# SAS® Optimization 8.5
# The OPTNETWORK Procedure

SAS® Documentation
August 18, 2020

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2019. *SAS® Optimization 8.5: The OPTNETWORK Procedure*. Cary, NC: SAS Institute Inc.

**SAS® Optimization 8.5: The OPTNETWORK Procedure**



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

August 2020



8.5-P1:casnopt

# Contents

# Chapter 1
# What's New in the OPTNETWORK Procedure in SAS Optimization 8.5

**Contents**



## Overview

SAS Optimization 8.5 contains several new features and performance enhancements in the OPTNETWORK procedure.

## OPTNETWORK Procedure Enhancements

Improving computational performance is a continuing theme for all algorithms in the OPTNETWORK procedure. In this release, many enhancements have been made to reduce the computation time for various algorithms.

The following list highlights changes that are related to graph handling for all algorithms:

- The INTERNALFORMAT= option has been deprecated. The best underlying internal structures are chosen for the requested algorithm.
- By default, MULTILINKS=TRUE for algorithms that support multilinks. Previously, this default was determined by the value of the INTERNALFORMAT= option.
- The LOADGRAPH statement enables you to load graph input data tables and build in-memory data structures for subsequent analyses.
- The UNLOADGRAPH statement enables you to delete in-memory data structures that have been loaded with the LOADGRAPH statement.

The following sections contain lists of new features and enhancements specific to a particular algorithm statement.

## CYCLE Statement

The cycle algorithm can now produce a table that contains the sequence of links in the cycles that are found when you specify the OUTCYCLESLINKS= option.

## LINEARASSIGNMENT Statement

The linear assignment problem solver now includes the MAXTIME= option, which enables you to specify the maximum amount of time to spend on the algorithm.

## MINCUT Statement

The minimum cut algorithm now includes the SOURCE=*s* and SINK=*t* options, which you can use to find a minimum *s*-*t* cut.

## SUMMARY Statement

You can now add the triangle count for undirected graphs to the graph summary output by using the CLUSTERINGCOEFFICIENT option.

# Chapter 2
# Introduction

**Contents**



## 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 *Base SAS 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: User's Guide*. 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: User's Guide*.

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 bold`** | 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 for an HTML ODS destination as follows:

```
ods html style=HTMLBlue;
```

You can also specify a style template for a PDF ODS destination as follows:

```
ods pdf style=PearlJ;
```

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

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 relationships that are based on natural co-occurrence; examples are 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 each interaction is modeled as a weight on the corresponding link 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. PROC OPTNETWORK provides a number of network analysis and optimization algorithms (listed in Table 3.1) that take an abstract graph or network as input, help explain the 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* $G = (N, E)$, which is defined over a set $N$ of nodes and a set $E$ of links. A *node* is an abstract representation of some entity (or object), and a *link* defines a relationship (or connection) between two nodes. The terms *node* and *vertex* are interchangeable in describing an entity. The term *link* is interchangeable with the term *edge* or *arc* 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 42 and "Examples: OPTNETWORK Procedure" on page 142.

## 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
```

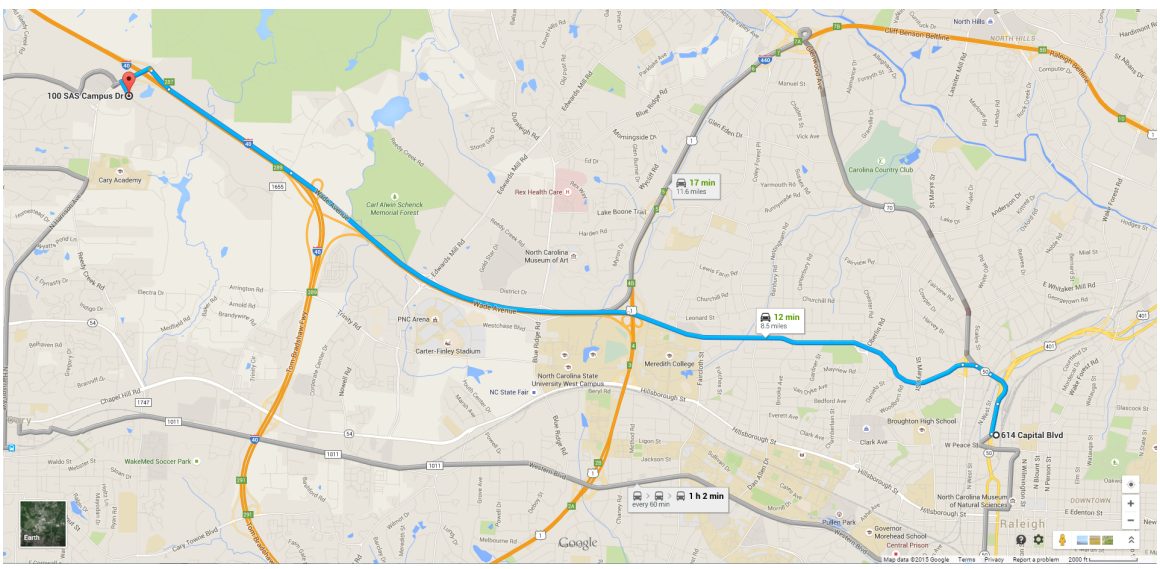
```
      outPaths = mycas.ShortPath
      source   = "614CapitalBlvd"
      sink     = "SASCampusDrive";
run;
```

For more information about shortest path algorithms in PROC OPTNETWORK, see the section "Shortest Path" on page 109. 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;
```

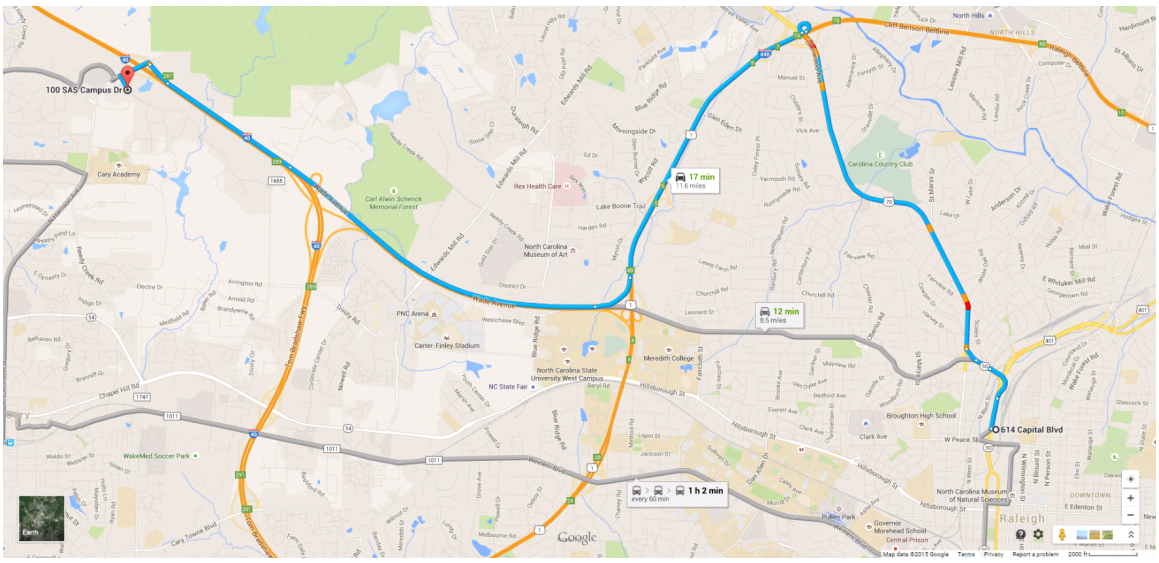
Now, the output data table mycas.ShortPath, shown in Figure 3.3, shows the best route for going home. Because the traffic on Wade Avenue is usually heavy at this time of day, the best route home is different from the best route to work.

**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* > ;
**LOADGRAPH** < *options* > ;
**NODESSUBSETVAR** < *options* > ;
**NODESVAR** < *options* > ;
**UNLOADGRAPH** < *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 | Option |
|---|---|
| PROC OPTNETWORK Statement | |
| **Input** | |
| Specifies the in-memory graph to use | GRAPH= |
| Specifies the links data table | LINKS= |
| Specifies the nodes data table | NODES= |
| Specifies the nodes subset data table | NODESSUBSET= |
| **Output** | |
| Specifies the links output data table | OUTLINKS= |
| Specifies the nodes output data table | OUTNODES= |
| **Options** | |
| Specifies whether to enforce determinism | DETERMINISTIC= |
| Specifies the graph direction | DIRECTION= |

**Table 3.2** *continued*

| Description | Option |
|---|---|
| Specifies whether to use a distributed graph | DISTRIBUTED= |
| Specifies the index offset for identifiers | INDEXOFFSET= |
| Specifies the desired frequency (in number of seconds) between log entries | LOGFREQTIME= |
| Specifies the overall log level | LOGLEVEL= |
| Specifies whether to include multilinks | MULTILINKS= |
| Specifies the maximum number of threads to use for multithreaded processing | NTHREADS= |
| Specifies whether to include self-links | SELFLINKS= |
| Specifies that the input graph data are in a standardized format | STANDARDIZEDLABELS |
| Requests that the output graph data include standardized format | STANDARDIZEDLABELSOUT |
| Specifies whether time units are in CPU time or real time | TIMETYPE= |
| **Data Input Statements** | |
| LINKSVAR Statement | |
| Specifies the data variable name for the auxiliary link weights | AUXWEIGHT= |
| Specifies the data variable name for the *from* nodes | FROM= |
| Specifies the data variable name for the link lower bounds | LOWER= |
| Specifies the data variable name for the *to* nodes | TO= |
| Specifies the data variable name for the link upper bounds | UPPER= |
| Specifies one or more data variable for the additional link attributes to carry over to the output results | VARS= |
| Specifies the data variable name for the link weights | WEIGHT= |
| LOADGRAPH Statement | |
| Specifies the output data table to contain summary information about in-memory graphs | OUTGRAPHLIST= |
| NODESSUBSETVAR Statement | |
| Specifies the data variable name for the nodes | NODE= |
| Specifies the data variable name for the sink indicator | SINK= |
| Specifies the data variable name for the source indicator | SOURCE= |
| NODESVAR Statement | |
| Specifies the data variable name for the node lower bounds | LOWER= |
| Specifies the data variable name for the nodes | NODE= |
| Specifies the data variable name for the node upper bounds | UPPER= |
| Specifies one or more data variables for the additional node attributes to carry over to the output results | VARS= |
| Specifies the data variable name for the node weights | WEIGHT= |
| UNLOADGRAPH Statement | |
| Specifies the output data table to contain summary information about in-memory graphs | OUTGRAPHLIST= |
| **Algorithm Statements** | |
| BICONNECTEDCOMPONENTS Statement | |
| Specifies the output data table for biconnected components | OUT= |

**Table 3.2** *continued*

| Description | Option |
|---|---|
| CLIQUE Statement | |
| Specifies the maximum number of cliques to return during clique enumeration | MAXCLIQUES= |
| Specifies the maximum link weight for the cliques found | MAXLINKWEIGHT= |
| Specifies the maximum node weight for the cliques found | MAXNODEWEIGHT= |
| Specifies the maximum size for the cliques found | MAXSIZE= |
| Specifies the maximum amount of time to spend finding cliques | MAXTIME= |
| Specifies the minimum link weight for the cliques found | MINLINKWEIGHT= |
| Specifies the minimum node weight for the cliques found | MINNODEWEIGHT= |
| Specifies the minimum size for the cliques found | MINSIZE= |
| Specifies the output data table for cliques | OUT= |
| CONNECTEDCOMPONENTS Statement | |
| Specifies the algorithm to use for connected components | ALGORITHM= |
| Specifies the output data table for connected components | OUT= |
| CYCLE Statement | |
| Specifies the algorithm to use for cycle enumeration | ALGORITHM= |
| Specifies the maximum number of cycles to return during cycle enumeration | MAXCYCLES= |
| Specifies the maximum length for the cycles found | MAXLENGTH= |
| Specifies the maximum link weight for the cycles found | MAXLINKWEIGHT= |
| Specifies the maximum node weight for the cycles found | MAXNODEWEIGHT= |
| Specifies the maximum amount of time to spend finding cycles | MAXTIME= |
| Specifies the minimum length for the cycles found | MINLENGTH= |
| Specifies the minimum link weight for the cycles found | MINLINKWEIGHT= |
| Specifies the minimum node weight for the cycles found | MINNODEWEIGHT= |
| Specifies the output data table for the links of the cycles | OUTCYCLESLINKS= |
| Specifies the output data table for the nodes of the cycles | OUTCYCLESNODES= |
| LINEARASSIGNMENT Statement | |
| Specifies the maximum amount of time to spend for linear assignment | MAXTIME= |
| Specifies the output data table for linear assignment | OUT= |
| MINCOSTFLOW Statement | |
| Specifies the iteration log frequency | LOGFREQ= |
| Specifies the maximum amount of time to spend calculating minimum-cost network flows | MAXTIME= |
| MINCUT Statement | |
| Specifies the maximum number of cuts to return | MAXCUTS= |
| Specifies the maximum weight of the cuts to return | MAXWEIGHT= |
| Specifies the output data table for minimum cut sets | OUTCUTSETS= |
| Specifies the output data table for minimum cut partitions | OUTPARTITIONS= |

**Table 3.2** *continued*

| Description | Option |
|---|---|
| Specifies the sink node for minimum cut calculations | SINK= |
| Specifies the source node for minimum cut calculations | SOURCE= |
| | |
| MINSPANTREE Statement | |
| Specifies the output data table for a minimum spanning tree | OUT= |
| | |
| PATH Statement | |
| Specifies the maximum length for the paths found | MAXLENGTH= |
| Specifies the maximum link weight for the paths found | MAXLINKWEIGHT= |
| Specifies the maximum node weight for the paths found | MAXNODEWEIGHT= |
| Specifies the maximum amount of time to spend finding paths | MAXTIME= |
| Specifies the minimum length for the paths found | MINLENGTH= |
| Specifies the minimum link weight for the paths found | MINLINKWEIGHT= |
| Specifies the minimum node weight for the paths found | MINNODEWEIGHT= |
| Specifies the output data table for path links | OUTPATHSLINKS= |
| Specifies the output data table for path nodes | OUTPATHSNODES= |
| Specifies the sink node for path calculations | SINK= |
| Specifies the source node for path calculations | SOURCE= |
| | |
| SHORTESTPATH Statement | |
| Specifies the maximum path weight | MAXPATHWEIGHT= |
| Specifies the output data table for shortest paths | OUTPATHS= |
| Specifies the output data table for shortest path descriptive statistics | OUTSUMMARY= |
| Specifies the output data table for shortest path weights | OUTWEIGHTS= |
| Specifies the sink node for shortest path calculations | SINK= |
| Specifies the source node for shortest path calculations | SOURCE= |
| | |
| SUMMARY Statement | |
| Calculates information about biconnected components | BICONNECTEDCOMPONENTS |
| Calculates information about clustering coefficients | CLUSTERINGCOEFFICIENT |
| Calculates information about connected components | CONNECTEDCOMPONENTS |
| Calculates the approximate diameter and chooses the weight type | DIAMETERAPPROX= |
| Includes only finite values when calculating descriptive statistics that are related to shortest paths | FINITEPATH |
| Specifies the output data table for summary results | OUT= |
| Calculates information about shortest paths and chooses the weight type | SHORTESTPATH= |
| | |
| TRANSITIVECLOSURE Statement | |
| Specifies the output data table for transitive closure results | OUT= |
| | |
| TSP Statement | |
| Specifies the stopping criterion based on the absolute objective gap | ABSOBJGAP= |
| Specifies the cutoff value for branch-and-bound node removal | CUTOFF= |
| Specifies the overall cut strategy level | CUTSTRATEGY= |

**Table 3.2** *continued*

| Description | Option |
|---|---|
| Specifies the initial and primal heuristics level | HEURISTICS= |
| Specifies the frequency of printing the branch-and-bound node log | LOGFREQ= |
| Specifies the maximum number of branch-and-bound nodes to be processed | MAXNODES= |
| Specifies the maximum number of solutions to be found | MAXSOLS= |
| Specifies the maximum amount of time to spend solving the traveling salesman problem | MAXTIME= |
| Specifies whether to use a mixed integer linear programming solver | MILP= |
| Specifies the output data table for the traveling salesman problem | OUT= |
| Specifies the stopping criterion based on the relative objective gap | RELOBJGAP= |
| Specifies the stopping criterion based on the target objective value | 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 | | |
| SINK=, SOURCE= | X | X |
| otherwise | X | |
| MINSPANTREE | X | |
| PATH | X | X |
| SHORTESTPATH | X | X |
| SUMMARY | | |
| BICONNECTEDCOMPONENTS | X | |
| 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 | OUT= |
| CLIQUE | | | OUT= |
| CONNECTEDCOMPONENTS | X | X | OUT= |
| CYCLE | | | OUTCYCLESLINKS=, OUTCYCLESNODES= |
| LINEARASSIGNMENT | | | OUT= |
| MINCOSTFLOW | | X | |
| MINCUT | | | OUTCUTSETS=, OUTPARTITIONS= |
| MINSPANTREE | | | OUT= |
| PATH | | | OUTPATHSLINKS=, OUTPATHSNODES= |
| SHORTESTPATH | | | OUTPATHS=, OUTWEIGHTS=, OUTSUMMARY= |
| 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:

**DETERMINISTIC=TRUE | FALSE**

specifies whether to enforce determinism. By default, DETERMINISTIC=TRUE, which ensures that each invocation (with the same machine configuration and parameter settings) produces the same final result. For more information about determinism, see the section "Determinism" on page 59.

**DIRECTION=DIRECTED | UNDIRECTED**

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

**DIRECTED** considers the input graph to be 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** considers the input graph to be 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 42.

**DISTRIBUTED=TRUE | FALSE**

specifies whether to use a distributed graph. By default, DISTRIBUTED=FALSE, which means that a distributed graph is not used. For more information about the algorithms that support distributed graph computation, see the section "Execution Modes and Data Movement" on page 59.

**GRAPH=***number*

specifies the in-memory graph to use. This option can be used with any algorithm that supports in-memory execution.

For more information about using the GRAPH= option, see the section "Persistent Data Structures (In-Memory Graphs)" on page 61.

**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 4 in Chapter 2, "Introduction."

For more information about this input table, see the section "Links Input Data" on page 43.

**LOGFREQTIME=***number*

**LOGFREQUENCYTIME=***number*

controls the frequency (in *number* of seconds) for displaying iteration logs for some algorithms, where *number* can be any integer greater than or equal to 1. This option is useful for computationally intensive algorithms. Setting *number* too low can hurt algorithm performance. This option is ignored for a single-server setup. 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.

**MULTILINKS=TRUE | FALSE**

specifies whether to include or aggregate multilinks when an input graph (specified by the LINKS= option) is read. You can specify the following values:

**FALSE** aggregates multilinks.

**TRUE** includes multilinks.

By default, MULTILINKS=TRUE for algorithms that support multilinks.

For more information about this option, see the section "Multigraphs" on page 48.

**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 4 in Chapter 2, "Introduction."

For more information about this input table, see the section "Nodes Input Data" on page 46.

**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 4 in Chapter 2, "Introduction."

For more information about this input table, see the section "Nodes Subset Input Data" on page 47.

**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 1024, 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 4 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 4 in Chapter 2, "Introduction."

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

**SELFLINKS=TRUE | FALSE**

specifies whether to include or remove self-links when an input graph (specified by the LINKS= option) is read. You can specify the following values:

**FALSE** removes self-links.

**TRUE** includes self-links.

By default, SELFLINKS=FALSE.

For more information about this option, see the section "Self-Links" on page 51.

**STANDARDIZEDLABELS**

specifies that the input graph data are in a standardized format, as described in the section "Standardized Labels Input" on page 55.

**STANDARDIZEDLABELSOUT**

specifies that the output graph data include standardized format, as described in the section "Standardized Labels Output" on page 57.

**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 70.

**OUT=***CAS-libref.data-table*
specifies the output data table to contain the biconnected components 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 4 in Chapter 2, "Introduction."

## 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 specified BY variables must come from the LINKS= data table. The BY statement in PROC OPTNETWORK is not supported when multiple input data tables are used. An example of using BY-group processing is shown in "Example 3.9: Shortest Path in a Road Network by Date and Time" on page 166.

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 74.

You can specify the following *options*:

**MAXCLIQUES=***number* | **ALL**
specifies the maximum number of cliques for clique enumeration to return. 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.

