SAS/OR® 15.1 User's Guide

Network Optimization Algorithms

The OPTNET Procedure

This document is an individual chapter from *SAS/OR® 15.1 User's Guide: Network Optimization Algorithms*.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2018. *SAS/OR® 15.1 User's Guide: Network Optimization Algorithms*. Cary, NC: SAS Institute Inc.

SAS/OR® 15.1 User's Guide: Network Optimization Algorithms

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

November 2018

Chapter 2
The OPTNET Procedure

Contents

Overview: OPTNET Procedure

The OPTNET procedure includes a number of graph theory, combinatorial optimization, and network analysis algorithms. The algorithm classes are listed in Table 2.1.

Table 2.1 Algorithm Classes in PROC OPTNET

Algorithm Class	PROC OPTNET Statement
Biconnected components	BICONCOMP
Maximal cliques	CLIQUE
Connected components	CONCOMP
Cycle detection	CYCLE
Weighted matching	LINEAR_ASSIGNMENT
Minimum-cost network flow	MINCOSTFLOW
Minimum cut	MINCUT
Minimum spanning tree	MINSPANTREE
Shortest path	SHORTPATH
Transitive closure	TRANSITIVE_CLOSURE
Traveling salesman	TSP

You can use the OPTNET procedure to analyze relationships between entities. These relationships are typically defined by using a *graph*. A graph $G = (N, A)$ is defined over a set N of nodes and a set A of arcs. A *node* is an abstract representation of some entity (or object), and an *arc* defines some relationship (or connection) between two nodes. The terms *node* and *vertex* are often interchanged in describing an entity. The term *arc* is often interchanged with the term *edge* or *link* when describing a connection.

You can also access these network algorithms via the network solver in PROC OPTMODEL. For more information, see the network solver chapter in *SAS/OR User's Guide: Mathematical Programming*.

Getting Started: OPTNET Procedure

Since 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 document shows a few potential applications through simple examples.

This section shows an introductory example for getting started with the OPTNET procedure. For more detail about the input formats expected and the various algorithms available, see the sections "Details: OPTNET Procedure" on page 34 and "Examples: OPTNET Procedure" on page 107.

Road Network Shortest Path

Consider the following road network between a SAS employee's home in Raleigh, NC, and the SAS headquarters in Cary, NC.

In this road network (graph), the links are the roads and the nodes are intersections between roads. For each road, you assign a *link attribute* in the 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 traverse between two points, based on the length of the road and the typical speed during normal traffic patterns:

```
data LinkSetInRoadNC10am;
   input start_inter $1-20 end_inter $20-40 miles miles_per_hour;
   datalines;
614CapitalBlvd      Capital/WadeAve      0.6  25
614CapitalBlvd      Capital/US70W        0.6  25
614CapitalBlvd      Capital/US440W       3.0  45
Capital/WadeAve     WadeAve/RaleighExpy  3.0  40
Capital/US70W       US70W/US440W         3.2  60
US70W/US440W        US440W/RaleighExpy   2.7  60
Capital/US440W      US440W/RaleighExpy   6.7  60
US440W/RaleighExpy  RaleighExpy/US40W    3.0  60
WadeAve/RaleighExpy RaleighExpy/US40W    3.0  60
RaleighExpy/US40W   US40W/HarrisonAve    1.3  55
US40W/HarrisonAve   SASCampusDrive       0.5  25
;

data LinkSetInRoadNC10am;
   set LinkSetInRoadNC10am;
   time_to_travel = miles * 1/miles_per_hour * 60;
run;
```

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

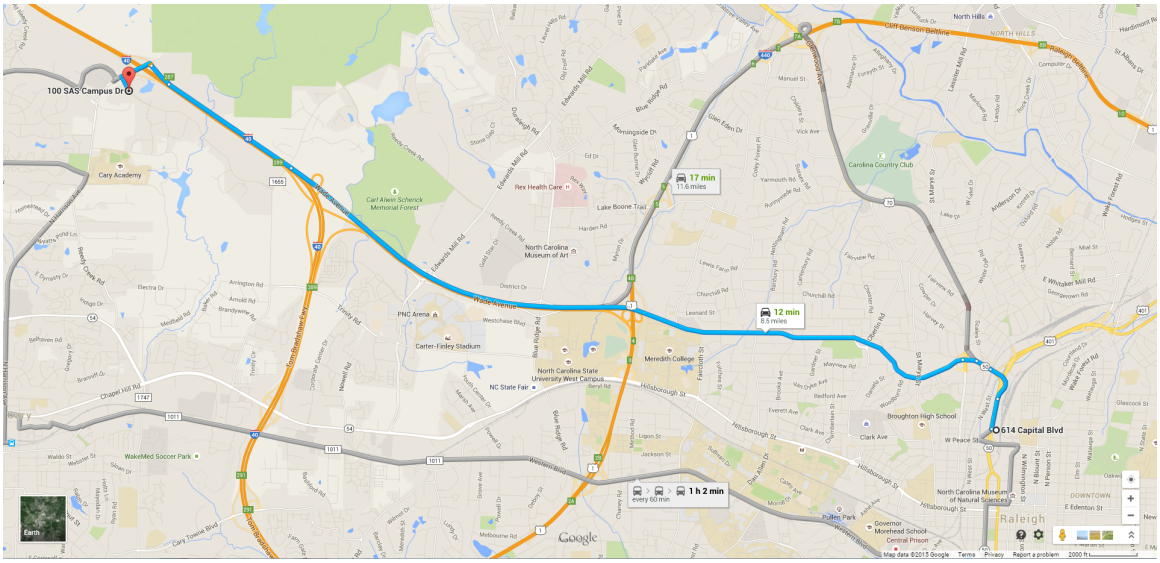
```
proc optnet
   data_links    = LinkSetInRoadNC10am;
   data_links_var
      from       = start_inter
      to         = end_inter
      weight     = time_to_travel;
   shortpath
      out_paths  = ShortPath
      source     = "614CapitalBlvd"
      sink       = "SASCampusDrive";
run;
```

For more information about shortest path algorithms in PROC OPTNET, see the section "Shortest Path" on page 76. Figure 2.1 displays the output data set ShortPath, which shows the best route to take to minimize travel time at 10:00 a.m. This route is also shown in Google Maps in Figure 2.2.

