOSURE statement, 38
TSP statement, 40
OUTCUTSETS= option
MINCUT statement, 33
OUTLINKS= option
PROC OPTNETWORK statement, 25
OUTNODES= option
PROC OPTNETWORK statement, 25
OUTPARTITIONS= option
MINCUT statement, 33
OUTPATHS= option
PATH statement, 36
SHORTESTPATH statement, 37
OUTPATHSLINKS= option
PATH statement, 36
OUTPATHSNODES= option
PATH statement, 36
OUTWEIGHTS= option
SHORTESTPATH statement, 37

PATH statement
statement options, 35
PROC OPTNETWORK statement
statement options, 23

RELATIVEOBJECTIVEGAP= option
TSP statement, 40
RELOBJGAP= option
TSP statement, 40
REPEATED option
DISPLAYOUT statement (OPTNETWORK), 30

SHORTESTPATH statement
statement options, 36
SHORTESTPATH= option
SUMMARY statement, 38
SINK= option
NODESSUBSETVAR statement, 34
PATH statement, 36
SHORTESTPATH statement, 37
SOURCE= option

NODESSUBSETVAR statement, 34
PATH statement, 36
SHORTESTPATH statement, 37
STANDARDIZEDLABELS option
PROC OPTNETWORK statement, 25
SUMMARY statement
statement options, 37

TARGET= option
TSP statement, 41
TIMETYPE= option
PROC OPTNETWORK statement, 25
TO= option
LINKSVAR statement, 31
TRACE option
DISPLAY statement (OPTNETWORK), 30
TRANSITIVECLOSURE statement
statement options, 38
TSP statement
statement options, 39

UPPER= option
LINKSVAR statement, 31
NODESVAR statement, 35

WEIGHT= option
LINKSVAR statement, 32
NODESVAR statement, 35