**MAXLINKWEIGHT=***number*
specifies the maximum sum of link weights in a clique. Any clique whose sum of link weights is greater than *number* is removed from the results. In the case of a multigraph, all of the links in the induced subgraph are included in the sum. The default is the largest number that can be represented by a double, which causes no cliques to be removed from the results.

**MAXNODEWEIGHT=***number*
specifies the maximum sum of node weights in a clique. Any clique 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 cliques to be removed from the results.

**MAXSIZE=***number*
specifies the maximum number of nodes in a clique. Any clique whose size 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 cliques to be removed from the results.

**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.

**MINLINKWEIGHT=***number*
specifies the minimum sum of link weights in a clique. Any clique 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 cliques to be removed from the results.

**MINNODEWEIGHT=***number*
specifies the minimum sum of node weights in a clique. Any clique 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 cliques to be removed from the results.

**MINSIZE=***number*
specifies the minimum number of nodes in a clique. Any clique that has fewer nodes than *number* is removed from the results. By default, MINSIZE=1 and no cliques are removed from the results.

**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 4 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 77. 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** uses the union-find algorithm for undirected graphs and the depth-first search algorithm for directed graphs.

**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. You can also enable this functionality by specifying DISTRIBUTED=TRUE in the PROC OPTNETWORK statement.

**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.

**OUT=***CAS-libref.data-table*
specifies the output data table to contain the connected components 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 4 in Chapter 2, "Introduction."

## 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 82.

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 for cycle enumeration to return. 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.

**OUTCYCLESLINKS=***CAS-libref.data-table*
specifies the output data table to contain the links of 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 4 in Chapter 2, "Introduction."

**OUTCYCLESNODES=***CAS-libref.data-table*

**OUT=***CAS-libref.data-table*

specifies the output data table to contain the nodes of 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 4 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 141. 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.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 141. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

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 166.

## 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 88.

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

**MAXTIME=***number*
specifies the maximum amount of time to spend for linear assignment. 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 solution to the linear assignment 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 4 in Chapter 2, "Introduction."

## 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 43.

You can specify the following *options*:

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

**FROM=***column*

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

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

**TO=***column*

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

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

**VARS=(***columns***)**
specifies the names of one or more additional data variables to carry over to the output results. The value of the variables can be numeric or character.

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

## LOADGRAPH Statement

**LOADGRAPH** < *options* > **;**

The LOADGRAPH statement reads the input graph from tables that are specified by the LINKS= option or the NODES= option (or both) in the PROC OPTNETWORK statement, and it retains the graph in memory (within the current CAS session). The LOADGRAPH statement is described further in the section "Persistent Data Structures (In-Memory Graphs)" on page 61.

You can specify the following *options*:

**OUTGRAPHLIST=***CAS-libref.data-table*
specifies the output data table to contain summary information about in-memory graphs.

## 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 89.

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

**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, and 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, and 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 97.

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 to be returned by the algorithm. 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.

**SINK=***t*

specifies the sink node for minimum cut calculations. If this option is specified, the SOURCE= option must also be specified.

**SOURCE=***s*
specifies the source node for minimum cut calculations. If this option is specified, the SINK= option must also be specified.

## 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 103.

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

**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 46.

You can specify the following *options*:

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

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

**SOURCE=***column*
specifies the name of the data variable for the source indicator. The value of the 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 46.

You can specify the following *options*:

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

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

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

**VARS=(***column(s)***)**
specifies the names of one or more additional data variables to carry over to the output results. The value of the variables can be numeric or character.

**WEIGHT=***column*
specifies the name of the data variable for the node weights. The value of the 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 105.

You can specify the following *options*:

**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 4 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 4 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 109.

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 4 in Chapter 2, "Introduction."

**OUTSUMMARY=***CAS-libref.data-table*

specifies the output data table to contain descriptive statistics of the finite shortest paths for each source. *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 4 in Chapter 2, "Introduction."

**OUTWEIGHTS=***CAS-libref.data-table*

specifies the output data table to contain the total weight of the shortest path for each source-sink pair. *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 4 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 121.

You can specify the following *options*:

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

**CLUSTERINGCOEFFICIENT**
**CLUSTERINGCOEF**
calculates information about clustering coefficients. 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 DIAMETERAPPROX=WEIGHT and DIAMETERAPPROX=UNWEIGHT both produce the same results (if you use 1.0 for each link weight). This option works only for undirected graphs.

**FINITEPATH**
includes only finite values when calculating descriptive statistics that are related to shortest paths (eccentricity, diameter, and so on).

**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 4 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 SHORTESTPATH=WEIGHT and SHORTESTPATH=UNWEIGHT both produce 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 127.

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 4 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 130. The algorithm that is used to solve this problem is built around the same method that the OPTMILP procedure uses: a branch-and-cut algorithm. Many of the following options are the same as those described for the OPTMILP procedure in the *SAS Optimization: Mathematical Optimization Procedures*.

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. TSP-specific cutting planes are always generated. You can specify the following values:

**AUTOMATIC** generates cutting planes on the basis of a strategy that is 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**

determines how frequently to apply primal heuristics during the branch-and-bound tree search, and 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 primal heuristics.

**BASIC** applies basic primal heuristics at low frequency.

**MODERATE** applies most primal heuristics at moderate frequency.

**AGGRESSIVE** applies all primal heuristics at high frequency.

By default, HEURISTICS=AUTOMATIC.

**LOGFREQ=***number*

**LOGFREQUENCY=***number*

specifies the time interval (in seconds) for printing information in the node log, where *number* can be any integer greater than or equal to 0. If *number* is 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. By default, LOGFREQ=5.

**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 that are 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 (TSP). The MILP solver attempts to find the overall best TSP tour by using a branch-and-cut 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 4 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 the relative objective gap becomes less than *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.

## UNLOADGRAPH Statement

**UNLOADGRAPH** < *options* > **;**

The UNLOADGRAPH statement deletes the in-memory graph that is specified in the GRAPH= option in the PROC OPTNETWORK statement. The UNLOADGRAPH statement is described further in the section "Persistent Data Structures (In-Memory Graphs)" on page 61.

You can specify the following *options*:

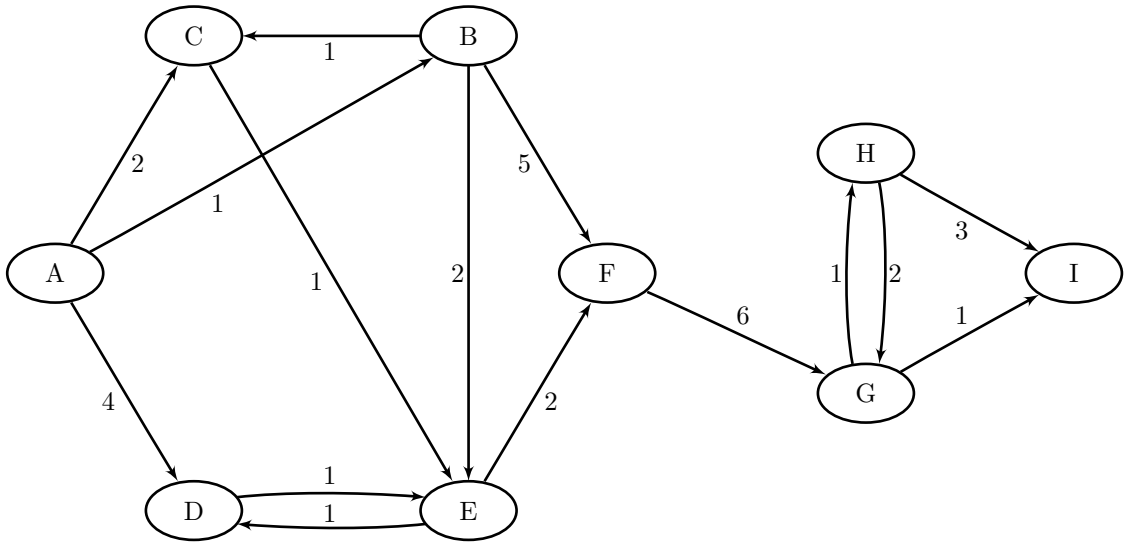
**OUTGRAPHLIST=***CAS-libref.data-table*
specifies the output data table to contain summary information about in-memory graphs.

# Details: OPTNETWORK Procedure

## Graph Input Data

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

**Figure 3.5** 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 the following possible variables:

- from: the *from* node (can be numeric or character)
- to: the *to* node (can be numeric or character)

The links data table can also contain some combination of the following built-in variables (attributes):

- 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 require 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 for a built-in variable, you must identify the variables by using the LINKSVAR statement, as described in the section "LINKSVAR Statement" on page 31.

In addition, the links data table can contain any number of user-defined additional variables (attributes). The attributes that are defined in the VARS= option in the LINKSVAR statement are carried over to the resulting output data tables.

For example, the two data tables that are created by the following DATA steps 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* that is shown in Figure 3.5 can be represented by the links data table, mycas.LinkSetIn, that is created by the following DATA step:

```
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 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 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** | G | I | 1 |
| **14** | H | G | 2 |
| **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 multilinks are removed by default. PROC OPTNETWORK aggregates the attributes of each multilink by taking the minimum value for each attribute. By default, DIRECTION=UNDIRECTED, so you can simply 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 shows the number of links that were declared as multilinks and aggregated.

**Figure 3.8** PROC OPTNETWORK Log: Links Data Table of an Undirected Graph

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: The number of nodes in the input graph is 9.
NOTE: The number of links in the input graph is 15.
NOTE: The Cloud Analytic Services server processed the request in 0.110349 seconds.
NOTE: The data set MYCAS.NODESETOUT has 9 observations and 1 variables.
NOTE: The data set MYCAS.LINKSETOUT has 15 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 multilinks are aggregated.

**Figure 3.9** Links Data Table of an 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 | D | 1 |
| 10 | E | F | 2 |
| 11 | F | G | 6 |
| 12 | G | H | 1 |
| 13 | G | I | 1 |
| 14 | H | G | 2 |
| 15 | H | I | 3 |

The MULTILINKS= option can be used to include (rather than aggregate) multilinks. For more information about this option, see the section "Multigraphs" on page 48.

## 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 the following variable:

- node: the node label (can be numeric or character)

The nodes data table can also contain some combination of the following built-in variables (attributes):

- 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)

If weights are not given for algorithms that require node weights, all weights are assumed to be 1.

You can specify any value that you want for the data table variable name. If you use a nonstandard name for a built-in variable, you must identify the variable by using the NODESVAR statement, as described in the section "NODESVAR Statement" on page 35.

In addition, the nodes data table can contain any number of user-defined additional variables (attributes). The attributes that are defined in the VARS= option in the NODESVAR statement are carried over to the resulting output data tables.

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.

The graph is defined as the union of the set of nodes that are specified in the nodes data table and the links data table. The associated attributes for nodes in the links data table that are not specified in the nodes data table default to 0 (for numeric attributes) or to an empty character string (for character attributes). The exception to this rule is the lower and upper variables for the MINCOSTFLOW algorithm, as described in "Minimum-Cost Network Flow" on page 89.

## 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 105, and "Shortest Path" on page 109, 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 code, which creates 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 $\{F\}$).

## Multigraphs

A *multigraph* is a graph that allows multiple (also called parallel) links between nodes (called *multilinks*). A graph that has no multilinks is called a *simple* graph (or just a graph). You can specify whether to include multilinks (MULTILINKS=TRUE) or to aggregate them (MULTILINKS=FALSE) when an input graph is read. For MULTILINKS=FALSE, each multilink is aggregated into one link that uses the minimum of each attribute value. That is, a multigraph is transformed into a simple graph. By default, MULTILINKS=TRUE (multilinks are included for algorithms that support them).

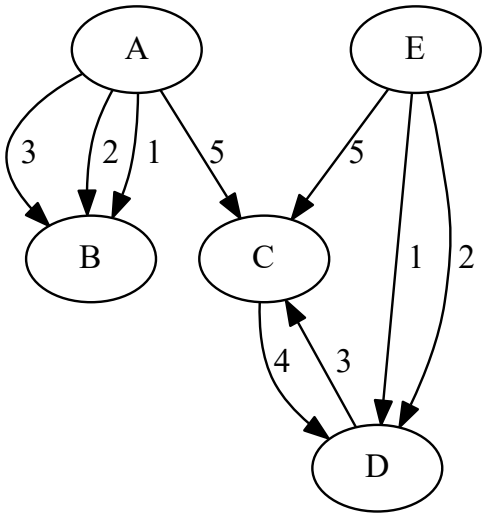
Aggregating links (MULTILINKS=FALSE) implies a performance cost. When multilinks are aggregated into a simple graph, the in-memory graph representation requires more memory and can take longer to build, relative to using MULTILINKS=TRUE. However, if your data contain many redundant parallel links, then using MULTILINKS=FALSE can improve overall performance by reducing the size of the working graph.

For certain algorithms in PROC OPTNETWORK, multigraphs are not supported. The algorithms that are specified by the following options use MULTILINKS=FALSE by default and issue an error message if you set MULTILINKS=TRUE:

- CLIQUE
- CYCLE
- MINCUT
- TSP

Consider the directed multigraph shown in Figure 3.10.

**Figure 3.10** Directed Multigraph




The links data table is created by the following DATA step:

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

The following statements read in this graph, declare it as a directed (simple) graph, and output the resulting links:

```
proc optnetwork
   direction  = directed
   multiLinks = false
   links      = mycas.LinkSetIn
   outLinks   = mycas.LinkSetOut;
run;
```

The progress of the procedure is shown in Figure 3.11. The log now shows the number of links that were declared as multilinks and aggregated.

**Figure 3.11** PROC OPTNETWORK Log: Links Data Table of a Directed Graph

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: Multilink aggregation removed 3 links from the input graph.
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 6.
NOTE: The Cloud Analytic Services server processed the request in 0.040482 seconds.
NOTE: The data set MYCAS.LINKSETOUT has 6 observations and 3 variables.
```

The output data table mycas.LinkSetOut, shown in Figure 3.12, contains the transformed (simple) graph from the input multigraph with attribute values aggregated (using the minimum attribute).

**Figure 3.12** Links Data Table of a Directed Graph

| Obs | from | to | weight |
|---|---|---|---|
| **1** | A | B | 1 |
| **2** | A | C | 5 |
| **3** | C | D | 4 |
| **4** | D | C | 3 |
| **5** | E | C | 5 |
| **6** | E | D | 1 |

The following statements read in this graph, declare it as a directed multigraph, and output the resulting links:

```
proc optnetwork
   direction  = directed
   multiLinks = true
   links      = mycas.LinkSetIn
   outLinks   = mycas.LinkSetOut;
run;
```

The progress of the procedure is shown in Figure 3.13. In this case, no links are aggregated and the resulting output data table mycas.LinkSetOut is equivalent to the input data table mycas.LinkSetIn.

**Figure 3.13** PROC OPTNETWORK Log: Links Data Table of a Directed Multigraph

```
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 9.
NOTE: The Cloud Analytic Services server processed the request in 0.053857 seconds.
NOTE: The data set MYCAS.LINKSETOUT has 9 observations and 3 variables.
```

When you want to transform your multigraph into a simple graph with finer control over how the aggregation step is performed, one possible approach is to use the FEDSQL procedure. The following statements use PROC FEDSQL to aggregate multilinks by summing their weights:

```
proc fedsql sessref=mysess;
   create table LinkSetAgg as
   select "from", "to", sum(weight) as weight
   from LinkSetIn
   group by "from", "to";
quit;
```

The resulting output data table mycas.LinkSetAgg, shown in Figure 3.14, can now be used as a simple graph in a subsequent PROC OPTNETWORK call. Using PROC FEDSQL to do the transformation of a multigraph to a simple graph provides greater flexibility in defining how the aggregation step is performed. For example, in place of the **sum** operator, you can use any of the operators that PROC FEDSQL supports. For more information about PROC FEDSQL, see *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*.

**Figure 3.14** Links Data Table after Aggregation

| Obs | from | to | WEIGHT |
|---|---|---|---|
| 1 | A | B | 6 |
| 2 | C | D | 4 |
| 3 | E | D | 3 |
| 4 | A | C | 5 |
| 5 | D | C | 3 |
| 6 | E | C | 5 |

## Self-Links

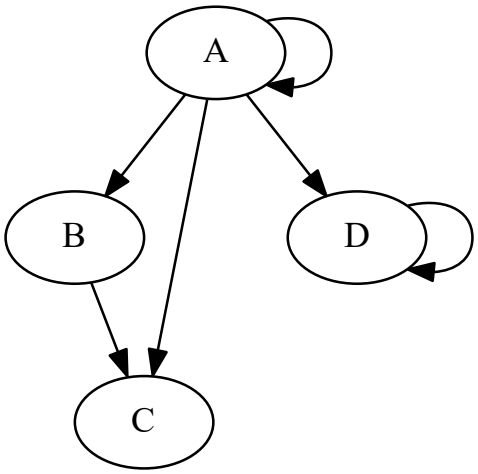
A *self-link* is a link for which the *from* node and *to* node are the same. You can specify whether to include self-links (SELFLINKS=TRUE) or to remove them (SELFLINKS=FALSE) when an input graph is read. By default, self-links are removed.

For certain algorithms in PROC OPTNETWORK, self-links are not supported. The following algorithm issues an error message if you set SELFLINKS=TRUE:

- TSP

Consider the directed graph shown in Figure 3.15.

**Figure 3.15** Directed Graph




The links data table is created by the following DATA step:

```
data mycas.LinkSetIn;
   input from $ to $;
   datalines;
A A
A B
A C
A D
B C
D D
;
```

The following statements read in this graph, declare it as a directed graph that does not allow self-links, and output the resulting links:

```
proc optnetwork
   direction  = directed
   selfLinks  = false
   links      = mycas.LinkSetIn
   outLinks   = mycas.LinkSetOut;
run;
```

The progress of the procedure is shown in Figure 3.16. The log shows the number of self-links that were ignored and therefore removed.

**Figure 3.16** PROC OPTNETWORK Log: Links Data Table of a Directed Graph

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: The graph contains 2 self-links that are ignored.
NOTE: The number of nodes in the input graph is 4.
NOTE: The number of links in the input graph is 4.
NOTE: The Cloud Analytic Services server processed the request in 0.087633 seconds.
NOTE: The data set MYCAS.LINKSETOUT has 4 observations and 2 variables.
```

The output data table mycas.LinkSetOut, shown in Figure 3.17, contains the remaining links after self-links have been removed.

**Figure 3.17** Links Data Table of a Directed Graph

| Obs | from | to |
|---|---|---|
| 1 | A | B |
| 2 | A | C |
| 3 | A | D |
| 4 | B | C |

The following statements read in the graph that is shown in Figure 3.15, declare it as a directed graph that allows self-links, and output the resulting links:

```
proc optnetwork
   direction  = directed
   selfLinks  = true
   links      = mycas.LinkSetIn
   outLinks   = mycas.LinkSetOut;
run;
```

The progress of the procedure is shown in Figure 3.18. In this case, no links are removed and the resulting output data table mycas.LinkSetOut is equivalent to the input data table mycas.LinkSetIn.

**Figure 3.18** PROC OPTNETWORK Log: Links Data Table of a Directed 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 6.
NOTE: The Cloud Analytic Services server processed the request in 0.084762 seconds.
NOTE: The data set MYCAS.LINKSETOUT has 6 observations and 2 variables.
```

## Output Carryover Variables

Any link or node attribute can be carried over to the output results table by using the VARS= option in the LINKSVAR or NODESVAR statement. This is supported for all output tables except the table that is specified in the OUTNODES= option when used in conjunction with a distributed graph algorithm. The list of algorithms that operate on a distributed graph is described in the section "Execution Modes and Data Movement" on page 59.

The following DATA step creates a graph that has four nodes (with two additional attributes) and three links (with a weight attribute and two additional attributes):

```
data mycas.NodeSetIn;
   input node $ attrStr $ attrNum;
   datalines;
A ThisIsA   13
B B          1
C LabelC  55.5
D NodeD      7
;
data mycas.LinkSetIn;
   input from $ to $ weight attrStr1 $ attrStr2 $12.;
   datalines;
A B 1 Link1 555-789-1234
A C 2 Link2 556-453-7456
B C 1 Link3 800-123-7787
;
```

The following statements read in this graph and output the resulting nodes and links data tables (including the carryover variables):

```
proc optnetwork
   nodes    = mycas.NodeSetIn
   links    = mycas.LinkSetIn
   outNodes = mycas.NodeSetOut
   outLinks = mycas.LinkSetOut;
   nodesVar
      vars  = (attrStr attrNum);
   linksVar
      vars  = (attrStr1 attrStr2);
run;
```

The nodes data table mycas.NodeSetOut, shown in Figure 3.19, contains the carryover variables that are defined in the VARS= option in the NODESVAR statement.