Figure 2.1 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 2.2 Shortest Path for Road Network at 10:00 A.M. in Google Maps



Now suppose that it is rush hour (5:00 p.m.) and the time to traverse the roads has changed because of traffic patterns. You want to find the route that is the shortest path for going home from SAS headquarters under different speed assumptions due to traffic. The following data set lists approximate travel times and speeds for driving in the opposite direction:

```
data LinkSetInRoadNC5pm;
   input start_inter $1-20 end_inter $20-40 miles miles_per_hour;
   datalines;
614CapitalBlvd      Capital/WadeAve      0.6  25
614CapitalBlvd      Capital/US70W        0.6  25
614CapitalBlvd      Capital/US440W       3.0  45
Capital/WadeAve     WadeAve/RaleighExpy  3.0  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
;

data LinkSetInRoadNC5pm;
   set LinkSetInRoadNC5pm;
   time_to_travel = miles * 1/miles_per_hour * 60;
run;
```

The following statements are similar to the first PROC OPTNET run, except that they use the data set LinkSetInRoadNC5pm and the SOURCE and SINK option values are reversed:

```
proc optnet
   data_links    = LinkSetInRoadNC5pm;
   data_links_var
      from       = start_inter
      to         = end_inter
      weight     = time_to_travel;
   shortpath
      out_paths  = ShortPath
      source     = "SASCampusDrive"
      sink       = "614CapitalBlvd";
run;
```

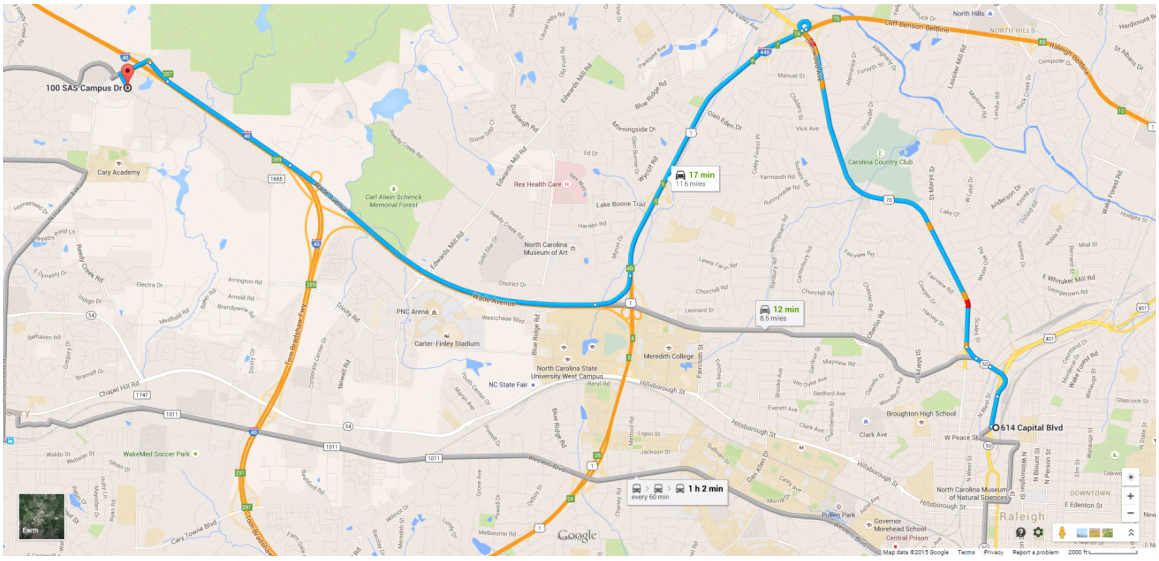
Now, the output data set ShortPath, shown in Figure 2.3, shows the best route for going home. Because the traffic on Wade Avenue is usually heavy at this time of day, the route home is different from the route to work.

Figure 2.3 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 2.4.

Figure 2.4 Shortest Path for Road Network at 5:00 P.M. in Google Maps



Syntax: OPTNET Procedure

PROC OPTNET *options* ;

Data Input Statements:
DATA_LINKS_VAR < *options* > ;
DATA_MATRIX_VAR <*column1,column2,. . .* > ;
DATA_NODES_VAR < *options* > ;

Algorithm Statements:
BICONCOMP < *option* > ;
CLIQUE < *options* > ;
CONCOMP < *options* > ;
CYCLE < *options* > ;
LINEAR_ASSIGNMENT < *options* > ;
MINCOSTFLOW < *options* > ;
MINCUT < *options* > ;
MINSPANTREE < *options* > ;
SHORTPATH < *options* > ;
TRANSITIVE_CLOSURE < *options* > ;
TSP < *options* > ;

Performance Statement:
PERFORMANCE < *options* > ;

PROC OPTNET statements are divided into four main categories: the PROC statement, the data input statements, the algorithm statements, and the PERFORMANCE statement. The PROC statement invokes

the procedure and sets option values that are used across multiple algorithms. The data input statements control the names of the variables that PROC OPTNET expects in the data input. The algorithm statements determine which algorithms are run and set options for each individual algorithm. The PERFORMANCE statement specifies performance options for multithreaded computing.

The section "Functional Summary" on page 11 provides a quick reference for each of the options for each statement. Each statement is then described in more detail in its own section; the PROC OPTNET statement is described first, and sections that describe all other statements are presented in alphabetical order.

Functional Summary

Table 2.2 summarizes the statements and options available with PROC OPTNET.