**Figure 3.19** Nodes Data Table with Carryover Variables

| Obs | node | attrStr | attrNum |
|---|---|---|---|
| 1 | A | ThisIsA | 13.0 |
| 2 | B | B | 1.0 |
| 3 | C | LabelC | 55.5 |
| 4 | D | NodeD | 7.0 |

The links data table mycas.LinkSetOut, shown in Figure 3.20, contains the carryover variables that are defined in the VARS= option in the LINKSVAR statement, as well as the built-in variable weight.

**Figure 3.20** Links Data Table with Carryover Variables

| Obs | from | to | weight | attrStr1 | attrStr2 |
|---|---|---|---|---|---|
| 1 | A | B | 1 | Link1 | 555-789-1234 |
| 2 | A | C | 2 | Link2 | 556-453-7456 |
| 3 | B | C | 1 | Link3 | 800-123-7787 |

## Standardized Labels Input

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 3.21.

**Figure 3.21** PROC OPTNETWORK Log: 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 0.111537 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 3.22, contains the unique numeric node labels, $\{0, 1, 3, 5\}$.

**Figure 3.22** Nodes Data Table of a Directed Graph

| Obs | node |
|---|---|
| 1 | 0 |
| 2 | 3 |
| 3 | 5 |
| 4 | 1 |

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 3.23.

**Figure 3.23** PROC OPTNETWORK Log: 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 0.076967 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 3.24, now contains all node labels from 0 to 5, based on the assumptions when you use the STANDARDIZEDLABELS option.

**Figure 3.24** Nodes Data Table of a Directed Graph

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

## Standardized Labels Output

You can specify the STANDARDIZEDLABELSOUT option in the PROC OPTNETWORK statement to generate a mapping between node labels and node identifiers that satisfies the requirements needed to use the STANDARDIZEDLABELS option in subsequent calls. Specifying the STANDARDIZEDLABELSOUT option can be useful to avoid repeated calls to the processing stage where the mapping is created. The standardized labels are written to the output links data table (or to the output nodes data table or to both) that you specify in the OUTLINKS= (OUTNODES=) option in the PROC OPTNETWORK statement.

The OUTLINKS= data table contains the following columns in addition to the variables that are described in the section "Links Input Data" on page 43:

- fromId: the *from* node identifier (a mapping to the label in the from column)
- toId: the *to* node identifier (a mapping to the label in the to column)

The OUTNODES= data table contains the following column in addition to the variables that are described in the section "Nodes Input Data" on page 46:

- nodeId: the node identifier (a mapping to the label in the node column)

Consider the following nodes and links data tables:

```
data mycas.NodeSetIn;
   input node $;
   datalines;
Z
;
data mycas.LinkSetIn;
   input from $ to $;
   datalines;
B C
B D
X Y
;
```

The following statements create the standardized mapping and output the results in the data tables mycas.NodeSetSLabels and mycas.LinkSetSLabels:

```
proc optnetwork
   nodes    = mycas.NodeSetIn
   links    = mycas.LinkSetIn
   outNodes = mycas.NodeSetSLabels
   outLinks = mycas.LinkSetSLabels
   standardizedLabelsOut;
run;
```

The output data table mycas.NodeSetSLabels, shown in Figure 3.25, contains the node mapping.

**Figure 3.25** Nodes Data Table with Standardized Label Mapping

| Obs | node | nodeId |
|---|---|---|
| 1 | B | 0 |
| 2 | C | 1 |
| 3 | D | 2 |
| 4 | X | 3 |
| 5 | Y | 4 |
| 6 | Z | 5 |

The output data table mycas.LinkSetSLabels, shown in Figure 3.26, contains the node mapping that is applied to each link.

**Figure 3.26** Links Data Table with Standardized Label Mapping

| Obs | from | to | fromId | toId |
|---|---|---|---|---|
| 1 | B | C | 0 | 1 |
| 2 | B | D | 0 | 2 |
| 3 | X | Y | 3 | 4 |

In subsequent calls, the mapping can be used with the STANDARDIZEDLABELS option, and the LINKSVAR (and NODESVAR) statements, to avoid the expense of the input processing stage. It also might be useful to use the VARS= option in the LINKSVAR (and NODESVAR) statements, as described in the section "Output Carryover Variables" on page 54, to carry over the mapping, as follows:

```
proc optnetwork
   standardizedLabels
   nodes    = mycas.NodeSetSLabels
   links    = mycas.LinkSetSLabels
   outNodes = mycas.NodeSetOut
   outLinks = mycas.LinkSetOut;
   nodesVar
      node  = nodeId
      vars  = (node);
   linksVar
      from  = fromId
      to    = toId
      vars  = (from to);
   connectedComponents;
run;
```

The output data table mycas.NodeSetOut, shown in Figure 3.27, contains the assignments of nodes to connected components, in addition to the label (identifier) mappings.

**Figure 3.27** Nodes Data Table Including Connected Components and Mappings

| Obs | nodeId | node | concomp |
|---|---|---|---|
| **1** | 0 | B | 1 |
| **2** | 1 | C | 1 |
| **3** | 2 | D | 1 |
| **4** | 3 | X | 2 |
| **5** | 4 | Y | 2 |
| **6** | 5 | Z | 3 |

The output data table mycas.LinkSetOut, shown in Figure 3.28, contains the assignments of links to connected components, in addition to the label (identifier) mappings.

**Figure 3.28** Links Data Table Including Connected Components and Mappings

| Obs | fromId | toId | from | to | concomp |
|---|---|---|---|---|---|
| **1** | 0 | 1 | B | C | 1 |
| **2** | 0 | 2 | B | D | 1 |
| **3** | 3 | 4 | X | Y | 2 |

## Determinism

Many algorithms are sensitive to the order in which PROC OPTNETWORK loads the data. Reading data tables in the same order at each invocation is not guaranteed. If the order of the nodes or links is different, the final result might change. By default, PROC OPTNETWORK ensures that each invocation (with the same machine configuration and parameter settings) produces the same final result. However, this comes at a performance cost, which can be avoided by specifying DETERMINISTIC=FALSE in the PROC OPTNETWORK statement. Specifying DETERMINISTIC=FALSE might improve performance, but the final results might differ. In some cases, this difference is simply a permutation of identifiers (for example, connected components). In other cases, when you use discrete branching decisions (for example, the traveling salesman problem), the final result might be a local (or alternative) solution.

## Execution Modes and Data Movement

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

The data movement and execution modes for each algorithm are listed in Table 3.5. The following abbreviations are used in Table 3.5 for processing mode:

- SM: single machine
- MM: multiple machines
- MT: multithreaded execution

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. The following phrases are used in Table 3.5 to describe data movements for a multiple-machine CAS server:

- Moved to SM: Some algorithms run only in single-machine mode. In such cases, one particular machine (chosen randomly) is assigned the role of the processing machine, and the data from all the other machines are moved over to this processing machine.
- Repeated on MM: 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.
- Shuffled across MM: Some algorithms use multiple machines and require only a portion of the data (distributed graphs). 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.
- None: Some algorithms require no data movement and process the original data (randomly distributed) directly.

**Table 3.5** Execution Modes and Data Movement

| Statement and Options | Data Movement | Processing Mode |
|---|---|---|
| PROC OPTNETWORK (input/output only) | | |
| DISTRIBUTED=FALSE | Moved to SM (MT) | SM (MT) |
| DISTRIBUTED=TRUE | None | MM (MT) |
| BICONNECTEDCOMPONENTS | Moved to SM | SM |
| BY | Shuffled across MM | MM (MT) |
| CLIQUE | Moved to SM | SM (MT) |
| CONNECTEDCOMPONENTS ALGORITHM= | | |
| DFS, UNIONFIND | Moved to SM | SM |
| PARALLEL† | None | 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 |

† To specify a distributed graph algorithm, you can also set DISTRIBUTED=TRUE in the PROC OPTNETWORK statement.

In addition, on each machine, some algorithms (in addition to the input phase) take advantage of multicore chip technology by executing on multiple threads simultaneously. You can use the NTHREADS=*number* 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). Specifying 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 partition your input data in advance by using the PARTITION= option in a DATA step. Partitioning in advance avoids the need for PROC OPTNETWORK to shuffle the data.

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, the CAS session is configured for four worker nodes, each having 32 cores, unless otherwise noted. For general information about CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*.

## Persistent Data Structures (In-Memory Graphs)

For large-scale graphs, loading the graph input data tables and building in-memory data structures can be computationally expensive. When the input data are not changing frequently and your workflow consists of multiple analyses, it might be more efficient to retain the in-memory data structures. You can do this by using the LOADGRAPH statement along with any other options that define the graph input. The options that define the graph input include the DIRECTION=, LINKS=, NODES=, MULTILINKS=, SELFLINKS=, and STANDARDIZEDLABELS options in the PROC OPTNETWORK statement, in addition to the LINKSVAR and NODESVAR statements.

### Loading and Unloading the Graph

Loading the graph (via the LOADGRAPH statement) consists of reading the graph input data tables and storing a standard representation of the graph in memory as part of the current CAS session. The resulting macro variable, _OROPTNETWORK_, includes GRAPH, which is a reference identifier for the in-memory graph. You can then use this identifier in the GRAPH= option in the PROC OPTNETWORK statement for subsequent calls to any supporting algorithm without needing to read from tables. Subsequent calls that include the GRAPH= option directly use the in-memory graph that was defined in the loading stage. This means that you cannot use options that are related to defining the graph in conjunction with the GRAPH= option.

When the GRAPH= option is used, additional data structures might need to be built, depending on the algorithm chosen, when a particular algorithm is first invoked. In some cases, this is an additional one-time expense. Subsequent calls can then use these data structures directly. For this reason, a first call that includes the GRAPH= option to some algorithms can sometimes take longer than subsequent calls.

You can load any number of graph input tables into memory. You can delete an in-memory graph and its persistent data structures by using the UNLOADGRAPH statement or by closing the CAS session.

### OUTGRAPHLIST= Option

The OUTGRAPHLIST= option in the LOADGRAPH and UNLOADGRAPH statements produces a data table that contains summary information about the in-memory graphs that are currently loaded (or unloaded) in the current CAS session. This data table contains the following columns:

- graph: in-memory graph identifier
- createTime: the creation time of the in-memory graph
- loaded: 1 if the graph has been loaded; 0 if the graph has been unloaded
- direction: the direction of the in-memory graph
- nodes: the number of nodes in the in-memory graph
- links: the number of links in the in-memory graph
- multiLinks: 1 if the in-memory graph load used MULTILINKS=TRUE; 0 otherwise
- selfLinks: 1 if the in-memory graph load used SELFLINKS=TRUE; 0 otherwise
- standardizedLabels: 1 if the in-memory graph load used STANDARDIZEDLABELS; 0 otherwise

### Supported Algorithms

The section "Execution Modes and Data Movement" on page 59 describes the execution modes for each algorithm and feature combination. Using persistent data structures is supported for all algorithms where data are moved to a single machine (Moved to SM) or repeated on multiple machines (Repeated on MM). Persistence of data structures is not supported when data are shuffled across multiple machines (Shuffled across MM) nor for algorithms that move no data (None).

### Using Persistent Data Structures with the Shortest Path Algorithm

As an example, consider the road network that is described in the section "Road Network Shortest Path" on page 12 and is defined by the data table mycas.LinkSetInRoadNC10am. To find the route that yields the shortest path between **source="614CapitalBlvd"** and **sink="SASCampusDrive"**, you can use the following statements:

```
proc optnetwork
   logLevel    = aggressive
   links       = mycas.LinkSetInRoadNC10am;
   linksVar
      from     = start_inter
```

```
      to       = end_inter
      weight   = time_to_travel;
   shortestPath
      outPaths = mycas.ShortPath
      source   = "614CapitalBlvd"
      sink     = "SASCampusDrive";
run;
```

These statements read the graph input from the table mycas.LinkSetInRoadNC10am, build the in-memory data structures that are required to run the shortest path algorithm, calculate the shortest path, and then delete the in-memory data structures that were needed to represent the graph data.

Next, you might want to calculate the shortest path between two other locations—for example, **source="US70W/US440W"** and **sink="SASCampusDrive"**. In this case, you would use the following statements:

```
proc optnetwork
   logLevel    = aggressive
   links       = mycas.LinkSetInRoadNC10am;
   linksVar
      from     = start_inter
      to       = end_inter
      weight   = time_to_travel;
   shortestPath
      outPaths = mycas.ShortPath
      source   = "US70W/US440W"
      sink     = "SASCampusDrive";
run;
```

These statements read the graph input from the table mycas.LinkSetInRoadNC10am, build the in-memory data structures, calculate the shortest path, and then delete the in-memory data structures. In this case, the graph input table and the in-memory data structures that are built are the same in both executions. The processing time that is needed to execute the second call can be reduced by using the LOADGRAPH statement to store the in-memory data structures.

First, load the graph by using the following statements:

```
proc optnetwork
   logLevel        = aggressive
   links           = mycas.LinkSetInRoadNC10am;
   linksVar
      from         = start_inter
      to           = end_inter
      weight       = time_to_travel;
   loadGraph
      outGraphList = mycas.OutGraphList;
run;
%put &_OROPTNETWORK_;
```

The macro variable result is written to the log, as shown in Figure 3.29.

**Figure 3.29** PROC OPTNETWORK Log: Loading Graph into Memory

```
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 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 11.
NOTE: The Cloud Analytic Services server processed the request in 0.152471 seconds.
NOTE: The data set MYCAS.OUTGRAPHLIST has 1 observations and 9 variables.
STATUS=OK  PROBLEM_TYPE=LOADGRAPH  SOLUTION_STATUS=OK  GRAPH=0  CREATETIME=23OCT2019:12:06:23
CPU_TIME=0.20  REAL_TIME=0.15
```

The macro variable indicates that the graph identifier (GRAPH) for the in-memory graph is 0. The following statements extract the identifier from the _OROPTNETWORK_ macro variable into the _GRAPH_ macro variable:

```
%macro GetValue(mac=, item=);
   %let prs = %sysfunc(prxparse(m/\b&item=/i));
   %if %sysfunc(prxmatch(&prs, &&&mac)) %then %do;
      %let prs = %sysfunc(prxparse(s/.*\b&item=([^ ]+).*/$1/i));
      %let return_val = %sysfunc(prxchange(&prs, 1, &&&mac));
      &return_val
   %end;
   %else %do;
      %put ERROR: Cannot find &item!;
      .
   %end;
%mend GetValue;
%let _GRAPH_ = %GetValue(mac=_OROPTNETWORK_,item=GRAPH);
```

The graph identifier and several other pieces of summary information about the in-memory graph are also contained in the output data table mycas.OutGraphList, as shown in Figure 3.30.

**Figure 3.30** Summary Information about the In-Memory Graphs

| graph | createTime | loaded | direction | nodes | links | multiLinks | selfLinks | standardizedLabels |
|---|---|---|---|---|---|---|---|---|
| 0 | 23OCT2019:12:06:23 | 1 | Undirected | 10 | 11 | 1 | 0 | 0 |

Next, the following statements use the in-memory graph that was loaded in the previous PROC OPTNETWORK call to find the route that yields the shortest path between **source="614CapitalBoulevard"** and **sink="SASCampusDrive"**:

```
proc optnetwork
   logLevel     = aggressive
   graph        = &_GRAPH_;
   shortestPath
```

```
        outPaths = mycas.ShortPath
        source   = "614CapitalBlvd"
        sink     = "SASCampusDrive";
run;
```

These statements point to the in-memory graph that is referenced by identifier _GRAPH_=0, build any additional data structures needed (since this is the first call to the shortest path algorithm), and then calculate the shortest path. The progress of the procedure is shown in Figure 3.31.

**Figure 3.31** PROC OPTNETWORK Log: Shortest Path Using an In-Memory Graph

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Building the input 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 11.
NOTE: Processing shortest paths problem using 128 threads across 4 machines.
NOTE: Processing the shortest paths problem between 1 source nodes and 1 sink nodes.
                                                   Real
      Algorithm              Sources   Complete     Time
      shortestPath                 1       100%     0.01
NOTE: Processing the shortest paths problem used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.04647 seconds.
NOTE: The data set MYCAS.SHORTPATH has 5 observations and 6 variables.
```

Figure 3.32 displays the output data table mycas.ShortPath, which shows the best route to take to minimize travel time between 614 Capital Boulevard and SAS Campus Drive.

**Figure 3.32** Shortest Path for Road Network between 614 Capital Boulevard and SAS Campus Drive

| 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** |

Next, to calculate the shortest path between **source="US70W/US440W"** and **sink="SASCampusDrive"**, use the following statements:

```
proc optnetwork
   logLevel    = aggressive
   graph       = &_GRAPH_;
   shortestPath
      outPaths = mycas.ShortPath
      source   = "US70W/US440W"
      sink     = "SASCampusDrive";
```

```
run;
```

These statements again point to the same in-memory graph (with no additional setup work required) and calculate the shortest path. The progress of the procedure is shown in Figure 3.33.

**Figure 3.33** PROC OPTNETWORK Log: Shortest Path Using an In-Memory Graph

```
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
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 shortest paths problem using 128 threads across 4 machines.
NOTE: Processing the shortest paths problem between 1 source nodes and 1 sink nodes.
                                                    Real
      Algorithm              Sources   Complete     Time
      shortestPath                 0         0%     0.00
NOTE: Processing the shortest paths problem used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.053983 seconds.
NOTE: The data set MYCAS.SHORTPATH has 4 observations and 6 variables.
```

Figure 3.34 displays the output data table mycas.ShortPath, which shows the best route to take to minimize travel time between US70W/US440W and SAS Campus Drive.

**Figure 3.34** Shortest Path for Road Network between US70W/US440W and SAS Campus Drive

| order | start_inter | end_inter | time_to_travel |
|---|---|---|---|
| 1 | US70W/US440W | US440W/RaleighExpy | 2.70000 |
| 2 | US440W/RaleighExpy | RaleighExpy/US40W | 3.00000 |
| 3 | RaleighExpy/US40W | US40W/HarrisonAve | 1.41818 |
| 4 | US40W/HarrisonAve | SASCampusDrive | 1.20000 |
| | | | **8.31818** |

The following statements load an additional graph, from the table mycas.LinkSetInRoadNC5pm:

```
proc optnetwork
   logLevel        = aggressive
   links           = mycas.LinkSetInRoadNC5pm;
   linksVar
      from         = start_inter
      to           = end_inter
      weight       = time_to_travel;
   loadGraph
      outGraphList = mycas.OutGraphList;
run;
```

The summary information about the in-memory graphs is contained in the output data table mycas.OutGraphList, as shown in Figure 3.35.

**Figure 3.35** Summary Information about the In-memory Graphs

| graph | createTime | loaded | direction | nodes | links | multiLinks | selfLinks | standardizedLabels |
|---|---|---|---|---|---|---|---|---|
| 0 | 23OCT2019:12:06:23 | 1 | Undirected | 10 | 11 | 1 | 0 | 0 |
| 1 | 23OCT2019:12:06:34 | 1 | Undirected | 10 | 11 | 1 | 0 | 0 |

Now, you can directly execute any supported algorithm for either graph (**GRAPH=0** or **GRAPH=1**), without reading in the links table again.

Finally, to delete the first in-memory graph (**GRAPH=0**), you can use the following statements:

```
proc optnetwork
   graph           = &_GRAPH_;
   unloadGraph
      outGraphList = mycas.OutGraphList;
run;
```

The output data table mycas.OutGraphList, shown in Figure 3.36, now indicates that the in-memory graph that has identifier 0 is no longer loaded.

**Figure 3.36** Summary Information about the In-memory Graphs

| graph | createTime | loaded | direction | nodes | links | multiLinks | selfLinks | standardizedLabels |
|---|---|---|---|---|---|---|---|---|
| 0 | 23OCT2019:12:06:23 | 0 | | . | . | . | . | . |
| 1 | 23OCT2019:12:06:34 | 1 | Undirected | 10 | 11 | 1 | 0 | 0 |

## 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 |
| CLIQUE | $\sqrt{M}$ | | | | $\sqrt{M}$ | | |
| CYCLE | $\sqrt{M}$ | | | | $\sqrt{M}$ | | |
| LINEARASSIGNMENT | $\sqrt{M}$ | | | | | | |
| MINCOSTFLOW | 1e15 | | 1e15 | 1e15 | | 1e15 | 1e15 |
| MINCUT | | | | | | | |
| SINK=, SOURCE= | 1e15 | | | | | | |
| otherwise | $\sqrt{M}$ | | | | | | |
| MINSPANTREE | $\sqrt{M}$ | | | | | | |
| PATH | $\sqrt{M}$ | $\sqrt{M}$ | | | $\sqrt{M}$ | | |

**Table 3.6** *continued*

| | Graph Links | | | | Graph Nodes | | |
|---|---|---|---|---|---|---|---|
| **Statement** | weight | auxweight | lower | upper | weight | lower | upper |
| 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 89.

### 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.

## Common Notation and Assumptions

This section briefly introduces some common notation and assumptions that are used throughout the chapter. Let $G = (N, E)$ define a graph (or multigraph) that contains a set $N$ of nodes and a set $E$ of links. Table 3.7 provides a list of commonly used notation.

**Table 3.7** Common Notation

| Notation | Description |
|---|---|
| $\delta_i^{\text{out}}$ | The set of outgoing links that are connected from node $i$ |
| $\delta_i^{\text{in}}$ | The set of incoming links that are connected to node $i$ |
| $\text{from}(e)$ | The *from* node of link $e$ |
| $\text{to}(e)$ | The *to* node of link $e$ |
| $\hat{N}_i$ | The set of neighbors excluding node $i$ itself (that is, unique nodes that are connected by links, excluding self-links, incident with node $i$) |
| $\hat{N}_i^{\text{out}}$ | The set of out-neighbors excluding node $i$ itself (that is, unique nodes that are connected by outgoing links, excluding self-links, from node $i$) |
| $\hat{N}_i^{\text{in}}$ | The set of in-neighbors excluding node $i$ itself (that is, unique nodes that are connected by incoming links, excluding self-links, to node $i$) |

A *complete graph* on node set $N$ is a simple graph (with no self-links) in which every pair of nodes in $N$ is connected by a link. The number of links in a complete graph on node set $N$ is

$$K(N) = \frac{|N|^2 - |N|}{2}$$

when DIRECTION=UNDIRECTED, or

$$K(N) = |N|^2 - |N|$$

when DIRECTION=DIRECTED.

## Biconnected Components and Articulation Points

A *biconnected component* of a graph $G = (N, E)$ 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 $O(|N| + |E|)$ and therefore should scale to very large graphs.

### Output Data Tables

Depending on the specified options, the biconnected components algorithm produces an additional output data table as described in the following section.

#### *OUT= Data Table*

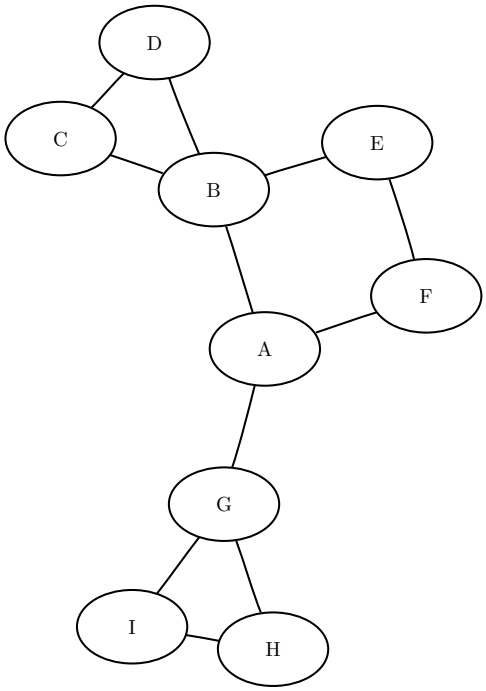
The OUT= data table describes the number of links in each biconnected component. This data table contains the following columns:

- biconcomp: biconnected component identifier
- links: number of links that are contained in the biconnected component

### Biconnected Components of an Undirected Graph

This section illustrates the use of the biconnected components algorithm on the undirected graph $G$ that is shown in Figure 3.37.

**Figure 3.37** 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, mycas.NodeSetOut, and mycas.BiConCompOut:

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

The output data table mycas.LinkSetOut contains the biconnected components of the input graph, as shown in Figure 3.38.

**Figure 3.38** Biconnected Components of an Undirected Graph

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

The output data table mycas.NodeSetOut contains the articulation points of the input graph, as shown in Figure 3.39.

**Figure 3.39** Articulation Points of an Undirected Graph

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

The output data table mycas.BiConCompOut contains the number of links in each biconnected component of the input graph, as shown in Figure 3.40.

**Figure 3.40** Summary for the Biconnected Components of an Undirected Graph

| biconcomp | links |
|---|---|
| 1 | 4 |
| 2 | 1 |
| 3 | 3 |
| 4 | 3 |

The biconnected components are shown graphically in Figure 3.41 and Figure 3.42.

**Figure 3.41** Biconnected Components $C^1$ and $C^2$

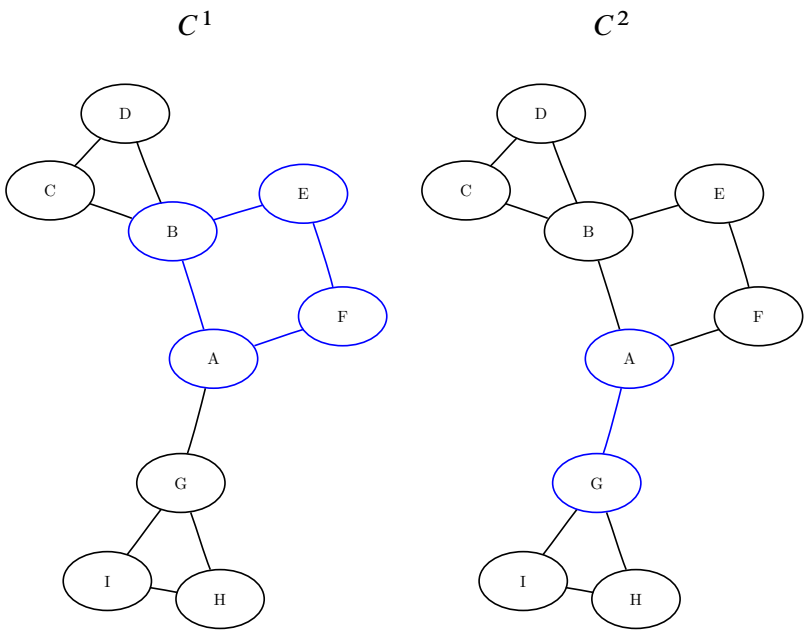
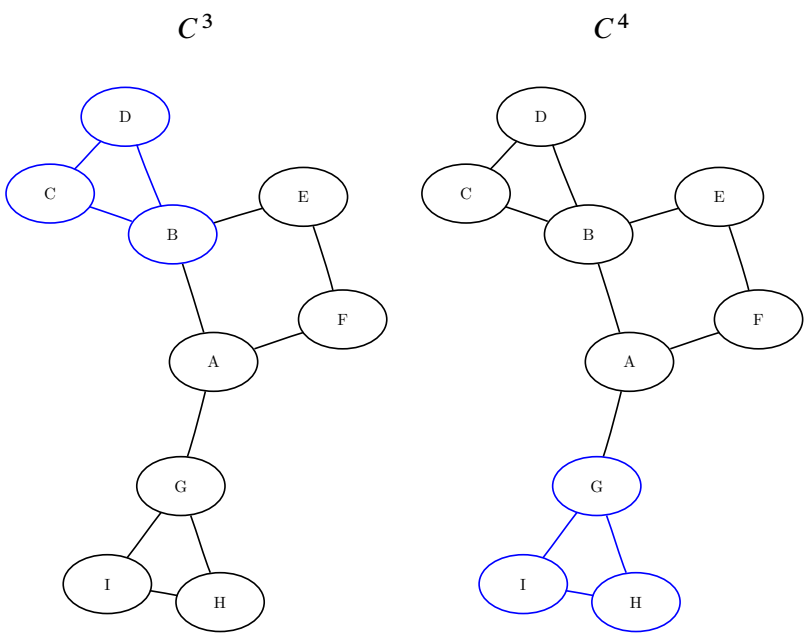



**Figure 3.42** Biconnected Components $C^3$ and $C^4$




For a more detailed example, see "Example 3.1: Articulation Points in a Terrorist Network" on page 142.

## Clique Enumeration

A *clique* of a graph $G = (N, E)$ 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 25. The clique algorithm works only with undirected simple graphs (with no self-links).

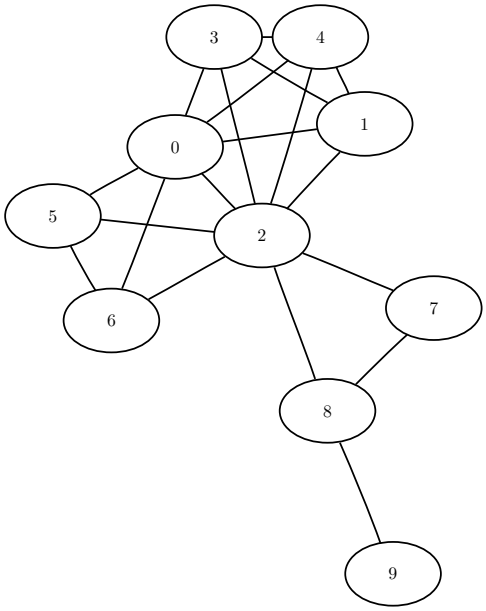
The results of the clique algorithm are written to the output data table that you specify in the OUT= option in the CLIQUE statement. Each node of each clique is listed in the output data table along with the variable clique to identify the clique to which it belongs. The clique identifiers are numbered 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 an Undirected Graph

This section illustrates the use of the clique algorithm on the undirected graph $G$ shown in Figure 3.43.

**Figure 3.43** 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 FEDSQL procedure as a convenient way to create a data table (mycas.CliqueSizes) of clique sizes:

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

proc fedsql sessref=mysess;
   create table CliqueSizes as
   select clique, count(*) as size
   from Cliques
   group by clique;
quit;
```

The output data table mycas.Cliques now contains the maximal cliques of the input graph, as shown in Figure 3.44.

**Figure 3.44** Maximal Cliques of an 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 3.45.

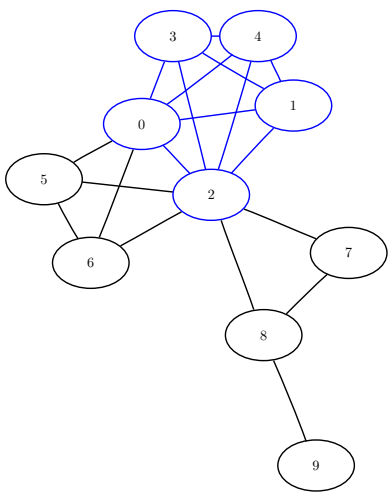
**Figure 3.45** Sizes of Maximal Cliques of an Undirected Graph

| clique | SIZE |
|---|---|
| 1 | 5 |
| 2 | 4 |
| 3 | 3 |
| 4 | 2 |

The maximal cliques are shown graphically in Figure 3.46 and Figure 3.47.

**Figure 3.46** Maximal Cliques $C^1$ and $C^2$

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




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

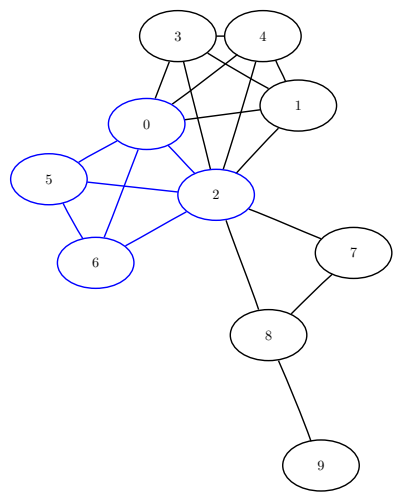
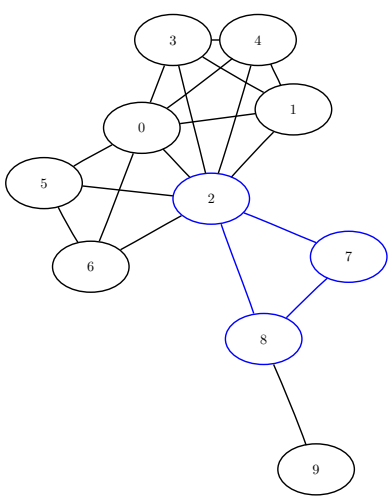
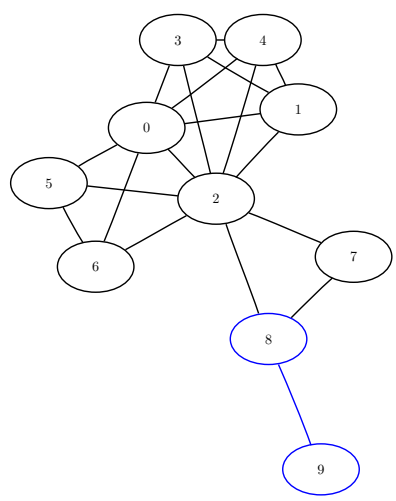



**Figure 3.47** Maximal Cliques $C^3$ and $C^4$

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




$C^4 = \{8, 9\}$



## 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 $G = (N, E)$, each algorithm runs in time $O(|N| + |E|)$ 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.

### Output Data Tables

Depending on the options selected, the connected components algorithm produces an additional output data table as described in the following section.

#### *OUT= Data Table*

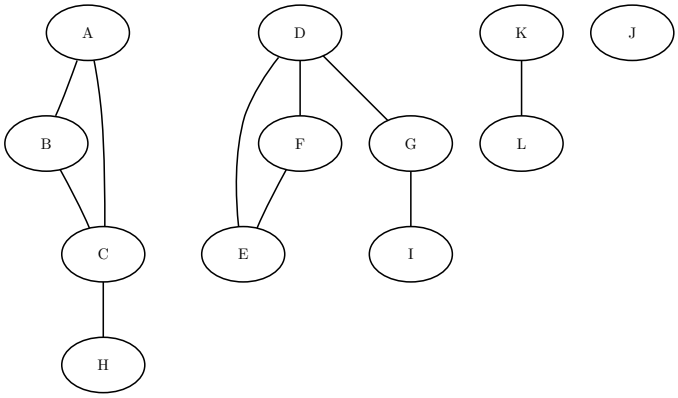
The OUT= data table describes the number of links in each connected component. This data table contains the following columns:

- concomp: connected component identifier
- nodes: number of nodes contained in the connected component

### Connected Components of an Undirected Graph

This section illustrates the use of the connected components algorithm on the undirected graph $G$ shown in Figure 3.48.

**Figure 3.48** 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 3.49.

**Figure 3.49** Connected Components of an 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 3.48, 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 3.50.

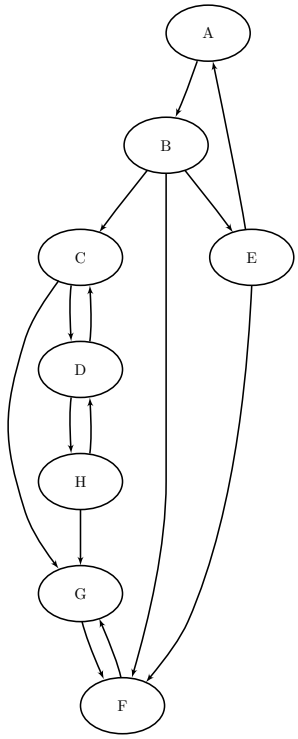
**Figure 3.50** Connected Components of an Undirected Graph

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

## Connected Components of a Directed Graph

This section illustrates the use of the connected components algorithm on the directed graph *G* shown in Figure 3.51.

**Figure 3.51** 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 tables mycas.NodeSetOut, mycas.LinkSetOut, and mycas.ConCompOut:

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

The output data table mycas.NodeSetOut, shown in Figure 3.52, now contains the node mappings for the connected components of the input graph.

**Figure 3.52** Node Mappings for the Connected Components of a Directed Graph

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

The output data table mycas.LinkSetOut, shown in Figure 3.53, now contains the link mappings for the connected components of the input graph.

**Figure 3.53** Links Mappings for the Connected Components of a Directed Graph

| from | to | concomp |
|---|---|---|
| C | G | . |
| H | G | . |
| B | C | . |
| E | F | . |
| B | F | . |
| A | B | 1 |
| E | A | 1 |
| B | E | 1 |
| C | D | 2 |
| H | D | 2 |
| D | C | 2 |
| D | H | 2 |
| F | G | 3 |
| G | F | 3 |

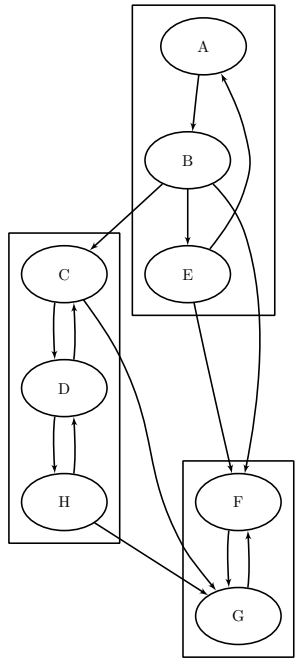
The output data table mycas.ConCompOut, shown in Figure 3.54, now contains the number of nodes in each connected component of the input graph.

**Figure 3.54** Summary for the Connected Components of a Directed Graph

| concomp | nodes |
|---|---|
| 1 | 3 |
| 2 | 3 |
| 3 | 2 |

The connected components are represented graphically in Figure 3.55.

**Figure 3.55** 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 27. To find the cycles and report them in an output data table, use either the OUTCYCLESNODES= option or the OUTCYCLESLINKS= option. To simply count the cycles, omit the output data table options.

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

By default, PROC OPTNETWORK uses depth-first search to quickly identify whether a cycle exists. This algorithm runs in time $O(|N| + |E|)$. This algorithm should scale to very large graphs. If filters are used or if the specified maximum number of cycles (MAXCYCLES=) is greater than 1, then enumerative algorithms are used.

The default algorithm that PROC OPTNETWORK uses to enumerate cycles when the value of the MAXLENGTH= option is greater than 20 (ALGORITHM=BACKTRACK) is a variant of the algorithm in Johnson (1975). This algorithm runs in time $O((|N| + |E|)(c + 1))$, 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 (ALGORITHM=BUILD) is described in Liu and

Wang (2006). 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 output data tables can become very large. It might be beneficial to check the number of cycles before you try to create an output data table. For more information about these options, see the section "CYCLE Statement" on page 27.

## Output Data Tables

Depending on the specified options, the cycle algorithm produces additional output data tables as described in the following section.

### *OUTCYCLESNODES= Data Table*

The OUTCYCLESNODES= data table describes the enumerated cycles as a sequence of nodes. This data table contains the following columns:

- cycle: the cycle identifier
- order: the order (sequence) of the node in the cycle
- node: the node label

### *OUTCYCLESLINKS= Data Table*

The OUTCYCLESLINKS= data table describes the enumerated cycles as a sequence of links. This data table contains the following columns:

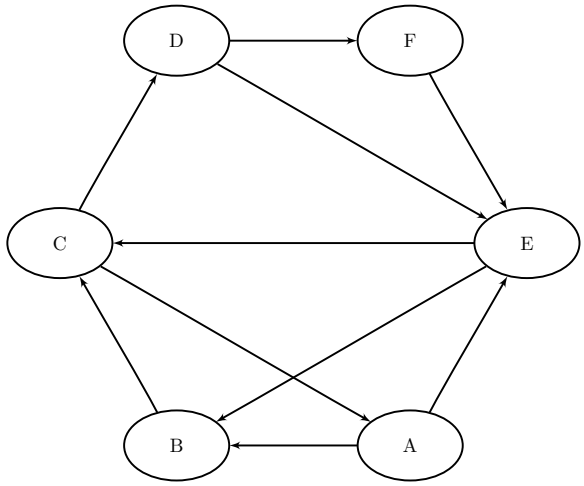
- cycle: the cycle identifier
- order: the order (sequence) of the link in the cycle
- from: the *from* node label
- to: the *to* node label

The cycle identifiers are numbered sequentially, starting from the value of the INDEXOFFSET= option in the PROC OPTNETWORK statement.

## Cycle Enumeration of a Directed Graph

This section provides a simple example of using the cycle enumeration algorithm on the directed graph $G$ shown in Figure 3.56. 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 154.