Table 2.2 Functional Summary

Description	Option
PROC OPTNET Options	
Input	
Specifies the link data set	DATA_LINKS=
Specifies the matrix data set	DATA_MATRIX=
Specifies the node data set	DATA_NODES=
Specifies the node subset data set	DATA_NODES_SUB=
Output	
Specifies the link output data set	OUT_LINKS=
Specifies the node output data set	OUT_NODES=
Options	
Specifies the graph direction	GRAPH_DIRECTION=
Specifies the internal graph format	GRAPH_INTERNAL_FORMAT=
Includes self links	INCLUDE_SELFLINK
Specifies the overall log level	LOGLEVEL=
Specifies whether time units are in CPU time or real time	TIMETYPE=
Data Input Statements	
DATA_LINKS_VAR Options	
Specifies the data set variable name for the *from* nodes	FROM=
Specifies the data set variable name for the link flow lower bounds	LOWER=
Specifies the data set variable name for the *to* nodes	TO=
Specifies the data set variable name for the link flow upper bounds	UPPER=
Specifies the data set variable name for the link weights	WEIGHT=
DATA_MATRIX_VAR	
Specifies the data set variable names for the matrix	
DATA_NODES_VAR Options	
Specifies the data set variable name for the nodes	NODE=
Specifies the data set variable name for node weights	WEIGHT=
Specifies the data set variable name for auxiliary node weights	WEIGHT2=

Table 2.2 (continued)

Description	Option
Algorithm Statements	
BICONCOMP Option	
Specifies the log level for biconnected components	LOGLEVEL=
CLIQUE Options	
Specifies the log level for clique calculations	LOGLEVEL=
Specifies the maximum number of cliques to return during clique calculations	MAXCLIQUES=
Specifies the maximum amount of time to spend calculating cliques	MAXTIME=
Specifies the output data set for cliques	OUT=
CONCOMP Options	
Specifies the algorithm to use for connected components	ALGORITHM=
Specifies the log level for connected components	LOGLEVEL=
CYCLE Options	
Specifies the log level for the cycle algorithm	LOGLEVEL=
Specifies the maximum number of cycles to return during cycle calculations	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 calculating 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 mode for the cycle calculations	MODE=
Specifies the output data set for cycles	OUT=
LINEAR_ASSIGNMENT Options	
Specifies the data set variable names for the linear assignment identifiers	ID=( )
Specifies the log level for the linear assignment algorithm	LOGLEVEL=
Specifies the output data set for linear assignment	OUT=
Specifies the data set variable names for costs (or weights)	WEIGHT=( )
MINCOSTFLOW Options	
Specifies the iteration log frequency	LOGFREQ=
Specifies the log level for the minimum-cost network flow algorithm	LOGLEVEL=
Specifies the maximum amount of time to spend calculating the optimal flow	MAXTIME=
MINCUT Options	
Specifies the log level for the minimum-cut algorithm	LOGLEVEL=
Specifies the maximum number of cuts to return	MAXNUMCUTS=
Specifies the maximum weight of the cuts to return	MAXWEIGHT=
Specifies the output data set for minimum cut	OUT=
MINSPANTREE Options	
Specifies the log level for the minimum spanning tree algorithm	LOGLEVEL=

Table 2.2 (continued)

Description	Option
Specifies the output data set for minimum spanning tree	OUT=
SHORTPATH Options	
Specifies the iteration log frequency (nodes)	LOGFREQ=
Specifies the log level for shortest paths	LOGLEVEL=
Specifies the output data set for shortest paths	OUT_PATHS=
Specifies the output data set for shortest path summaries	OUT_WEIGHTS=
Specifies the type of output for shortest paths results	PATHS=
Specifies the sink node for shortest paths calculations	SINK=
Specifies the source node for shortest paths calculations	SOURCE=
Specifies whether to use weights in calculating shortest paths	USEWEIGHT=
Specifies the data set variable name for the auxiliary link weights	WEIGHT2=
TRANSITIVE_CLOSURE Options	
Specifies the log level for transitive closure	LOGLEVEL=
Specifies the output data set for transitive closure results	OUT=
TSP Options	
Specifies the stopping criterion based on the absolute objective gap	ABSOBJGAP=
Specifies the level of conflict search	CONFLICTSEARCH=
Specifies the cutoff value for branch-and-bound node removal	CUTOFF=
Specifies the overall cut strategy level	CUTSTRATEGY=
Emphasizes feasibility or optimality	EMPHASIS=
Specifies the initial and primal heuristics level	HEURISTICS=
Specifies the frequency of printing the branch-and-bound node log	LOGFREQ=
Specifies the log level for the traveling salesman algorithm	LOGLEVEL=
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 in the algorithm	MAXTIME=
Specifies whether to use a mixed integer linear programming solver	MILP=
Specifies the branch-and-bound node selection strategy	NODESEL=
Specifies the output data set for traveling salesman problem	OUT=
Specifies the probing level	PROBE=
Specifies the stopping criterion based on the relative objective gap	RELOBJGAP=
Specifies the number of simplex iterations to be performed on each variable in the strong branching strategy	STRONGITER=
Specifies the number of candidates for the strong branching strategy	STRONGLEN=
Specifies the stopping criterion based on the target objective value	TARGET=
Specifies the rule for selecting branching variable	VARSEL=

For more information about the options available for the PERFORMANCE statement, see the section "PERFORMANCE Statement" on page 26.

Table 2.3 lists the valid input formats, GRAPH_DIRECTION= values, and GRAPH_INTERNAL_FORMAT= values for each statement in the OPTNET procedure.

Table 2.3 Supported Input Formats and Graph Types by Statement

	Input Format		DIRECTION		INTERNAL_FORMAT	
Statement	Graph	Matrix	UNDIRECTED	DIRECTED	THIN	FULL
BICONCOMP	X		X			X
CLIQUE	X		X			X
CONCOMP						
ALGORITHM=						
DFS	X		X	X		X
UNION_FIND	X		X		X	X
CYCLE	X		X	X		X
LINEAR_ASSIGNMENT	X	X		X		X
MINCOSTFLOW	X			X	X	X
MINCUT	X		X			X
MINSPANTREE	X		X		X	X
SHORTPATH	X		X	X		X
TRANSITIVE_CLOSURE	X		X	X		X
TSP	X		X	X		X

Table 2.4 indicates for each algorithm statement in the OPTNET procedure which output data set options you can specify and whether the algorithm populates the data sets specified in the OUT_NODES= and OUT_LINKS= options in the PROC OPTNET statement.

Table 2.4 Output Options by Statement

Statement	OUT_NODES	OUT_LINKS	Algorithm Statement Options
BICONCOMP	X	X	
CLIQUE			OUT=
CONCOMP	X		
CYCLE			OUT=
LINEAR_ASSIGNMENT			OUT=
MINCOSTFLOW		X	
MINCUT	X		OUT=
MINSPANTREE			OUT=
SHORTPATH			OUT_PATHS=, OUT_WEIGHTS=
TRANSITIVE_CLOSURE			OUT=
TSP	X		OUT=

PROC OPTNET Statement

PROC OPTNET < *options* > **;**

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

DATA_LINKS=*SAS-data-set*

LINKS=*SAS-data-set*

specifies the input data set that contains the graph link information, where the links are defined as a list.

See the section "Link Input Data" on page 34 for more information.

DATA_MATRIX=*SAS-data-set*

MATRIX=*SAS-data-set*

specifies the input data set that contains the matrix to be processed. This is a generic matrix (as opposed to an adjacency matrix, which defines an underlying graph).

See the section "Matrix Input Data" on page 41 for more information.

DATA_NODES=*SAS-data-set*

NODES=*SAS-data-set*

specifies the input data set that contains the graph node information.

See the section "Node Input Data" on page 38 for more information.

DATA_NODES_SUB=*SAS-data-set*

NODES_SUB=*SAS-data-set*

specifies the input data set that contains the graph node subset information.

See the section "Node Subset Input Data" on page 38 for more information.

GRAPH_DIRECTION=DIRECTED | UNDIRECTED

DIRECTION=DIRECTED | UNDIRECTED

specifies whether the input graph should be considered directed or undirected.

Table 2.5 Values for the GRAPH_DIRECTION= Option

Option Value	Description
DIRECTED	Specifies the graph as directed. In a directed graph, each link (i, j) has a direction that defines how something (for example, information) might flow over that link. In link (i, j), information flows from node i to node j $(i \rightarrow j)$. The node i is called the *source* (*tail*) node, and j is called the *sink* (*head*) node.
UNDIRECTED	Specifies the graph as undirected. In an undirected graph, each link $\{i, j\}$ has no direction and information can flow in either direction. That is, $\{i, j\} = \{j, i\}$. This is the default.

By default, GRAPH_DIRECTION=UNDIRECTED. See the section "Graph Input Data" on page 34 for more information.

GRAPH_INTERNAL_FORMAT=FULL | THIN

INTERNAL_FORMAT=FULL | THIN

requests the internal graph format for the algorithms to use.

Table 2.6 Values for the GRAPH_INTERNAL_FORMAT= Option

Option Value	Description
FULL	Stores the graph in standard (full) format. This is the default.
THIN (experimental)	Stores the graph in thin format. This option can improve performance in some cases both by reducing memory and by simplifying the construction of the internal data structures. The thin format causes PROC OPTNET to skip the removal of duplicate links when it reads in the graph. So this option should be used with caution. For some algorithms, the thin format is not allowed and this option is ignored. Setting GRAPH_INTERNAL_FORMAT= THIN can often be helpful when you do calculations that are decomposed by subgraph.

See the section "Graph Input Data" on page 34 for more information.

INCLUDE_SELFLINK

includes self links—for example, (i, i)—when an input graph is read. By default, when PROC OPTNET reads the DATA_LINKS= data set, it removes all self links.

LOGLEVEL=*number* | *string*

controls the amount of information that is displayed in the SAS log. Each algorithm has its own specific log level. This setting sets the log level for all algorithms except those for which you specify the LOGLEVEL= option in the algorithm statement. Table 2.7 describes the valid values for this option.

Table 2.7 Values for LOGLEVEL= Option

number	*string*	Description
0	NONE	Turns off all procedure-related messages in the SAS log
1	BASIC	Displays a basic summary of the input, output, and algorithmic processing
2	MODERATE	Displays a summary of the input, output, and algorithmic processing
3	AGGRESSIVE	Displays a detailed summary of the input, output, and algorithmic processing

By default, LOGLEVEL=BASIC.

OUT_LINKS=*SAS-data-set*

specifies the output data set to contain the graph link information along with any results from the various algorithms that calculate metrics on links.

See the various algorithm sections for examples of the content of this output data set.

OUT_NODES=*SAS-data-set*

specifies the output data set to contain the graph node information along with any results from the various algorithms that calculate metrics on nodes.

See the various algorithm sections for examples of the content of this output data set.

STANDARDIZED_LABELS
specifies that the input graph data is in a standardized format described in section "Standardized Labels" on page 39.

TIMETYPE=*number* | *string*
specifies whether CPU time or real time is used for the MAXTIME= option for each applicable algorithm. Table 2.8 describes the valid values of the TIMETYPE= option.

Table 2.8 Values for TIMETYPE= Option

number	*string*	**Description**
0	CPU	Specifies units of CPU time
1	REAL	Specifies units of real time

By default, TIMETYPE=CPU.

BICONCOMP Statement

BICONCOMP < *option* > **;**

The BICONCOMP statement requests that PROC OPTNET find biconnected components and articulation points of an undirected input graph.

See the section "Biconnected Components and Articulation Points" on page 44 for more information.

You can specify the following *option* in the BICONCOMP statement.

LOGLEVEL=*number* | *string*
controls the amount of information that is displayed in the SAS log. Table 2.9 describes the valid values for this option.

Table 2.9 Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turns off all algorithm-related messages in the SAS log
1	BASIC	Displays a basic summary of the algorithmic processing
2	MODERATE	Displays a summary of the algorithmic processing
3	AGGRESSIVE	Displays a detailed summary of the algorithmic processing

The default is the value that is specified in the LOGLEVEL= option in the PROC OPTNET statement (or BASIC if that option is not specified).