**Figure 3.56** 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 3.57.

**Figure 3.57** PROC OPTNETWORK Log: Count the Number of Cycles in a 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 enumeration using 1 threads across 1 machines.
NOTE: The algorithm found 7 cycles.
NOTE: Processing cycle enumeration used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.032114 seconds.
STATUS=OK  PROBLEM_TYPE=CYCLE  SOLUTION_STATUS=OK  NUM_CYCLES=7  CPU_TIME=0.06  REAL_TIME=0.03
```

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

```
proc optnetwork
   direction         = directed
   links             = mycas.LinkSetIn;
   cycle
      outCyclesNodes = mycas.CyclesNodes
      outCyclesLinks = mycas.CyclesLinks;
run;
```

The output data table mycas.CyclesNodes now contains the nodes of the first cycle that is found in the input graph, as shown in Figure 3.58.

**Figure 3.58** Nodes of the First Cycle Found in a Directed Graph

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

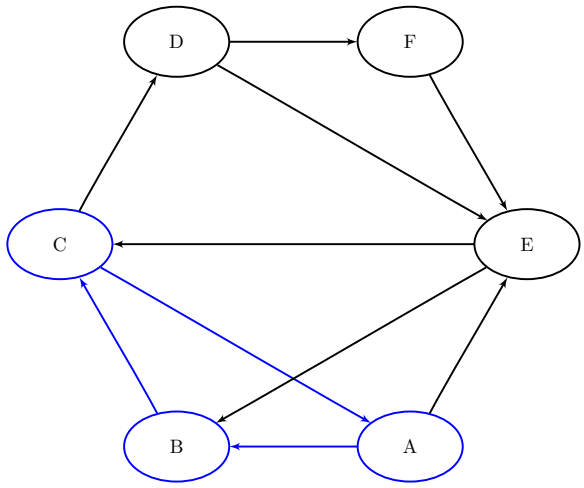
The output data table mycas.CyclesLinks now contains the links of the first cycle that is found in the input graph, as shown in Figure 3.59.

**Figure 3.59** Links of the First Cycle Found in a Directed Graph

| cycle | order | from | to |
|---|---|---|---|
| 1 | 1 | A | B |
| 1 | 2 | B | C |
| 1 | 3 | C | A |

The first cycle that is found in the input graph is shown graphically in Figure 3.60.

**Figure 3.60** $A \rightarrow B \rightarrow C \rightarrow A$




The following statements return all the cycles in the graph:

```
proc optnetwork
   direction         = directed
   links             = mycas.LinkSetIn;
   cycle
      maxCycles      = all
      outCyclesNodes = mycas.CyclesNodes;
run;
```

The output data table mycas.CyclesNodes now contains the nodes of all the cycles in the input graph, as shown in Figure 3.61.

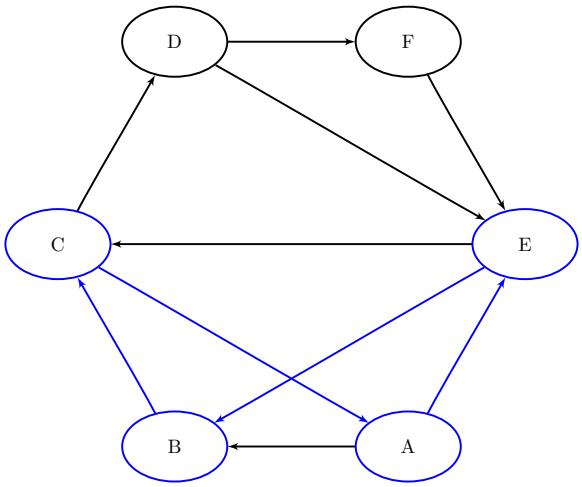
**Figure 3.61** Nodes of All Cycles in a Directed Graph

| 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 |
| 4 | 5 | B |
| 5 | 1 | B |
| 5 | 2 | C |
| 5 | 3 | D |
| 5 | 4 | F |
| 5 | 5 | E |
| 5 | 6 | B |
| 6 | 1 | C |
| 6 | 2 | D |
| 6 | 3 | E |
| 6 | 4 | C |
| 7 | 1 | C |
| 7 | 2 | D |
| 7 | 3 | F |
| 7 | 4 | E |
| 7 | 5 | C |

The six additional cycles are shown graphically in Figure 3.62 through Figure 3.64.

**Figure 3.62** Cycles

$A \to E \to B \to C \to A$




$A \to E \to C \to A$

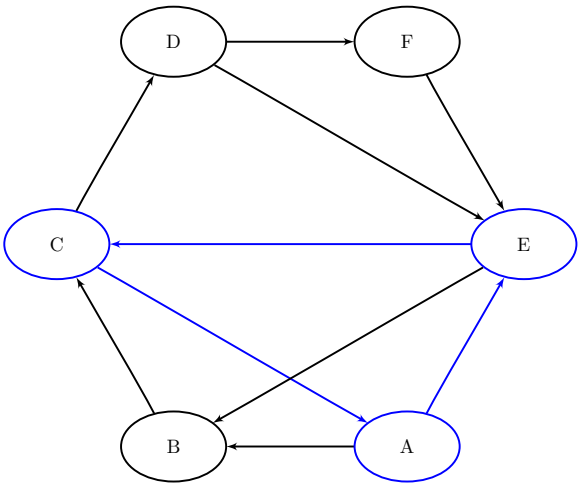
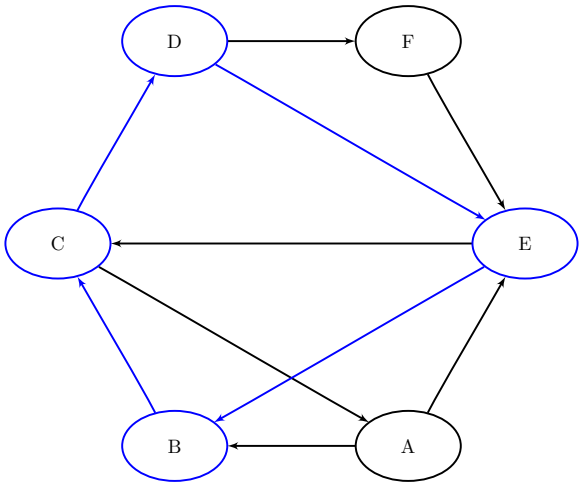



**Figure 3.63** Cycles

$B \to C \to D \to E \to B$




$B \to C \to D \to F \to E \to B$

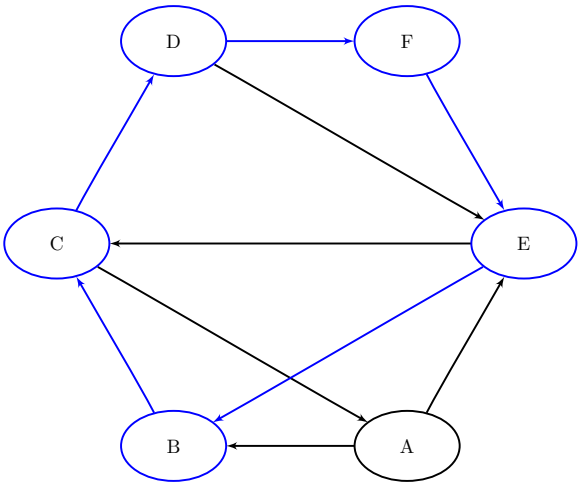
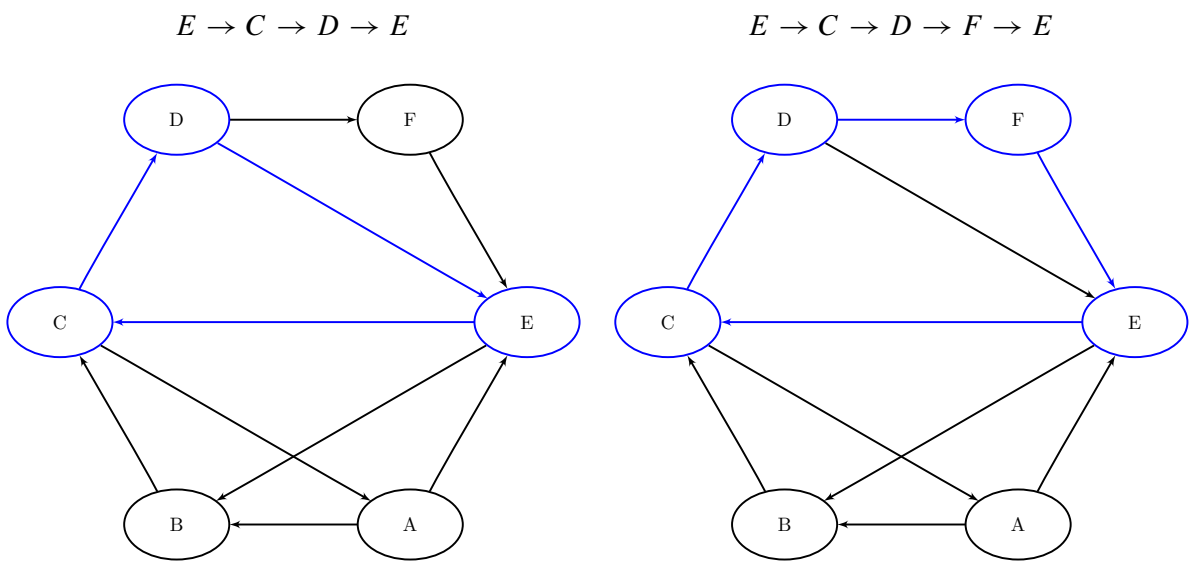


**Figure 3.64** Cycles




## 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 $E$ 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:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{e \in E} c_e x_e \\
\text{subject to} \quad & \sum_{e \in \delta_i^{\text{out}}} x_e \leq 1 \qquad i \in S \\
& \sum_{e \in \delta_j^{\text{in}}} x_e = 1 \qquad j \in T \\
& x_e \in \{0, 1\} \qquad e \in E
\end{aligned}
$$

where $\delta_i^{\text{out}}$ represents the set of outgoing links that are connected from node $i$ and $\delta_j^{\text{in}}$ represents the set of incoming links that are connected to node $j$.

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:

$$\begin{aligned}
\text{minimize} \quad & \sum_{e \in E} c_e x_e \\
\text{subject to} \quad & \sum_{e \in \delta_i^{\text{out}}} x_e = 1 \quad i \in S \\
& \sum_{e \in \delta_j^{\text{in}}} x_e \leq 1 \quad j \in T \\
& x_e \in \{0, 1\} \quad e \in E
\end{aligned}$$

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 algorithm that 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 150.

## Minimum-Cost Network Flow

The *minimum-cost network flow* (MCF) problem is a fundamental problem in network analysis that involves sending flow through a network at minimal cost. Let $G = (N, E)$ be a directed graph. For each link $e \in E$, associate a cost per unit of flow, designated as $c_e$. The demand (or supply) at each node $i \in N$ is designated as $b_i$, where $b_i \geq 0$ denotes a supply node and $b_i < 0$ denotes a demand node. These values must be within $[b_i^l, b_i^u]$. Define decision variables $x_e$ that denote the amount of flow sent across link $e$. The amount of flow that can be sent across each link is bounded to be within $[l_e, u_e]$. The problem can be modeled as a linear programming problem as

$$\begin{aligned}
\text{minimize} \quad & \sum_{e \in E} c_e x_e \\
\text{subject to} \quad & b_i^l \leq \sum_{e \in \delta_i^{\text{out}}} x_e - \sum_{e \in \delta_i^{\text{in}}} x_e \leq b_i^u \quad i \in N \\
& l_e \leq x_e \leq u_e \quad e \in E
\end{aligned}$$

where $\delta_i^{\text{out}}$ represents the set of outgoing links that are connected from node $i$ and $\delta_i^{\text{in}}$ represents the set of incoming links that are connected to node $i$. 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 supply and demand values 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 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 42. The links data table, which you specify in the LINKS= option in the PROC OPTNETWORK statement, can contain the following columns:

- weight, which defines the link cost $c_e$
- lower, which defines the link lower bound $l_e$. The default is 0.
- upper, which defines the link upper bound $u_e$. The default is $\infty$.

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 $\infty$.

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 $\infty$, 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:

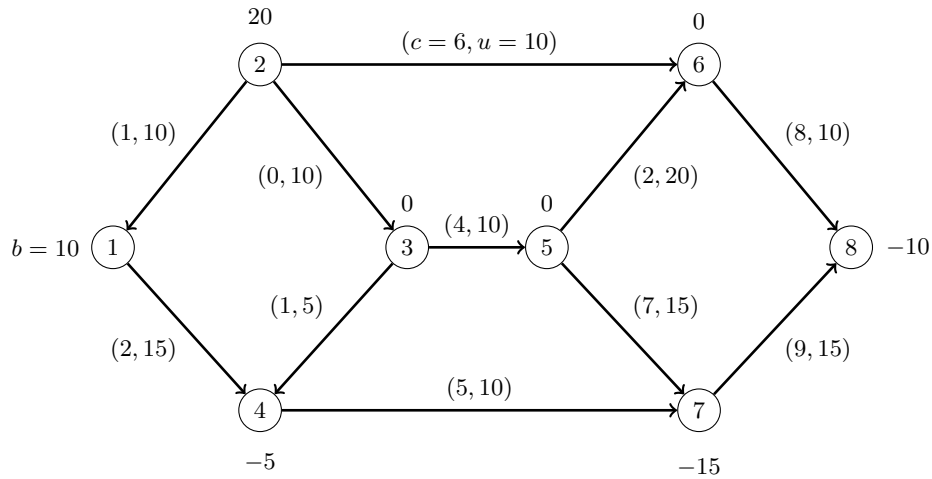
- The flow on a link must be bounded from below (that is, $l_e = -\infty$ is not allowed).
- Flow balance constraints cannot be *free* (that is, $b_i^l = -\infty$ and $b_i^u = \infty$ is not allowed).

The resulting optimal flow (mcf_flow) through the network and reduced cost (mcf_rc) of each link are written to the links output data table, which you specify in the OUTLINKS= option in the PROC OPTNETWORK statement. The optimal dual value (mcf_dual) for each node is written to the nodes output data table, which you specify in the OUTNODES= option in the PROC OPTNETWORK statement.

## Minimum-Cost Network Flow for a 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 3.65, which appears in Ahuja, Magnanti, and Orlin (1993).

**Figure 3.65** Minimum-Cost Network Flow Problem: Data




The directed graph $G$ can be represented by the links data table, mycas.LinkSetIn, and nodes data table, mycas.NodeSetIn, that are created by the following DATA steps:

```
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
      outNodes    = mycas.NodeSetOut;
      minCostFlow
         logFreq  = 1;
   run;
   %put &_OROPTNETWORK_;
```

The progress of the procedure is shown in Figure 3.66.

**Figure 3.66** 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 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 using 1 threads across 1 machines.
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   7.500000E+01   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 0.049403 seconds.
NOTE: The data set MYCAS.LINKSETOUT has 11 observations and 6 variables.
NOTE: The data set MYCAS.NODESETOUT has 8 observations and 3 variables.
STATUS=OK  PROBLEM_TYPE=MINCOSTFLOW  SOLUTION_STATUS=OPTIMAL  OBJECTIVE=270  CPU_TIME=0.12
REAL_TIME=0.05
```

The optimal flow and reduced costs are displayed in Figure 3.67.

**Figure 3.67** Minimum-Cost Network Flow Problem: Optimal Flows and Reduced Costs

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

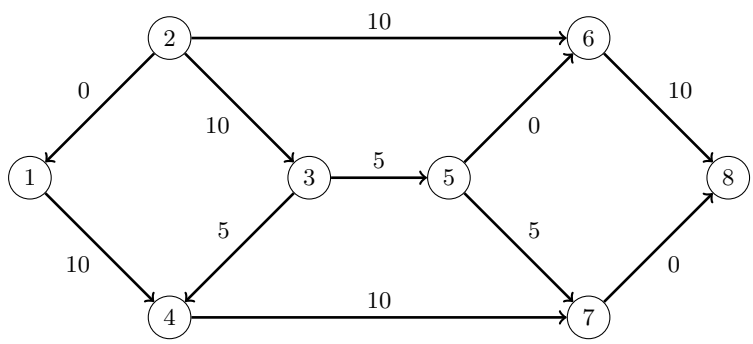
The optimal dual values are displayed in Figure 3.68.

**Figure 3.68** Minimum-Cost Network Flow Problem: Optimal Dual Values

| Obs | node | lower | mcf_dual |
|---|---|---|---|
| **1** | 1 | 10 | 14 |
| **2** | 2 | 20 | 14 |
| **3** | 3 | . | 14 |
| **4** | 4 | -5 | 12 |
| **5** | 5 | . | 10 |
| **6** | 6 | . | 8 |
| **7** | 7 | -15 | 3 |
| **8** | 8 | -10 | 0 |

The optimal flows are represented graphically in Figure 3.69.

**Figure 3.69** Minimum-Cost Network Flow Problem: Optimal Flows



### Minimum-Cost Network Flow with Flexible Supply and Demand

Using the same directed graph shown in Figure 3.65, 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 nodes data table that is created by the following DATA step:

```
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
   outNodes    = mycas.NodeSetOut;
   minCostFlow
      logFreq  = 1;
run;
%put &_OROPTNETWORK_;
```

The progress of the procedure is shown in Figure 3.70.

**Figure 3.70** 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 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 using 1 threads across 1 machines.
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   7.500000E+01   1.500000E+01   8.600000E+01     0.00
              5   7.500000E+01   1.500000E+01   1.690000E+02     0.00
              6   1.300000E+02   1.000000E+01   1.600000E+02     0.00
              7   1.300000E+02   1.000000E+01   1.580000E+02     0.00
              8   1.300000E+02   1.000000E+01   1.580000E+02     0.00
              9   2.400000E+02   5.000000E+00   8.400000E+01     0.00
             10   2.400000E+02   5.000000E+00   8.400000E+01     0.00
             11   2.400000E+02   5.000000E+00   1.660000E+02     0.00
             12   2.540000E+02   4.000000E+00   9.500000E+01     0.00
             13   2.540000E+02   4.000000E+00   9.500000E+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 0.046447 seconds.
NOTE: The data set MYCAS.LINKSETOUT has 11 observations and 6 variables.
NOTE: The data set MYCAS.NODESETOUT has 8 observations and 4 variables.
STATUS=OK  PROBLEM_TYPE=MINCOSTFLOW  SOLUTION_STATUS=OPTIMAL  OBJECTIVE=226  CPU_TIME=0.12
REAL_TIME=0.05
```

The optimal flow and reduced costs are displayed in Figure 3.71.

**Figure 3.71** Minimum-Cost Network Flow Problem: Optimal Flows and Reduced Costs

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

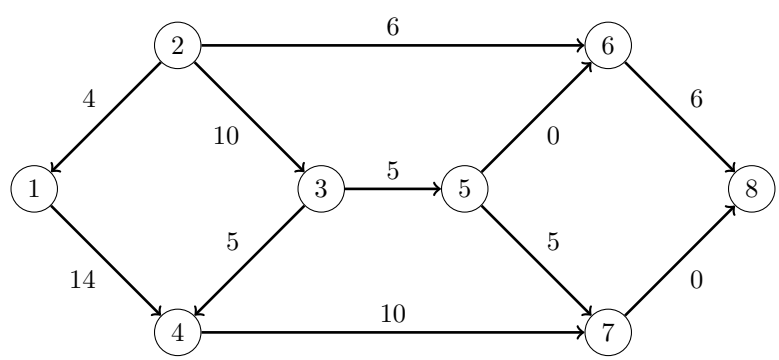
The optimal dual values are displayed in Figure 3.72.

**Figure 3.72** Minimum-Cost Network Flow Problem: Optimal Dual Values

| Obs | node | lower | upper | mcf_dual |
|---|---|---|---|---|
| **1** | 1 | 10 | I | 2 |
| **2** | 2 | 20 | 20 | 3 |
| **3** | 3 | . | . | 3 |
| **4** | 4 | M | -5 | 0 |
| **5** | 5 | . | . | -1 |
| **6** | 6 | . | . | -3 |
| **7** | 7 | -15 | -15 | -8 |
| **8** | 8 | -10 | -6 | -11 |

The optimal flows are represented graphically in Figure 3.73.

**Figure 3.73** Minimum-Cost Network Flow Problem: Optimal Flows



## Minimum Cut

A *cut* is a partition of the nodes $N$ of a graph into two disjoint subsets $S$ and $T = N \setminus S$. A *cut set* is the set of links that join a node in $S$ to a node in $T$. A *minimum cut* of a 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. Unless you specify the SOURCE= and SINK= options, 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 in terms of 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 whether it belongs to $S$ or $T$, respectively. 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.

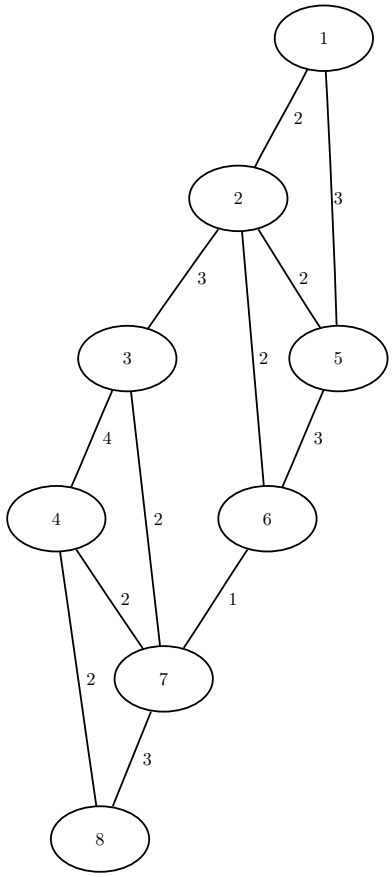
PROC OPTNETWORK uses the Stoer-Wagner algorithm (Stoer and Wagner 1997) to compute the minimum cuts. This algorithm runs in time $O(|N||E| + |N|^2 \log |N|)$.

When you specify the SOURCE=*s* and SINK=*t* options in the MINCUT statement, PROC OPTNETWORK solves the *minimum s-t cut* problem, which is to find a minimum cut such that $s \in S$ and $t \in T$. The cut set for this cut intersects every path from the source node $s$ to the sink node $t$. If you specify either the SOURCE= or the SINK= option, the other is also required. Only one cut is returned by the minimum *s-t* cut algorithm, so the MAXCUTS= option is ignored when the SOURCE= and SINK= options are specified.

### Minimum Cut for an Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 3.74.

**Figure 3.74** 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 3.75.

**Figure 3.75** 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 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 using 1 threads across 1 machines.
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 0.064852 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=0.06
```

The output data table mycas.Partitions now contains the partition of the nodes and is shown by cut in Figure 3.76.