CLIQUE Statement

CLIQUE < *options* > **;**

The CLIQUE statement invokes an algorithm that finds maximal cliques on the input graph. Maximal cliques are described in the section "Clique" on page 47.

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

LOGLEVEL=*number* | *string*
controls the amount of information that is displayed in the SAS log. Table 2.10 describes the valid values for this option.

Table 2.10 Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turns off all algorithm-related messages in the SAS log
1	BASIC	Displays a basic summary of the algorithmic processing
2	MODERATE	Displays a summary of the algorithmic processing
3	AGGRESSIVE	Displays a detailed summary of the algorithmic processing

The default is the value that is specified in the LOGLEVEL= option in the PROC OPTNET statement (or BASIC if that option is not specified).

MAXCLIQUES=*number*
specifies the maximum number of cliques to return during clique calculations. The default is the positive number that has the largest absolute value that can be represented in your operating environment.

MAXTIME=*number*
specifies the maximum amount of time to spend calculating cliques. The type of time (either CPU time or real time) is determined by the value of the TIMETYPE= option. The value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

OUT=*SAS-data-set*
specifies the output data set to contain the maximal cliques.

CONCOMP Statement

CONCOMP < *options* > **;**

The CONCOMP statement invokes an algorithm that finds the connected components of the input graph. Connected components are described in the section "Connected Components" on page 51.

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

ALGORITHM=DFS | UNION_FIND
specifies the algorithm to use for calculating connected components.

Table 2.11 Values for the ALGORITHM= Option

Option Value	Description
DFS	Uses the depth-first search algorithm for connected components. You cannot specify this value when you specify GRAPH_INTERNAL_FORMAT=THIN in the PROC OPTNET statement.
UNION_FIND	Uses the union-find algorithm for connected components. You can specify this value with either the THIN or FULL value for the GRAPH_INTERNAL_FORMAT= option in the PROC OPTNET statement. This value can be faster than DFS when used with GRAPH_INTERNAL_FORMAT=THIN. However, you can use it only with undirected graphs.

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

LOGLEVEL=*number* | *string*
controls the amount of information that is displayed in the SAS log. Table 2.12 describes the valid values for this option.

Table 2.12 Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turns off all algorithm-related messages in the SAS log
1	BASIC	Displays a basic summary of the algorithmic processing
2	MODERATE	Displays a summary of the algorithmic processing
3	AGGRESSIVE	Displays a detailed summary of the algorithmic processing

The default is the value that is specified in the LOGLEVEL= option in the PROC OPTNET statement (or BASIC if that option is not specified).

CYCLE Statement

CYCLE < *options* > **;**

The CYCLE statement invokes an algorithm that finds the cycles (or the existence of a cycle) in the input graph. Cycles are described in the section "Cycle" on page 55.

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

LOGLEVEL=*number* | *string*
controls the amount of information that is displayed in the SAS log. Table 2.13 describes the valid values for this option.

Table 2.13 Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turns off all algorithm-related messages in the SAS log
1	BASIC	Displays a basic summary of the algorithmic processing
2	MODERATE	Displays a summary of the algorithmic processing
3	AGGRESSIVE	Displays a detailed summary of the algorithmic processing

The default is the value that is specified in the LOGLEVEL= option in the PROC OPTNET statement (or BASIC if that option is not specified).

MAXCYCLES=*number*

specifies the maximum number of cycles to return. The default is the positive number that has the largest absolute value representable in your operating environment. This option works only when you also specify MODE=ALL_CYCLES.

MAXLENGTH=*number*

specifies the maximum number of links to allow in a cycle. Any cycle whose length is greater than *number* is removed from the results. The default is the positive number that has the largest absolute value that can be represented in your operating environment. By default, nothing is removed from the results. This option works only when you also specify MODE=ALL_CYCLES.

MAXLINKWEIGHT=*number*

specifies the maximum sum of link weights to allow in a cycle. Any cycle whose sum of link weights is greater than *number* is removed from the results. The default is the positive number that has the largest absolute value that can be represented in your operating environment. By default, nothing is filtered. This option works only when you also specify MODE=ALL_CYCLES.

MAXNODEWEIGHT=*number*

specifies the maximum sum of node weights to allow in a cycle. Any cycle whose sum of node weights is greater than *number* is removed from the results. The default is the positive number that has the largest absolute value that can be represented in your operating environment. By default, nothing is filtered. This option works only when you also specify MODE=ALL_CYCLES.

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. The value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment. This option works only when you also specify MODE=ALL_CYCLES.

MINLENGTH=*number*

specifies the minimum number of links to allow in a cycle. Any cycle that has fewer links than *number* is removed from the results. By default MINLENGTH=1 and nothing is filtered. This option works only when you also specify MODE=ALL_CYCLES.

MINLINKWEIGHT=*number*
specifies the minimum sum of link weights to allow in a cycle. Any cycle whose sum of link weights is less than *number* is removed from the results. The default is the negative number that has the largest absolute value that can be represented in your operating environment. By default, nothing is filtered. This option works only when you also specify MODE=ALL_CYCLES.

MINNODEWEIGHT=*number*
specifies the minimum sum of node weights to allow in a cycle. Any cycle whose sum of node weights is less than *number* is removed from the results. The default is the negative number that has the largest absolute value that can be represented in your operating environment. By default, nothing is filtered. This option works only when you also specify MODE=ALL_CYCLES.

MODE=ALL_CYCLES | FIRST_CYCLE
specifies the mode for processing cycles.

Table 2.14 Values for the MODE= Option

Option Value	Description
ALL_CYCLES	Returns all (unique, elementary) cycles found.
FIRST_CYCLE	Returns the first cycle found. This is the default.

OUT=*SAS-data-set*
specifies the output data set to contain the cycles found.

DATA_LINKS_VAR Statement

DATA_LINKS_VAR < *options* > **;**

LINKS_VAR < *options* > **;**

The DATA_LINKS_VAR statement enables you to explicitly define the data set variable names for PROC OPTNET to use when it reads the data set that is specified in the DATA_LINKS= option in the PROC OPTNET statement. The format of the links input data set is defined in the section "Link Input Data" on page 34.

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

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

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

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

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

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

DATA_MATRIX_VAR Statement

DATA_MATRIX_VAR *<column1,column2,. . . >* **;**

MATRIX_VAR *<column1,column2,. . . >* **;**

The DATA_MATRIX_VAR statement enables you to explicitly define the data set variable names for PROC OPTNET to use when it reads the data set that is specified in the DATA_MATRIX= option in the PROC OPTNET statement. The format of the matrix input data set is defined in the section "Matrix Input Data" on page 41. The value of each *column* variable must be numeric.

DATA_NODES_VAR Statement

DATA_NODES_VAR *< options >* **;**

NODES_VAR *< options >* **;**

The DATA_NODES_VAR statement enables you to explicitly define the data set variable names for PROC OPTNET to use when it reads the data set that is specified in the DATA_NODES= option in the PROC OPTNET statement. The format of the node input data set is defined in the section "Node Input Data" on page 38.

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

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

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

WEIGHT2=*column*
specifies the data set variable name for auxiliary node weights. The value of the *column* variable must be numeric.

LINEAR_ASSIGNMENT Statement

LINEAR_ASSIGNMENT *< options >* **;**

LAP < *options* > **;**

The LINEAR_ASSIGNMENT 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. The input data (the cost matrix) is typically defined in the input data set that is specified in the DATA_MATRIX= option in the PROC OPTNET statement. The data can also be defined as a directed graph by specifying the DATA_LINKS= option in the PROC OPTNET 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 61.

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

ID=(*<column1,column2,. . . >***)**
specifies the data set variable names that identify the matrix rows (*from* nodes). The information in these columns is carried to the output data set that is specified in the OUT= option. The value of each *column* variable can be numeric or character.

LOGLEVEL=*number* | *string*
controls the amount of information that is displayed in the SAS log. Table 2.15 describes the valid values for this option.

Table 2.15 Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turns off all algorithm-related messages in the SAS log
1	BASIC	Displays a basic summary of the algorithmic processing
2	MODERATE	Displays a summary of the algorithmic processing
3	AGGRESSIVE	Displays a detailed summary of the algorithmic processing

The default is the value that is specified in the LOGLEVEL= option in the PROC OPTNET statement (or BASIC if that option is not specified).

OUT=*SAS-data-set*
specifies the output data set to contain the solution to the linear assignment problem.

WEIGHT=(*<column1,column2,. . . >***)**
specifies the data set variable names for the cost matrix. The value of each *column* variable must be numeric. If this option is not specified, the matrix is assumed to be defined by all of the numeric variables in the data set (excluding those specified in the ID= option).

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 62.

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

LOGFREQ=*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* 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* 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 frequency *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.

LOGLEVEL=*number* | *string*

controls the amount of information that is displayed in the SAS log. Table 2.16 describes the valid values for this option.

Table 2.16 Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turns off all algorithm-related messages in the SAS log
1	BASIC	Displays a basic summary of the algorithmic processing
2	MODERATE	Displays a summary of the algorithmic processing including a progress log using the interval that is specified in the LOGFREQ= option
3	AGGRESSIVE	Displays a detailed summary of the algorithmic processing including a progress log using the interval that is specified in the LOGFREQ= option

The default is the value that is specified in the LOGLEVEL= option in the PROC OPTNET statement (or BASIC if that option is not specified).

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. The value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

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 70.

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

LOGLEVEL=*number* | *string*
controls the amount of information that is displayed in the SAS log. Table 2.17 describes the valid values for this option.

Table 2.17 Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turns off all algorithm-related messages in the SAS log
1	BASIC	Displays a basic summary of the algorithmic processing
2	MODERATE	Displays a summary of the algorithmic processing
3	AGGRESSIVE	Displays a detailed summary of the algorithmic processing

The default is the value that is specified in the LOGLEVEL= option in the PROC OPTNET statement (or BASIC if that option is not specified).

MAXNUMCUTS=*number*
specifies the maximum number of cuts to return from the algorithm. The minimal cut and any others found during the search, up to *number*, are returned. By default, MAXNUMCUTS=1.

MAXWEIGHT=*number*
specifies the maximum weight of the cuts to return from the algorithm. Only cuts that have weight less than or equal to *number* are returned. The default is the positive number that has the largest absolute value that can be represented in your operating environment.

OUT=*SAS-data-set*
specifies the output data set to contain the solution to the minimum-cut problem.

MINSPANTREE Statement

MINSPANTREE < *options* > **;**

MST < *options* > **;**

The MINSPANTREE statement invokes an algorithm that solves the minimum link-weighted spanning tree problem on an input graph.

The minimum spanning tree problem is described in the section "Minimum Spanning Tree" on page 74.

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

LOGLEVEL=*number* | *string*
controls the amount of information that is displayed in the SAS log. Table 2.18 describes the valid values for this option.

Table 2.18 Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turns off all algorithm-related messages in the SAS log
1	BASIC	Displays a basic summary of the algorithmic processing

Table 2.18 (continued)

number	*string*	**Description**
2	MODERATE	Displays a summary of the algorithmic processing
3	AGGRESSIVE	Displays a detailed summary of the algorithmic processing

The default is the value that is specified in the LOGLEVEL= option in the PROC OPTNET statement (or BASIC if that option is not specified).

OUT=*SAS-data-set*
specifies the output data set to contain the solution to the minimum link-weighted spanning tree problem.

PERFORMANCE Statement

PERFORMANCE < *performance-options* > **;**

The PERFORMANCE statement specifies performance options for multithreaded computing and requests detailed results about the performance characteristics of the OPTNET procedure.

The PERFORMANCE statement enables you to control the number of threads used and the output of the ODS table that reports procedure timing. When you specify the PERFORMANCE statement, the PerformanceInfo ODS table is produced. This table lists performance characteristics such as execution mode and number of threads.