**Figure 3.76** Minimum Cut Node Partition

**cut=1**

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

**cut=2**

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

**Figure 3.76** *continued*

cut=3

| node | partition |
|---|---|
| 1 | 1 |
| 7 | 1 |
| 6 | 1 |
| 5 | 1 |
| 4 | 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 3.77.

**Figure 3.77** 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** |

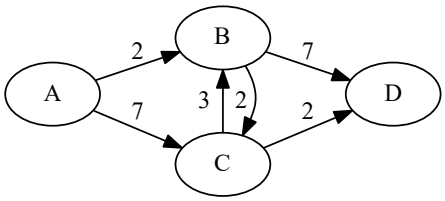
cut=3

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

## Minimum *s*-*t* Cut for a Directed Graph

As an example of the minimum *s*-*t* cut algorithm, consider the weighted directed graph in Figure 3.78.

**Figure 3.78** Directed Graph




The links data table can be represented as follows:

```
data mycas.LinkSetIn;
   input from $ to $ weight;
   datalines;
A B 2
A C 7
B C 2
B D 7
C B 3
C D 2
;
```

The following statements calculate a minimum cut that intersects all paths from A to D and output the results in the data tables mycas.CutSets and mycas.Partitions:

```
proc optnetwork
   logLevel        = moderate
   direction       = directed
   links           = mycas.LinkSetIn;
   minCut
      outCutSets    = mycas.CutSets
      outPartitions = mycas.Partitions
      source        = A
      sink          = D;
run;
%put &_OROPTNETWORK_;
```

The progress of the procedure is shown in Figure 3.79.

**Figure 3.79** 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 graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 4.
NOTE: The number of links in the input graph is 6.
NOTE: Processing the minimum-cut problem using 1 threads across 1 machines.
NOTE: The minimum-cut algorithm found 1 cuts.
NOTE: Objective = 7.
NOTE: Processing the minimum-cut problem used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.043671 seconds.
NOTE: The data set MYCAS.CUTSETS has 3 observations and 4 variables.
NOTE: The data set MYCAS.PARTITIONS has 4 observations and 3 variables.
STATUS=OK  PROBLEM_TYPE=MINCUT  SOLUTION_STATUS=OPTIMAL  OBJECTIVE=7  CPU_TIME=0.10
REAL_TIME=0.04
```

The output data table mycas.Partitions now contains the partition of the nodes and is shown in Figure 3.80.

**Figure 3.80** Minimum *s*-*t* Cut Node Partition

**cut=1**

| node | partition |
|---|---|
| B | 1 |
| C | 0 |
| D | 1 |
| A | 0 |

The output data table mycas.CutSets contains the set of links in the minimum cut. This data table is shown in Figure 3.81.

**Figure 3.81** Minimum Cut Sets

| cut | from | to | weight |
|---|---|---|---|
| 1 | A | B | 2 |
| 1 | C | B | 3 |
| 1 | C | D | 2 |

## 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 34. You can use this algorithm only on undirected graphs.

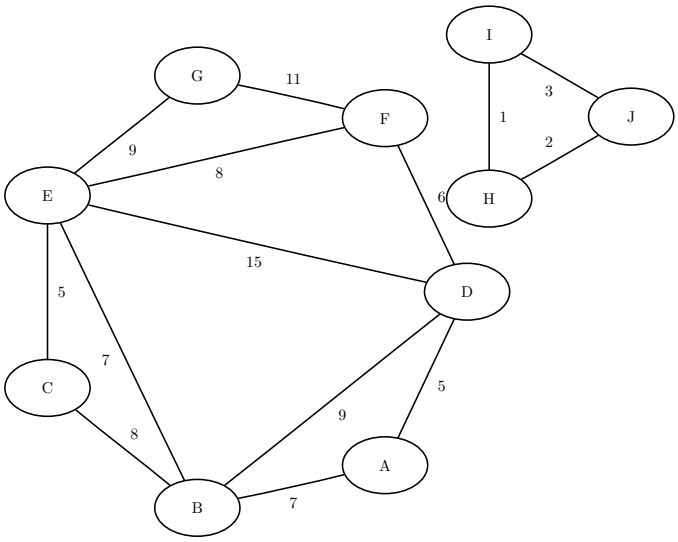
The resulting minimum spanning tree is contained in the output data table that you specify in the OUT= option in the MINSPANTREE statement.

PROC OPTNETWORK uses Kruskal's algorithm (Kruskal 1956) to compute the minimum spanning tree. This algorithm runs in time $O(|E| \log |N|)$ and therefore should scale to very large graphs.

### Minimum Spanning Tree for an Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 3.82.

**Figure 3.82** 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;
```

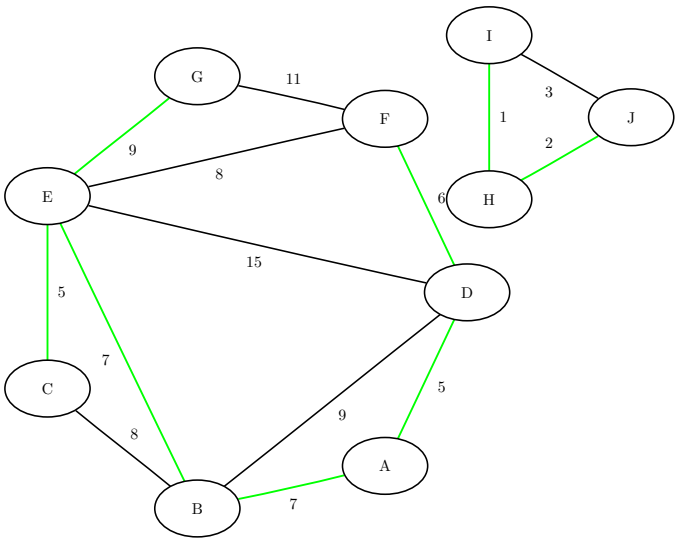
The output data table mycas.MinSpanForest now contains the links that belong to a minimum spanning forest, which is shown in Figure 3.83.

**Figure 3.83** Minimum Spanning Forest

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

The minimal cost links are shown in green in Figure 3.84.

**Figure 3.84** Minimum Spanning Forest




For a more detailed example, see "Example 3.4: Minimum Spanning Tree for Computer Network Design" on page 152.

## 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 is visited more than once. A path between two nodes, *i* and *j*, in a graph is a path that starts at *i* and ends at *j*. 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 47. 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 109.

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: Fundamentals*.

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.

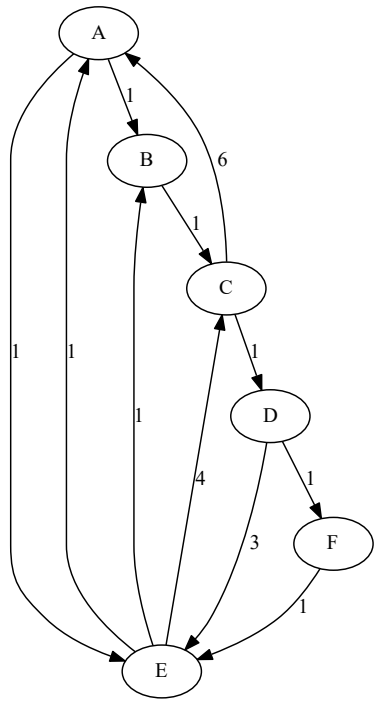
The OUTPATHSNODES= data table contains the following columns:

- 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 directed graph $G$ shown in Figure 3.85 to find all paths between one source-sink pair by using the SOURCE= and SINK= options.

**Figure 3.85** 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 3.86.

**Figure 3.86** Links for All (Short) Paths in a 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 3.87.

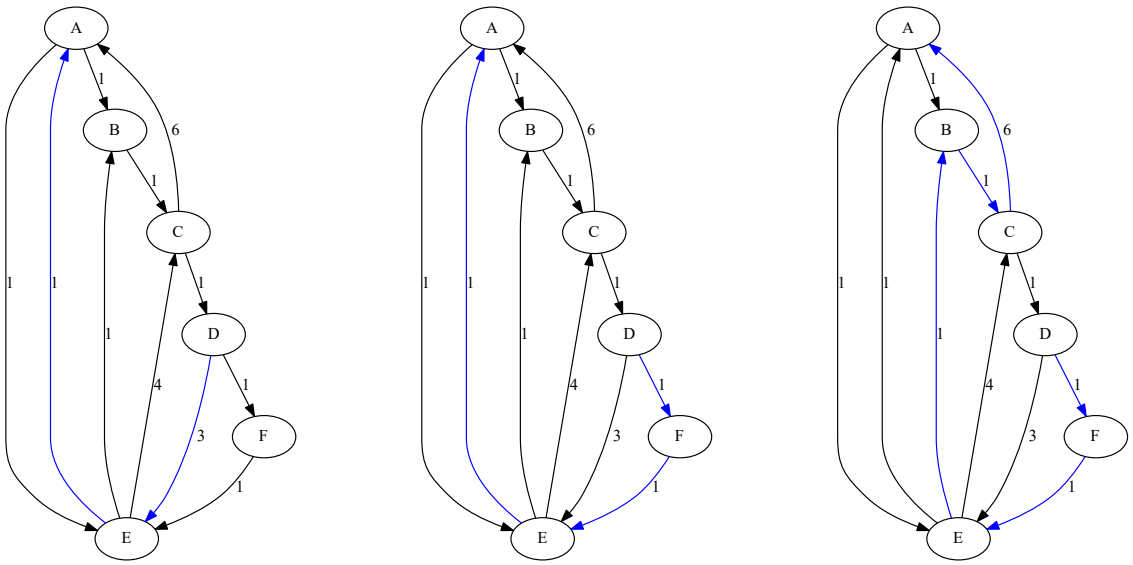
**Figure 3.87** Nodes for All (Short) Paths in a Directed Graph

| 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 3.88.

**Figure 3.88** 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, *i* and *j*, in a graph is a path that starts at *i* and ends at *j* 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 37.

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 47. 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 3.8.

**Table 3.8** Algorithms for Shortest Paths

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

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 three 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. The output data table that you specify in the OUTSUMMARY= option contains descriptive statistics of the finite shortest paths for each source.

### *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: Fundamentals*.

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 165.

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

If you use the AUXWEIGHT= option in the LINKSVAR statement, the following column also appears in the summary output table:

- *column*: the auxiliary weight of this link

### *OUTSUMMARY= Data Table*

The OUTSUMMARY= data table contains descriptive statistics for the finite shortest paths for each source. This data table contains the following columns:

- source: the source node label
- paths: the number of finite shortest paths from source to requested sinks
- path_weight_min: the minimum weight of shortest paths from source to requested sinks
- path_weight_max: the maximum weight of shortest paths from source to requested sinks
- path_weight_avg: the average weight of shortest paths from source to requested sinks
- path_weight_std: the standard deviation of weight of shortest paths from source to requested sinks
- path_weight_var: the variance of weight of shortest paths from source to requested sinks

If you use the AUXWEIGHT= option in the LINKSVAR statement, the following columns also appear in the summary output table:

- path_auxweight_min: the minimum auxiliary weight of shortest paths from source to requested sinks
- path_auxweight_max: the maximum auxiliary weight of shortest paths from source to requested sinks
- path_auxweight_avg: the average auxiliary weight of shortest paths from source to requested sinks
- path_auxweight_std: the standard deviation of auxiliary weight of shortest paths from source to requested sinks
- path_auxweight_var: the variance of auxiliary weight of shortest paths from source to requested sinks

### *OUTWEIGHTS= Data Table*

The OUTWEIGHTS= data table contains the weight (and 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 weight of the shortest path for this source-sink pair

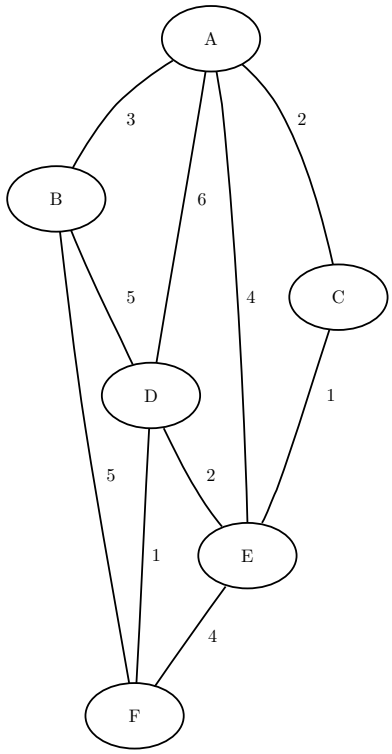
If you use the AUXWEIGHT= option in the LINKSVAR statement, the following column also appears in the summary output table:

- path_auxweight: the 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 undirected graph *G* shown in Figure 3.89.

**Figure 3.89** 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
      outSummary = mycas.ShortPathS
      outWeights = mycas.ShortPathW
      outPaths   = mycas.ShortPathP;
run;
```

The output data table mycas.ShortPathP contains the shortest paths, as shown in Figure 3.90.

**Figure 3.90** 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 |

| source | sink | order | from | to | weight |
|---|---|---|---|---|---|
| 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 3.91.

**Figure 3.91** All-Pairs Shortest Paths Weights

| source | sink | path_weight |
|---|---|---|
| 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 |
| B | A | 3 |
| B | C | 5 |
| B | D | 5 |
| B | E | 6 |
| B | F | 5 |

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

The output data table mycas.ShortPathS contains descriptive statistics of the finite shortest paths for each source, as shown in Figure 3.92.

**Figure 3.92** All-Pairs Shortest Paths Summary

| source | paths | path_weight_min | path_weight_max | path_weight_avg | path_weight_std | path_weight_var |
|---|---|---|---|---|---|---|
| A | 5 | 2 | 6 | 3.8 | 1.64317 | 2.7 |
| E | 5 | 1 | 6 | 3.0 | 1.87083 | 3.5 |
| B | 5 | 3 | 6 | 4.8 | 1.09545 | 1.2 |
| F | 5 | 1 | 6 | 3.8 | 1.92354 | 3.7 |
| C | 5 | 1 | 5 | 3.0 | 1.58114 | 2.5 |
| D | 5 | 1 | 5 | 3.2 | 1.78885 | 3.2 |

## 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 3.93.

**Figure 3.93** 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 114 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 3.94.

**Figure 3.94** Shortest Paths for a Subset of Source Pairs

| source | sink | order | from | to | weight |
|---|---|---|---|---|---|
| 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 |
| 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 |

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 3.95.

**Figure 3.95** Shortest Paths for a Subset of Sink Pairs

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

## 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;
```

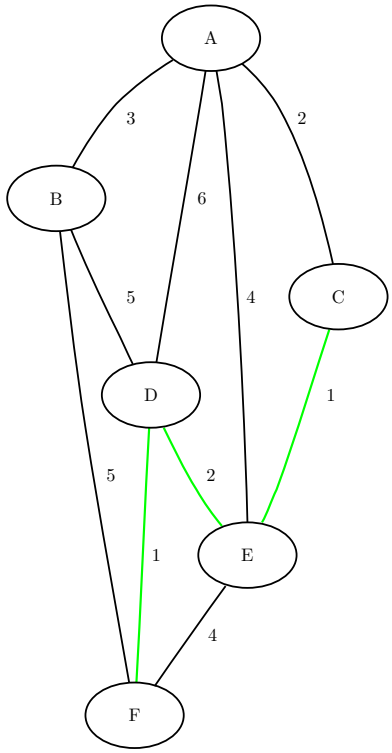
The output data table mycas.ShortPath contains this shortest path, as shown in Figure 3.96.

**Figure 3.96** 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 3.97.

**Figure 3.97** 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 3.98. 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 3.98** Shortest Paths Including Auxiliary Weights in Calculation

| 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 |
| 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 |

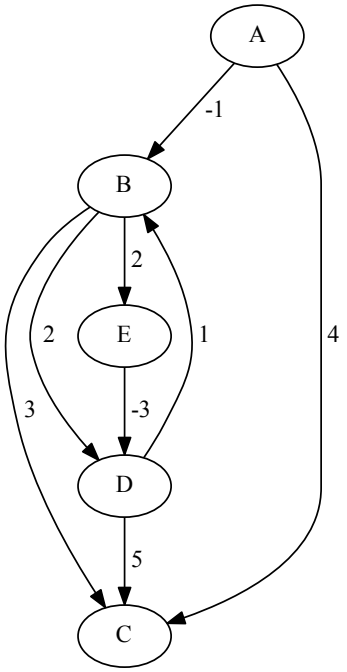
| source | sink | path_weight | path_auxweight |
|---|---|---|---|
| F | A | 6 | 4 |
| F | B | 5 | 1 |
| F | C | 4 | 3 |
| F | D | 1 | 1 |
| F | E | 3 | 2 |
| B | A | 3 | 1 |
| B | C | 5 | 2 |
| B | D | 5 | 1 |
| B | E | 6 | 3 |
| B | F | 5 | 1 |
| C | A | 2 | 1 |
| C | B | 5 | 2 |
| C | D | 3 | 2 |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

The section "Road Network Shortest Path" on page 12 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 directed graph $G$ with negative link weights, shown in Figure 3.99.