You can specify the following *performance-options* in the PERFORMANCE statement:

DETAILS
requests that PROC OPTNET produce the Timing ODS table, which shows a breakdown of the time used in each step of the procedure.

NTHREADS=*number* | **CPUCOUNT**
specifies the number of threads that PROC OPTNET can use. This option overrides the SAS system option THREADS | NOTHREADS. The value of *number* can be any integer between 1 and 256, inclusive. The default value is CPUCOUNT, which sets the thread count to the number determined by the SAS system option CPUCOUNT=.

Setting this option to a number greater than the number of available cores might result in reduced 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 the PROC OPTNET. In some circumstances, the OPTNET procedure might use fewer threads than the specified *number* because the procedure's internal algorithms have determined that a smaller number is preferable.

For example, the following call to PROC OPTNET uses eight threads to read the data input in parallel:

```
proc optnet
   data_links       = LinkSetIn
   graph_direction  = directed
```

```
   out_nodes        = NodeSetOut;
   performance
      nthreads      = 8;
run;
```

SHORTPATH Statement

SHORTPATH < *options* > **;**

The SHORTPATH statement invokes an algorithm that calculates shortest paths between sets of nodes on the input graph.

The shortest path algorithm is described in the section "Shortest Path" on page 76.

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

LOGFREQ=*number*
displays iteration logs for shortest path calculations every *number* nodes. The value of *number* can be any integer greater than or equal to 1. The default is determined automatically based on the size of the graph. Setting this value too low can hurt performance on large-scale graphs.

LOGLEVEL=*number*
controls the amount of information that is displayed in the SAS log. Table 2.19 describes the valid values for this option.

Table 2.19 Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turns off all algorithm-related messages in the SAS log
1	BASIC	Displays a basic summary of the algorithmic processing
2	MODERATE	Displays a summary of the algorithmic processing
3	AGGRESSIVE	Displays a detailed summary of the algorithmic processing

The default is the value that is specified in the LOGLEVEL= option in the PROC OPTNET statement (or BASIC if that option is not specified).

OUT_PATHS=*SAS-data-set*
OUT=*SAS-data-set*
specifies the output data set to contain the shortest paths.

OUT_WEIGHTS=*SAS-data-set*
specifies the output data set to contain the shortest path summaries.

PATHS=ALL | LONGEST | SHORTEST
specifies the type of output to produce in the output data set that is specified in the OUT_PATHS= option.

Table 2.20 Values for the PATHS= Option

Option Value	Description
ALL	Outputs shortest paths for all pairs of source-sinks.
LONGEST	Outputs shortest paths for the source-sink pair with the longest (finite) length. If other source-sink pairs (up to 100) have equally long length, they are also output.
SHORTEST	Outputs shortest paths for the source-sink pair with the shortest length. If other source-sink pairs (up to 100) have equally short length, they are also output.

By default, PATHS=ALL.

SINK=*sink-node*
specifies the sink node for shortest paths calculations. This setting overrides the use of the variable sink in the data set that is specified in the DATA_NODES_SUB= option in the PROC OPTNET statement.

SOURCE=*source-node*
specifies the source node for shortest paths calculations. This setting overrides the use of the variable source in the data set that is specified in the DATA_NODES_SUB= option in the PROC OPTNET statement.

USEWEIGHT=YES | NO
specifies whether to use link weights (if they exist) in calculating shortest paths.

Table 2.21 Values for the WEIGHT= Option

Option Value	Description
YES	Uses weights (if they exist) in shortest path calculations. This is the default.
NO	Does not use weights in shortest path calculations.

WEIGHT2=*column*
specifies the data set variable name for the auxiliary link weights. The value of the *column* variable must be numeric.

TRANSITIVE_CLOSURE Statement

TRANSITIVE_CLOSURE < *options* > **;**

TRANSCL < *options* > **;**

The TRANSITIVE_CLOSURE statement invokes an algorithm that calculates the transitive closure of an input graph.

Transitive closure is described in the section "Transitive Closure" on page 87.

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

LOGLEVEL=*number*

controls the amount of information that is displayed in the SAS log. Table 2.22 describes the valid values for this option.

Table 2.22 Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turns off all algorithm-related messages in the SAS log
1	BASIC	Displays a basic summary of the algorithmic processing
2	MODERATE	Displays a summary of the algorithmic processing
3	AGGRESSIVE	Displays a detailed summary of the algorithmic processing

The default is the value that is specified in the LOGLEVEL= option in the PROC OPTNET statement (or BASIC if that option is not specified).

OUT=*SAS-data-set*

specifies the output data set to contain the transitive closure results.

TSP Statement

TSP < *options* > **;**

The TSP statement invokes an algorithm that solves the traveling salesman problem.

The traveling salesman problem is described in the section "Traveling Salesman Problem" on page 89. The algorithm that is used to solve this problem is built around the same method as is used in PROC OPTMILP: a branch-and-cut algorithm. Many of the following options are the same as those described for the OPTMILP procedure in the *SAS/OR User's Guide: Mathematical Programming*.

You can specify the following *options*:

ABSOBJGAP=*number*

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.

CONFLICTSEARCH=*number* | *string*

specifies the level of conflict search that PROC OPTNET performs. The solver performs a conflict search to find clauses that result from infeasible subproblems that arise in the search tree. Table 2.23 describes the valid values for this option.

Table 2.23 Values for CONFLICTSEARCH= Option

number	*string*	**Description**
–1	AUTOMATIC	Performs a conflict search based on a strategy that is determined by PROC OPTNET
0	NONE	Disables conflict search
1	MODERATE	Performs a moderate conflict search
2	AGGRESSIVE	Performs an aggressive conflict search

By default, CONFLICTSEARCH=AUTOMATIC.

CUTOFF=*number*

cuts off any branch-and-bound nodes in a minimization problem that has an objective value that is greater than *number*. The value of *number* can be any number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

CUTSTRATEGY=*number* | *string*

specifies the level of mixed integer linear programming cutting planes to be generated by PROC OPTNET. TSP-specific cutting planes are always generated. Table 2.24 describes the valid values for this option.