**Figure 3.99** 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 3.100.

**Figure 3.100** 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 3.101.

**Figure 3.101** 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 shortest paths problem using 128 threads across 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 0.02 (cpu: 0.00) seconds.
ERROR: The action stopped due to errors.
NOTE: The Cloud Analytic Services server processed the request in 0.609204 seconds.
NOTE: The SAS System stopped processing this step because of errors.
STATUS=ERROR  PROBLEM_TYPE=SHORTESTPATH  CPU_TIME=0.14  REAL_TIME=0.61
```

## 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 38.

### 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.

In an undirected graph, $\hat{N}_i$ represents the set of neighbors excluding node $i$ itself (that is, unique nodes that are connected by links, excluding self-links, incident with node $i$). In a directed graph, $\hat{N}_i^{\text{out}}$ represents the set of out-neighbors excluding node $i$ itself (that is, unique nodes that are connected by outgoing links,

excluding self-links, from node $i$), and $\hat{N}_i^{\text{in}}$ represents the set of in-neighbors excluding node $i$ itself (that is, unique nodes that are connected by incoming links, excluding self-links, to node $i$).

### 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 ($|E|$)
- 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 $\left(\frac{|E|}{K(N)}\right)$
- self_links_ignored: the number of self-links that are ignored
- dup_links_ignored: the number of links removed in multilink aggregation
- leaf_nodes: the number of leaf nodes
  - for an undirected graph: a node $i$ is a leaf node if $|\hat{N}_i| = 1$
  - for a directed graph: a node $i$ is a leaf node if $|\hat{N}_i^{\text{out}}| = 0$ and $|\hat{N}_i^{\text{in}}| > 0$
- singleton_nodes: the number of singleton nodes
  - for an undirected graph: a node $i$ is a singleton node if $|\hat{N}_i| = 0$
  - a directed graph: a node $i$ is a singleton node if $|\hat{N}_i^{\text{out}}| + |\hat{N}_i^{\text{in}}| = 0$

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 77 and "Biconnected Components and Articulation Points" on page 70, 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 in which one node $i \in C$ has $|\hat{N}_i| = |C| - 1$ and all other nodes $i \in C \setminus \{i\}$ have $|\hat{N}_i| = 1$)

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 in which one node $i \in C$ has $|\hat{N}_i^{\text{out}}| = |C| - 1$ and all other nodes $i \in C \setminus \{i\}$ have $|\hat{N}_i^{\text{in}}| = 1$)
- isolated_stars_in: the number of isolated inward stars (a weakly connected component, $C$, of size greater than 2 in which one node $i \in C$ has $|\hat{N}_i^{\text{in}}| = |C| - 1$ and all other nodes $i \in C \setminus \{i\}$ have $|\hat{N}_i^{\text{out}}| = 1$)

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 109. If you use the SHORTESTPATH= option, the following columns also appear in the summary output data table:

- diameter_wt: the longest weighted shortest path distance in the graph
- diameter_unwt: the longest unweighted shortest path distance in the graph
- avg_shortpath_wt: the average weighted shortest path distance in the graph
- avg_shortpath_unwt: the average unweighted shortest path distance 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 $O(|N| \times (|N| \log |N| + |E|))$; the approximate method runs in time $O(|E|\sqrt{|N|})$ 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 distance in the graph
- diameter_approx_unwt: the approximate longest unweighted shortest path distance in the graph

You can produce statistics about clustering coefficients by using the CLUSTERINGCOEFFICIENT option. The *triangle count* of an undirected graph is the number of distinct three-node sets in which each node is a neighbor of the other two. If you use the CLUSTERINGCOEFFICIENT option, the following column also appears in the summary output data table:

- triangles: the total triangle count of 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 *i* is the longest of all possible shortest path distances between *i* 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 (.)

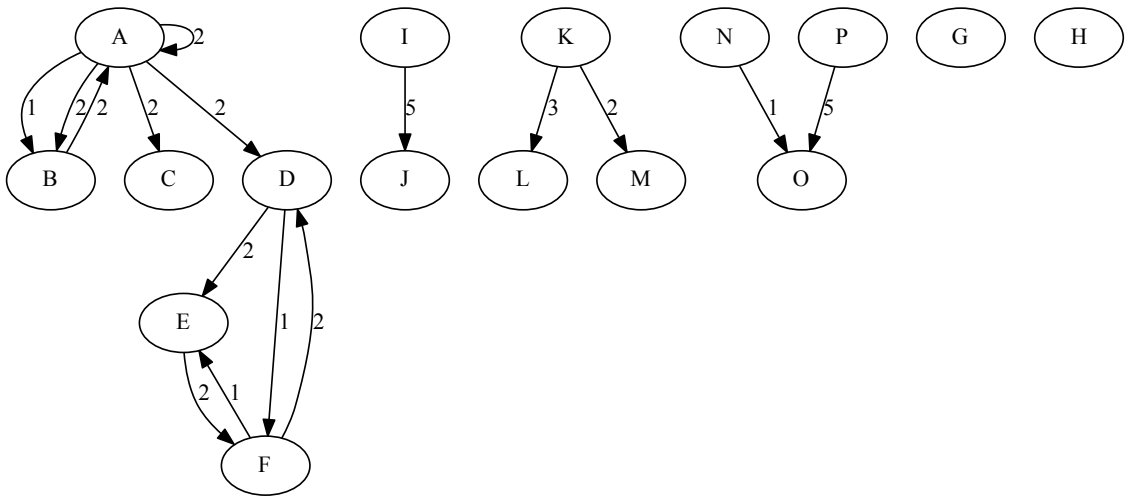
The following columns are appended for directed graphs:

- 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 Directed Graph

This section illustrates the calculation of summary statistics on the directed graph *G* shown in Figure 3.102.

**Figure 3.102** 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
;
```

### *Basic Summary Statistics*

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 3.103.

**Figure 3.103** Graph Summary Statistics of a Directed Graph

| nodes | links | avg_links_per_node | density | self_links_ignored | dup_links_ignored | leaf_nodes | singleton_nodes |
|---|---|---|---|---|---|---|---|
| 16 | 15 | 0.9375 | 0.0625 | 1 | 0 | 5 | 2 |

### *Basic Summary Statistics and Connected Components*

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 3.104.

**Figure 3.104** Graph Summary and Connectedness Statistics of a Directed Graph

| nodes | links | avg_links_per_node | density | self_links_ignored | dup_links_ignored | leaf_nodes | singleton_nodes |
|---|---|---|---|---|---|---|---|
| 16 | 15 | 0.9375 | 0.0625 | 1 | 0 | 5 | 2 |

| concomp | isolated_pairs | isolated_stars_out | isolated_stars_in |
|---|---|---|---|
| 13 | 1 | 1 | 1 |

### *Basic Summary Statistics and Shortest Paths*

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. Because the graph is disconnected, we use the FINITEPATH option in the SUMMARY statement so that the shortest path descriptive statistics only consider finite paths.

```
proc optnetwork
   direction       = directed
   nodes           = mycas.NodeSetIn
   links           = mycas.LinkSetIn
   outNodes        = mycas.NodeSetOut;
   summary
      finitePath
      out          = mycas.Summary
      shortestPath = weight;
run;
```

The output data table mycas.Summary contains the summary and shortest path statistics of the input graph, as shown in Figure 3.105.

**Figure 3.105** Graph Summary and Shortest Path Statistics of an Undirected Graph

| nodes | links | avg_links_per_node | density | self_links_ignored | dup_links_ignored | leaf_nodes | singleton_nodes | diameter_wt | avg_shortpath_wt |
|---|---|---|---|---|---|---|---|---|---|
| 16 | 15 | 0.9375 | 0.0625 | 1 | 0 | 5 | 2 | 6 | 2.90476 |

The output data table mycas.NodeSetOut contains per-node summary and shortest path statistics of the input graph, as shown in Figure 3.106.

**Figure 3.106** Per-Node Summary and Shortest Path Statistics of an Undirected Graph

| node | leaf_node | singleton_node | neighbor_leaf_nodes | sum_in_and_out_wt | eccentr_wt_out | eccentr_wt_in |
|---|---|---|---|---|---|---|
| A | 0 | 0 | 1 | 9 | 4 | 2 |
| B | 0 | 0 | 0 | 5 | 6 | 1 |
| C | 1 | 0 | 0 | 2 | 0 | 4 |
| D | 0 | 0 | 0 | 7 | 2 | 4 |
| E | 0 | 0 | 0 | 5 | 4 | 6 |
| F | 0 | 0 | 0 | 6 | 2 | 5 |
| G | 0 | 1 | 0 | 0 | 0 | 0 |
| H | 0 | 1 | 0 | 0 | 0 | 0 |
| I | 0 | 0 | 1 | 5 | 5 | 0 |
| J | 1 | 0 | 0 | 5 | 0 | 5 |
| K | 0 | 0 | 2 | 5 | 3 | 0 |
| L | 1 | 0 | 0 | 3 | 0 | 3 |
| M | 1 | 0 | 0 | 2 | 0 | 2 |
| N | 0 | 0 | 1 | 1 | 1 | 0 |
| O | 1 | 0 | 0 | 6 | 0 | 5 |
| P | 0 | 0 | 1 | 5 | 5 | 0 |

## Transitive Closure

The *transitive closure* of a graph $G$ is a graph $G^T = (N, E^T)$ such that for all $i, j \in N$ there is a link $(i, j) \in E^T$ 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 39.

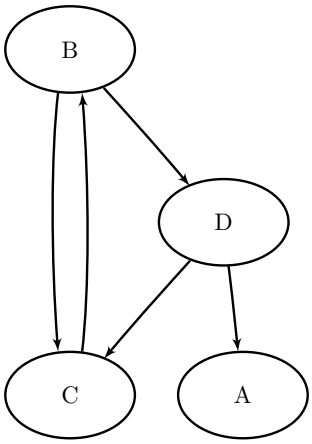
The links that define the transitive closure of the input graph are written to the output data table that you specify in the OUT= option in the TRANSITIVECLOSURE statement.

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 Directed Graph

This example illustrates the use of the transitive closure algorithm on the directed graph $G$ shown in Figure 3.107.

**Figure 3.107** 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;
```

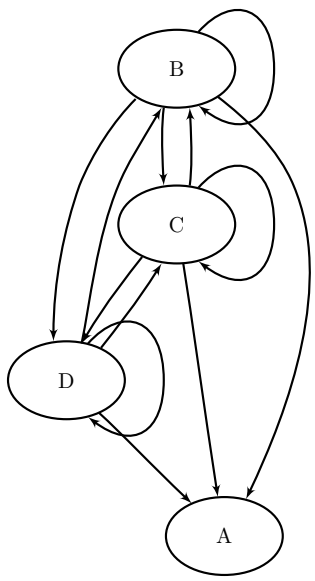
The output data table mycas.TransClosure contains the transitive closure of $G$, as shown in Figure 3.108.

**Figure 3.108** Transitive Closure of a Directed Graph

| from | to |
|---|---|
| B | C |
| B | D |
| C | B |
| D | A |
| D | C |
| B | B |
| D | B |
| C | C |
| C | D |
| D | D |
| B | A |
| C | A |

The transitive closure of $G$ is shown graphically in Figure 3.109.

**Figure 3.109** 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 154.

## 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 ($E$). 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 otherwise no node appears more than once in the sequence. A *Hamiltonian cycle* (or *tour*) is an elementary cycle that visits every node. In solving the TSP, then, the goal is to find a Hamiltonian cycle of minimum total cost, where the total cost is the sum of the costs of the links in the tour. Associated with each link $e \in E$ are a binary variable $x_e$, which indicates whether link $x_e$ is part of the tour, and a cost $c_e$. Let $\delta(S) = \{e \in E\colon \mathrm{from}(e) \in S,\ \mathrm{to}(e) \notin S\}$, where $\mathrm{from}(e)$ represents the *from* node of link $e$ and $\mathrm{to}(e)$ represents the *to* node of link $e$. Then an integer linear programming formulation of the TSP (for an undirected graph $G$) is as follows:

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

The two_match equations represent the *matching constraints*, which ensure that each node has degree 2 in the subgraph. The subtour_elim inequalities represent 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 39.

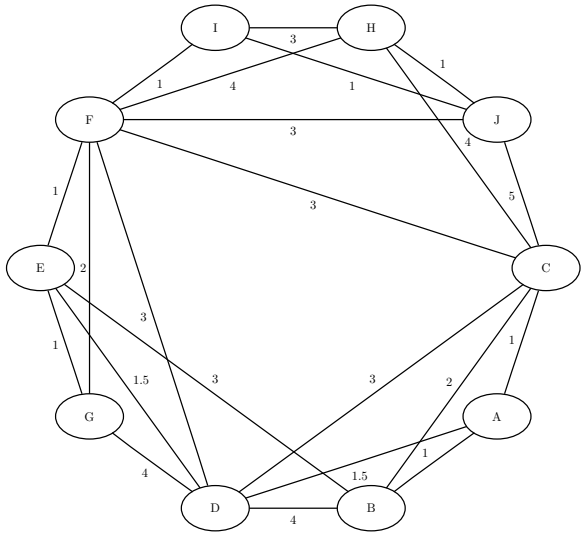
The algorithm that PROC OPTNETWORK uses for solving the TSP is based on a variant of the branch-and-cut process described in Applegate et al. (2006).

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 an Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 3.110.

**Figure 3.110** 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 3.111.

**Figure 3.111** PROC OPTNETWORK Log: Optimal Traveling Salesman Tour of an 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 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 using 1 threads across 1 machines.
NOTE: The initial TSP heuristics found a tour with cost 16 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     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 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.087209 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=16  CPU_TIME=0.11  REAL_TIME=0.09
```

The output data table mycas.NodeSetOut now contains a sequence of nodes in the optimal tour and is shown in Figure 3.112.

**Figure 3.112** 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 |

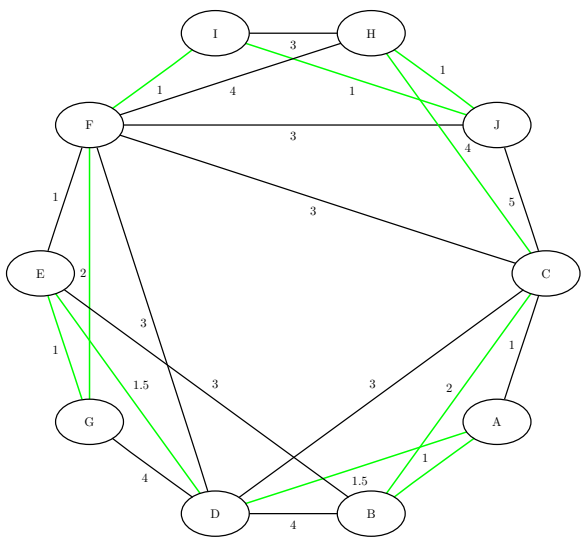
The output data table mycas.TSPTour now contains a sequence of links in the optimal tour and is shown in Figure 3.113.

**Figure 3.113** 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 3.114.

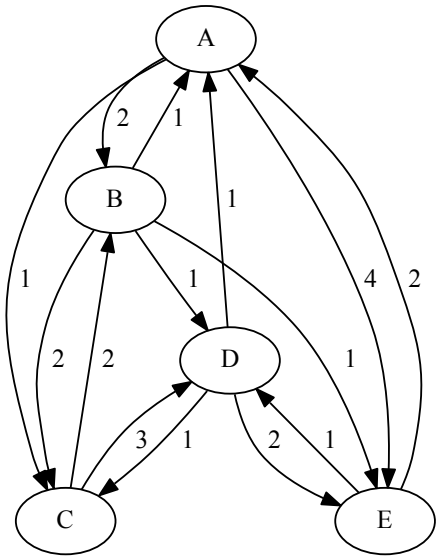
**Figure 3.114** Optimal Traveling Salesman Tour




## Traveling Salesman Problem Applied to a Directed Graph

As another simple example, consider the weighted directed graph in Figure 3.115. 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 3.115** 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 3.116.

**Figure 3.116** PROC OPTNETWORK Log: Optimal Traveling Salesman Tour of a 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 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 using 1 threads across 1 machines.
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 0.044593 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=0.10  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 3.117.

**Figure 3.117** Nodes in the Optimal Traveling Salesman Tour

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

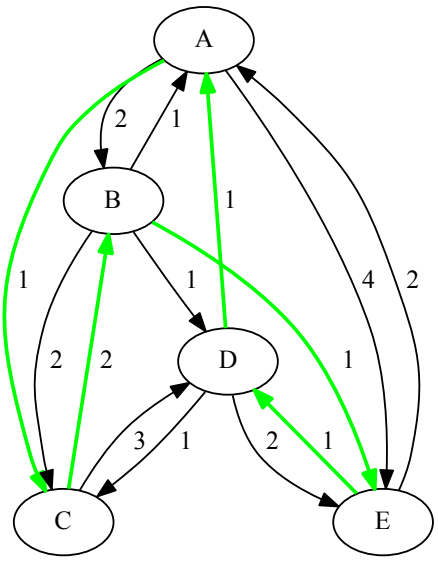
The output data table mycas.TSPTour now contains a sequence of links in the optimal tour and is shown in Figure 3.118.

**Figure 3.118** 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 3.119.

**Figure 3.119** 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 |
| LOADGRAPH | Loading a graph (no algorithm) |
| MINCOSTFLOW | Minimum-cost network flow |
| MINCUT | Minimum cut |
| MINSPANTREE | Minimum spanning tree |
| PATH | Path enumeration |
| READGRAPH | Reading a graph (no algorithm) |
| SHORTESTPATH | Shortest path |
| SUMMARY | Graph summary |
| TRANSITIVECLOSURE | Transitive closure |
| TSP | Traveling salesman |
| UNLOADGRAPH | Unloading a graph (no algorithm) |

**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 LOADGRAPH statement provides the following additional information:

**CREATETIME**
indicates the creation time of the in-memory graph.

**GRAPH**
indicates the in-memory graph identifier.

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 3.9.

**Table 3.9** ODS Tables Produced by PROC OPTNETWORK

| Table Name | Description |
|---|---|
| ProblemSummary | Summary of the graph |
| SolutionSummary | Summary of the solution status, timing, and results |
| OutputCasTables | See the section "OutputCasTables Table" on page 142 |

The following statements use the example in the section "Shortest Paths for All Pairs" on page 112 and find all-pairs shortest paths for a small undirected graph. By default, this code produces the two ODS output tables listed in Table 3.9.

```
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 3.120 provides a basic summary of the graph input.

**Figure 3.120** 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 3.121 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 136. 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 reading, loading, or unloading a graph (with no algorithm), the time in the solution summary represents the input and graph building time. 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 3.121** Solution Summary Table

| Solution Summary | |
|---|---|
| **Problem Type** | Shortest Path |
| **Solution Status** | OK |
| **Number of Paths** | 30 |
| **CPU Time** | 0.01 |
| **Real Time** | 0.00 |

### OutputCasTables Table

The OutputCasTables table is a special table that has information about each CAS table that is created during a CAS action execution. The information for each CAS table consists of the CAS table name, the caslib in which the table resides, and the number of columns and rows in the CAS table. Because this table is not a typical ODS table that contains analytical results, you cannot include it in the *table-spec-list* in the DISPLAYOUT statement.

# 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 3.122 shows this network, which was constructed after the attacks, based on collected intelligence information.

**Figure 3.122** 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.

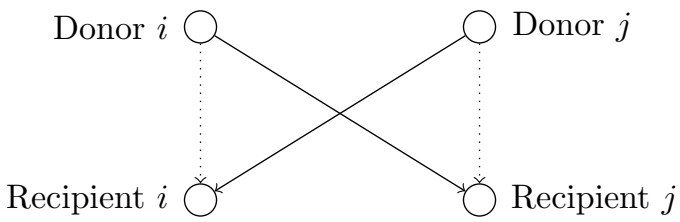
**Output 3.1.1** Articulation Points of Terrorist Communications Network from 9/11

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

## Example 3.2: Cycle Enumeration for Kidney Donor Exchange

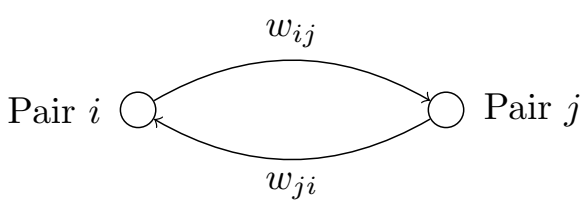
This example looks at an application of cycle enumeration to help create a kidney donor exchange. Suppose someone needs a kidney transplant and a family member is willing to be a donor. If the donor and recipient are incompatible (because of blood type, tissue mismatch, and so on), the transplant cannot 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 3.123. More generally, an $n$-way swap can be performed involving $n$ donors and $n$ recipients (CNN 2012).

**Figure 3.123** Kidney Donor Exchange Two-Way Swap




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 3.124.

**Figure 3.124** Kidney Donor Exchange Network




The link weight is a measure of the quality of the match. By introducing dummy links whose weight is 0, you can also include recipients who have no donors and altruistic donors who have no recipients. 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 88. 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 use 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 link 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
      maxCycles      = all
      outCyclesLinks = mycas.CyclesLinks;
run;
%put &_OROPTNETWORK_;
```

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

**Output 3.2.1** PROC OPTNETWORK Log: Cycles for Kidney Donor Exchange

```
                                    The SAS 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 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 enumeration using 32 threads across 1 machines.
NOTE: Processing cycle enumeration using the build algorithm.
NOTE: The algorithm found 395 cycles.
NOTE: Processing cycle enumeration used 0.00 (cpu: 0.01) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.047847 seconds.
NOTE: The data set MYCAS.CYCLESLINKS has 3431 observations and 5 variables.
STATUS=OK  PROBLEM_TYPE=CYCLE  SOLUTION_STATUS=OK  NUM_CYCLES=395  CPU_TIME=0.10  REAL_TIME=0.05
```

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 $c$, $E_c$ define the set of links in cycle $c$, and $w_e$ denote the link weight for link $e$. 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 in order to maximize the quality of the kidney exchange:

$$
\begin{aligned}
\text{maximize} \quad & \sum_{c \in C} \left( \sum_{e \in E_c} w_e \right) x_c \\
\text{subject to} \quad & \sum_{c \in C : i \in N_c} x_c \leq 1 \qquad i \in N \qquad \text{(incomp_pair)} \\
& x_c \in \{0, 1\} \qquad c \in C
\end{aligned}
$$

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> LINKS;
   num weight {LINKS};
   read data mycas.LinkSetIn into LINKS=[from to] weight;
   set <num,num,num> TRIPLES;
   read data mycas.CyclesLinks into TRIPLES=[cycle from to];
   set CYCLES = setof {<c,i,j> in TRIPLES} c;
   set LINKS_c {c in CYCLES} = setof {<(c),i,j> in TRIPLES} <i,j>;
   set NODES_c {c in CYCLES} = union {<i,j> in LINKS_c[c]} {i,j};
```

```
   set NODES = union {c in CYCLES} NODES_c[c];
   num cycle_weight {c in CYCLES} = sum {<i,j> in LINKS_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 NodePacking {i in NODES}:
      sum {c in CYCLES: i in NODES_c[c]} UseCycle[c] <= 1;

   /* call solver */
   solve;

   /* 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** PROC OPTMODEL Log: Cycles for Kidney Donor Exchange

```
NOTE: There were 208 observations read from the data set MYCAS.LINKSETIN.
NOTE: There were 3431 observations read from the data set MYCAS.CYCLESLINKS.
NOTE: Problem generation will use 16 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 OPTMODEL presolver is disabled for linear problems.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 122 variables and 30 constraints.
NOTE: The MILP presolver removed 1720 constraint coefficients.
NOTE: The MILP presolver modified 6 constraint coefficients.
NOTE: The presolved problem has 273 variables, 34 constraints, and 1711 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 16 threads.
          Node   Active   Sols    BestInteger      BestBound      Gap    Time
             0        1      3     22.3747690   1160.1140129   98.07%       0
             0        1      3     22.3747690     25.4194215   11.98%       0
             0        1      3     22.3747690     24.9759382   10.41%       0
             0        1      4     24.8508554     24.8508554    0.00%       0
             0        0      4     24.8508554     24.8508554    0.00%       0
NOTE: The MILP solver added 15 cuts with 1584 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=5.551115E-15 BOUND_INFEASIBILITY=5.551115E-15
INTEGER_INFEASIBILITY=5.551115E-15 BEST_BOUND=24.850855395 NODES=1 SOLUTIONS_FOUND=4
ITERATIONS=122 PRESOLVE_TIME=0.03 SOLUTION_TIME=0.21
```

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 |
|---|---|
| 26 | 4.3542 |
| 62 | 4.3403 |
| 121 | 4.9748 |
| 155 | 5.0843 |
| 362 | 1.9424 |
| 385 | 1.7253 |
| 392 | 2.4295 |
| | **24.8509** |

## Example 3.3: Linear Assignment Problem for Minimizing Relay Times

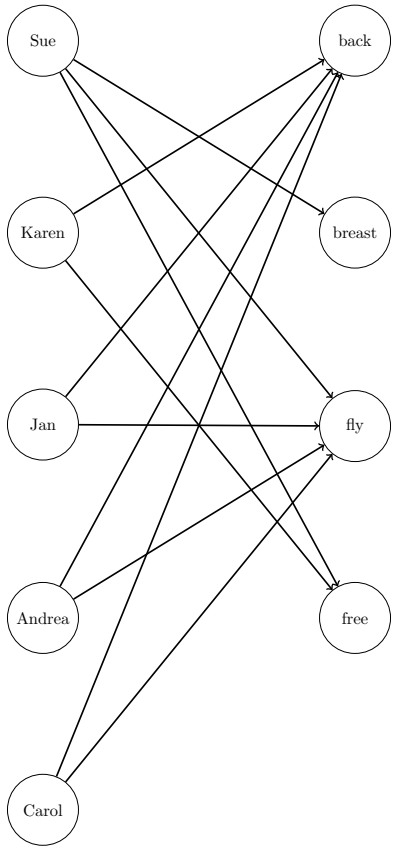
A swimming coach needs to assign a swimmer to each leg of a medley relay team; each swimmer in the relay uses a different stroke (backstroke, breaststroke, butterfly, or freestyle). The swimmers' best times for each stroke are stored in a SAS data set. The LINEARASSIGNMENT statement evaluates the times 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 3.125.

**Figure 3.125** 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;
```

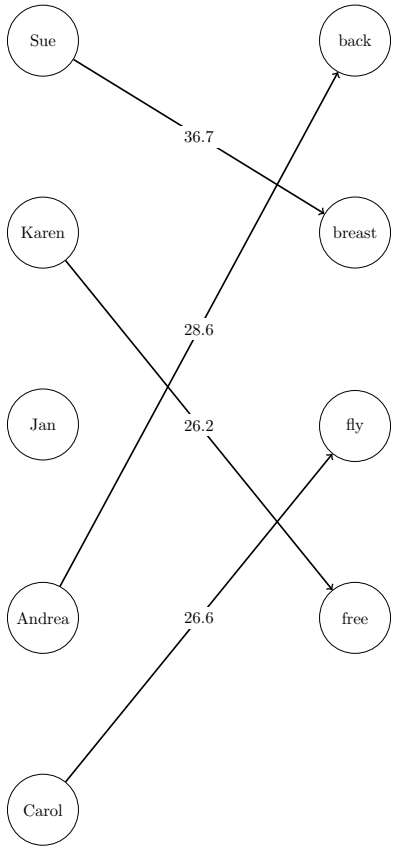
The output data table mycas.LinearAssignLinks contains the optimal assignment, as shown in Output 3.3.1.

**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 3.126.

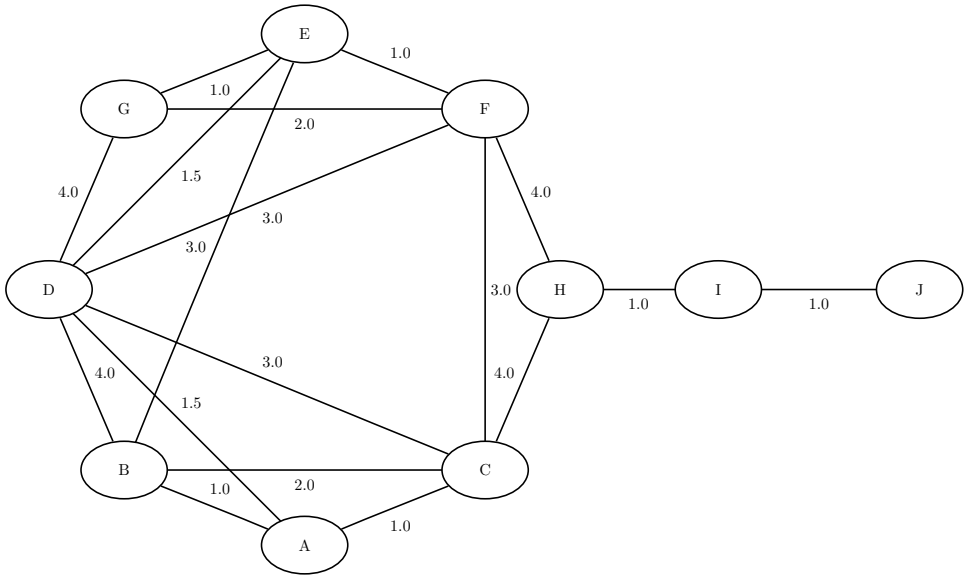
**Figure 3.126** Optimal Assignments for Swim Times




## 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 3.127. 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 3.127** Potential Office Computer Network




Define the links data table as follows:

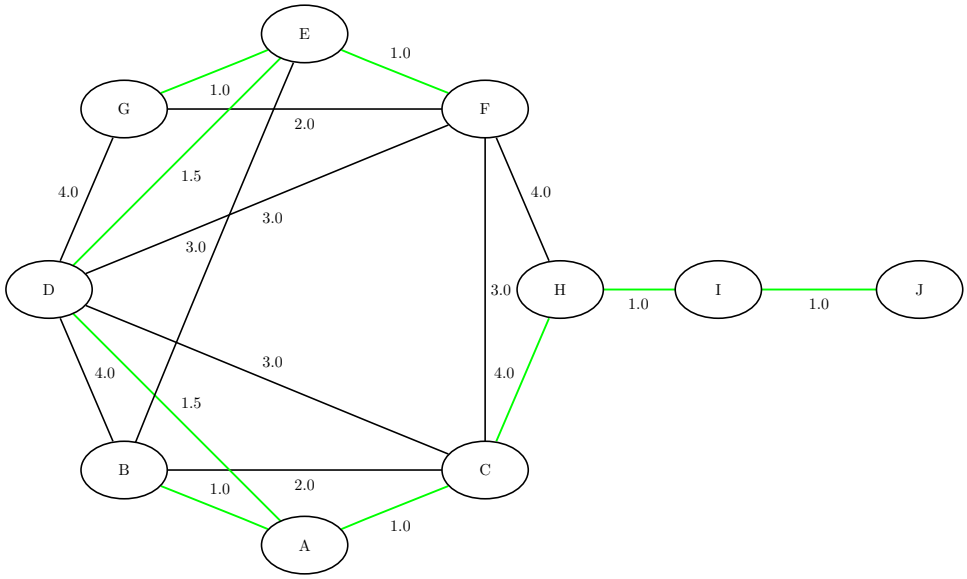
```
data mycas.LinkSetInCompNet;
   input from $ to $ weight @@;
   datalines;
A B 1.0  A C 1.0  A D 1.5  B C 2.0  B D 4.0
B E 3.0  C D 3.0  C F 3.0  C H 4.0  D E 1.5
D F 3.0  D G 4.0  E F 1.0  E G 1.0  F G 2.0
F H 4.0  H I 1.0  I J 1.0
;
```

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 3.128, with the minimal cost links shown in green.

**Figure 3.128** Minimum Spanning Tree for Office Computer Network




**Output 3.4.1** Minimum Spanning Tree for a Computer Network Design

| from | to | weight |
|---|---|---|
| A | B | 1.0 |
| A | C | 1.0 |
| E | F | 1.0 |
| E | G | 1.0 |
| H | I | 1.0 |
| I | J | 1.0 |
| A | D | 1.5 |
| D | E | 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 82. 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 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 enumeration using 1 threads across 1 machines.
NOTE: Processing cycle enumeration using the backtrack algorithm.
NOTE: The algorithm found 1 cycles.
NOTE: Processing cycle enumeration used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.083046 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=0.15  REAL_TIME=0.08
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 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 using 1 threads across 1 machines.
NOTE: Processing the transitive closure used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.05328 seconds.
NOTE: The data set MYCAS.TRANSCLOSURE has 20 observations and 2 variables.
STATUS=OK  PROBLEM_TYPE=TRANSITIVECLOSURE  SOLUTION_STATUS=OK  CPU_TIME=0.14  REAL_TIME=0.05
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 130.

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

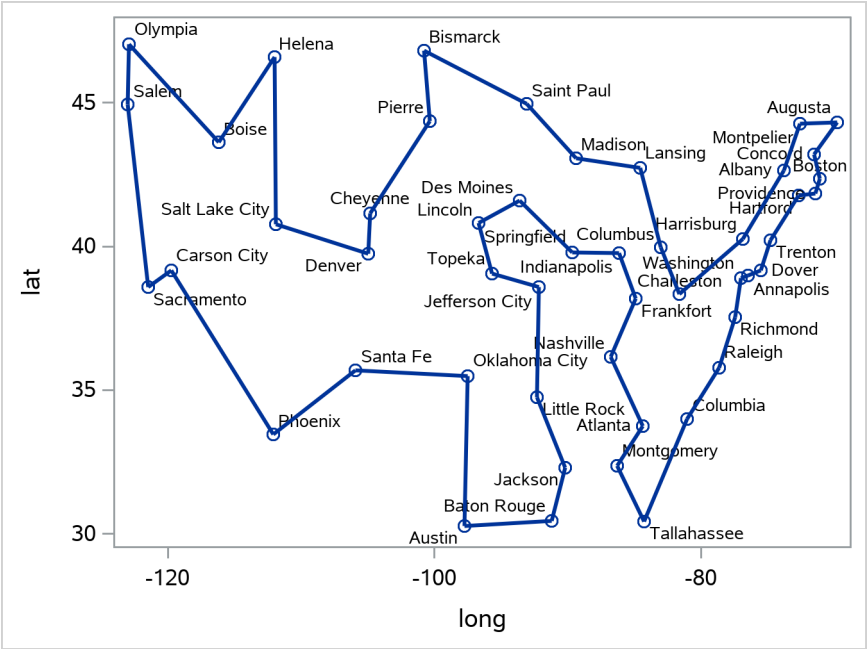
```
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Running OPTNETWORK.
NOTE: ----------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------
NOTE: Reading the links data.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: Building the input 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 using 1 threads across 1 machines.
NOTE: The initial TSP heuristics found a tour with cost 10635.088264 using 0.08 (cpu: 0.08)
      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  10635.0882639  10054.2390851    5.78%       0
             0        1      1  10635.0882639  10260.8132168    3.65%       0
             0        1      1  10635.0882639  10281.6424575    3.44%       0
             0        1      1  10635.0882639  10283.6430214    3.42%       0
             0        1      1  10635.0882639  10313.3113690    3.12%       0
             0        1      1  10635.0882639  10406.5613211    2.20%       0
             0        1      1  10635.0882639  10486.4120251    1.42%       0
             0        1      1  10635.0882639  10496.2272412    1.32%       0
             0        1      1  10635.0882639  10547.4389577    0.83%       0
             0        1      1  10635.0882639  10578.8621709    0.53%       0
             0        1      1  10635.0882639  10622.9107653    0.11%       0
             0        1      1  10635.0882639  10627.5814570    0.07%       0
             0        1      2  10635.0882639  10635.0882639    0.00%       0
             0        0      2  10635.0882639  10635.0882639    0.00%       0
NOTE: The MILP solver added 13 cuts with 3626 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 10635.088264.
NOTE: Processing the traveling salesman problem used 0.10 (cpu: 0.10) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.166938 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=2  OBJECTIVE=10635.088264
RELATIVE_GAP=3.42E-16  ABSOLUTE_GAP=-3.63798E-12  PRIMAL_INFEASIBILITY=0  BOUND_INFEASIBILITY=0
 INTEGER_INFEASIBILITY=0  BEST_BOUND=10635.088264  NODES=1  ITERATIONS=177  CPU_TIME=0.28
REAL_TIME=0.17
```

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 Output 3.6.3.

**Output 3.6.3** Links in the Optimal Traveling Salesman Tour

| city1 | city2 | distance |
|---|---|---|
| Albany, NY | Harrisburg, PA | 230.66 |
| Harrisburg, PA | Charleston, WV | 286.98 |
| Charleston, WV | Columbus, OH | 134.58 |
| Columbus, OH | Lansing, MI | 207.73 |
| Lansing, MI | Madison, WI | 246.15 |
| Madison, WI | Saint Paul, MN | 226.17 |
| Saint Paul, MN | Bismarck, ND | 392.00 |
| Bismarck, ND | Pierre, SD | 170.77 |
| Pierre, SD | Cheyenne, WY | 318.33 |
| Cheyenne, WY | Denver, CO | 97.07 |
| Denver, CO | Salt Lake City, UT | 371.76 |
| Salt Lake City, UT | Helena, MT | 401.06 |
| Helena, MT | Boise, ID | 289.44 |
| Boise, ID | Olympia, WA | 402.81 |
| Olympia, WA | Salem, OR | 144.92 |
| Salem, OR | Sacramento, CA | 446.18 |
| Sacramento, CA | Carson City, NV | 101.57 |
| Carson City, NV | Phoenix, AZ | 581.40 |
| Phoenix, AZ | Santa Fe, NM | 383.30 |
| Santa Fe, NM | Oklahoma City, OK | 475.13 |
| Oklahoma City, OK | Austin, TX | 359.82 |
| Austin, TX | Baton Rouge, LA | 391.58 |
| Baton Rouge, LA | Jackson, MS | 140.42 |
| Jackson, MS | Little Rock, AR | 207.66 |
| Little Rock, AR | Jefferson City, MO | 264.34 |
| Jefferson City, MO | Topeka, KS | 191.90 |
| Topeka, KS | Lincoln, NE | 133.00 |
| Lincoln, NE | Des Moines, IA | 170.15 |
| Des Moines, IA | Springfield, IL | 241.45 |
| Springfield, IL | Indianapolis, IN | 185.89 |
| Indianapolis, IN | Frankfort, KY | 129.26 |
| Frankfort, KY | Nashville, TN | 174.68 |
| Nashville, TN | Atlanta, GA | 215.02 |
| Atlanta, GA | Montgomery, AL | 145.79 |
| Montgomery, AL | Tallahassee, FL | 179.13 |
| Tallahassee, FL | Columbia, SC | 310.53 |
| Columbia, SC | Raleigh, NC | 183.13 |
| Raleigh, NC | Richmond, VA | 138.52 |
| Richmond, VA | Washington, DC | 96.00 |
| Washington, DC | Annapolis, MD | 28.57 |
| Annapolis, MD | Dover, DE | 53.67 |
| Dover, DE | Trenton, NJ | 83.53 |
| Trenton, NJ | Hartford, CT | 152.49 |
| Hartford, CT | Providence, RI | 65.63 |
| Providence, RI | Boston, MA | 40.60 |
| Boston, MA | Concord, NH | 63.38 |
| Concord, NH | Augusta, ME | 116.18 |
| Augusta, ME | Montpelier, VT | 138.86 |
| Montpelier, VT | Albany, NY | 125.89 |
| | | **10,635.09** |

## 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.NodeSetOut;
   connectedComponents
      out       = mycas.ConCompOut
      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 threads across 4 machines.
NOTE: The graph has 3627 connected components.
NOTE: Processing connected components used 1.10 (cpu: 2.55) seconds.
NOTE: The Cloud Analytic Services server processed the request in 7.329512 seconds.
NOTE: The data set MYCAS.NODESETOUT has 3774768 observations and 2 variables.
NOTE: The data set MYCAS.CONCOMPOUT has 3627 observations and 2 variables.
STATUS=OK  PROBLEM_TYPE=CONNECTEDCOMPONENTS  SOLUTION_STATUS=OK  NUM_COMPONENTS=3627
CPU_TIME=35.64  REAL_TIME=7.33
```

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 | nodes |
|---|---|---|
| 1 | 1 | 3764117 |
| 2 | 299 | 19 |
| 3 | 146 | 16 |
| 4 | 220 | 15 |
| 5 | 8 | 14 |
| 6 | 388 | 14 |
| 7 | 421 | 14 |
| 8 | 1911 | 14 |
| 9 | 11 | 13 |
| 10 | 81 | 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 733,846 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 100 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
   direction        = directed
   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 0.53 (cpu: 16.19) seconds.
NOTE: Building the input graph storage used 0.88 (cpu: 64.24) seconds.
NOTE: The number of nodes in the input graph is 264346.
NOTE: The number of links in the input graph is 733846.
NOTE: Processing shortest paths problem using 3200 threads across 100 machines.
NOTE: Processing the shortest paths problem between 264346 source nodes and 264346 sink nodes.
                                                    Real
      Algorithm              Sources   Complete     Time
      shortestPath             30400        11%     9.10
      shortestPath             62295        23%    19.36
      shortestPath            120445        45%    29.17
      shortestPath            167520        63%    39.19
      shortestPath            216480        81%    49.25
      shortestPath            259546        98%    59.03
      shortestPath            264346       100%    60.56
NOTE: Processing the shortest paths problem used 59.08 (cpu: 142245.36) seconds.
NOTE: The Cloud Analytic Services server processed the request in 61.701771 seconds.
NOTE: The data set MYCAS.SHORTPATHSUMMARY has 104263396 observations and 3 variables.
NOTE: The data set MYCAS.SHORTPATHPATHS has 1419295876 observations and 6 variables.
STATUS=OK  PROBLEM_TYPE=SHORTESTPATH  SOLUTION_STATUS=OK  NUM_PATHS=104263396
CPU_TIME=143425.34  REAL_TIME=61.70
```

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 12. 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 across 1 machines.
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 across 1 machines.
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=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 across 1 machines.
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(tiarno)' has 3 rows and 5 columns.
NOTE: The CAS table 'SOLUTIONSUMMARY' in caslib 'CASUSERHDFS(tiarno)' has 3 rows and 7 columns.
NOTE: The Cloud Analytic Services server processed the request in 0.100243 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=0.37  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 | 0.00 | 0.00 |
| 18-APR-2013 | 8:30:00 | Shortest Path | OK | 1 | 0.00 | 0.00 |

Output 3.9.4 displays the output data table mycas.ShortPathW, which shows the total time to travel on the best route for each time snapshot.

**Output 3.9.4** 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 |

Output 3.9.5 displays the output data table mycas.ShortPathP, which shows (by date and time) the best route for each time snapshot.

**Output 3.9.5** 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** |

**Output 3.9.5** *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.

CNN (2012). "Record Kidney Transplant Chain." CNN Health. Video. Accessed April 2, 2019. https://www.cnn.com/videos/health/2012/02/20/dnt-kidney-transplant-chain.wls.

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.

# Index