Table 2.24 Values for CUTSTRATEGY= Option

number	*string*	**Description**
–1	AUTOMATIC	Generates cutting planes based on a strategy determined by the mixed integer linear programming solver
0	NONE	Disables generation of mixed integer linear programming cutting planes (some TSP-specific cutting planes are still active for validity)
1	MODERATE	Uses a moderate cut strategy
2	AGGRESSIVE	Uses an aggressive cut strategy

By default, CUTSTRATEGY=NONE.

EMPHASIS=*number* | *string*

specifies a search emphasis *option*. Table 2.25 describes the valid values for this option.

Table 2.25 Values for EMPHASIS= Option

number	*string*	**Description**
0	BALANCE	Performs a balanced search
1	OPTIMAL	Emphasizes optimality over feasibility
2	FEASIBLE	Emphasizes feasibility over optimality

By default, EMPHASIS=BALANCE.

HEURISTICS=*number* | *string*

controls the level of initial and primal heuristics that PROC OPTNET applies. This level determines how frequently PROC OPTNET applies primal heuristics during the branch-and-bound tree search. It also affects the maximum number of iterations that are allowed in iterative heuristics. Some computationally expensive heuristics might be disabled by the solver at less aggressive levels. Table 2.26 lists the valid values for this option.

Table 2.26 Values for HEURISTICS= Option

number	*string*	**Description**
–1	AUTOMATIC	Applies the default level of heuristics
0	NONE	Disables all initial and primal heuristics

Table 2.26 (continued)

number	*string*	**Description**
1	BASIC	Applies basic initial and primal heuristics at low frequency
2	MODERATE	Applies most initial and primal heuristics at moderate frequency
3	AGGRESSIVE	Applies all initial primal heuristics at high frequency

By default, HEURISTICS=AUTOMATIC.

LOGFREQ=*number*

specifies how often to print information in the branch-and-bound node log. The value of *number* can be any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value is 100. If *number* is set to 0, then the node log is disabled. If *number* is positive, then an entry is made in the node log at the first node, at the last node, and at intervals that are controlled by the value of *number*. An entry is also made each time a better integer solution is found.

LOGLEVEL=*number* | *string*

controls the amount of information displayed in the SAS log by the solver, from a short description of presolve information and summary to details at each branch-and-bound node. Table 2.27 describes the valid values for this option.

Table 2.27 Values for LOGLEVEL= Option

number	*string*	**Description**
0	NONE	Turns off all solver-related messages in the SAS log
1	BASIC	Displays a solver summary after stopping
2	MODERATE	Prints a solver summary and a node log by using the interval that is specified in the LOGFREQ= option
3	AGGRESSIVE	Prints a detailed solver summary and a node log by using the interval that is specified in the LOGFREQ= option

The default value is MODERATE.

MAXNODES=*number*

specifies the maximum number of branch-and-bound nodes to be processed. The value of *number* can be any nonnegative integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value is $2^{31} - 1$.

MAXSOLS=*number*

specifies a stopping criterion. If *number* solutions have been found, then the procedure stops. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value is $2^{31} - 1$.

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. The value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

MILP=*number* | *string*
specifies whether to use a mixed integer linear programming (MILP) solver for solving the traveling salesman problem. The MILP solver attempts to find the overall best TSP tour by using a branch-and-bound based algorithm. This algorithm can be expensive for large-scale problems. If MILP=OFF, then PROC OPTNET uses its initial heuristics to find a feasible, but not necessarily optimal, tour as quickly as possible. Table 2.28 describes the valid values for this option.

Table 2.28 Values for MILP= Option

number	*string*	**Description**
1	ON	Uses a mixed integer linear programming solver
0	OFF	Does not use a mixed integer linear programming solver

By default, MILP=ON.

NODESEL=*number* | *string*
specifies the branch-and-bound node selection strategy option. Table 2.29 describes the valid values for this option.

Table 2.29 Values for NODESEL= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses automatic node selection
0	BESTBOUND	Chooses the node that has the best relaxed objective (best-bound-first strategy)
1	BESTESTIMATE	Chooses the node that has the best estimate of the integer objective value (best-estimate-first strategy)
2	DEPTH	Chooses the most recently created node (depth-first strategy)

By default, NODESEL=AUTOMATIC. For more information about node selection, see Chapter 14, "The OPTMILP Procedure" (*SAS/OR User's Guide: Mathematical Programming*).

OUT=*SAS-data-set*
specifies the output data set to contain the solution to the traveling salesman problem.

PROBE=*number* | *string*
specifies a probing *option*. Table 2.30 describes the valid values for this option.

Table 2.30 Values for PROBE= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses an automatic probing strategy
0	NONE	Disables probing
1	MODERATE	Uses the probing moderately
2	AGGRESSIVE	Uses the probing aggressively

By default, PROBE=NONE.

RELOBJGAP=*number*

specifies a stopping criterion that is based on the best integer objective (BestInteger) and the objective of the best remaining node (BestBound). The relative objective gap is equal to

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

When this value becomes less than the specified gap size *number*, the solver stops. The value of *number* can be any nonnegative number. By default, RELOBJGAP=1E–4.

STRONGITER=*number* | **AUTOMATIC**

specifies the number of simplex iterations that PROC OPTNET performs for each variable in the candidate list when it uses the strong branching variable selection strategy. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC or the value –1, PROC OPTNET uses the default value, which it calculates automatically.

STRONGLEN=*number* | **AUTOMATIC**

specifies the number of candidates that PROC OPTNET considers when it uses the strong branching variable selection strategy. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. If you specify the keyword AUTOMATIC or the value –1, PROC OPTNET uses the default value, which it calculates automatically.

TARGET=*number*

specifies a stopping criterion for minimization problems. If the best integer objective is better than or equal to *number*, the solver stops. The value of *number* can be any number; the default is the negative number that has the largest absolute value that can be represented in your operating environment.

VARSEL=*number* | *string*

specifies the rule for selecting the branching variable. Table 2.31 describes the valid values for this option.

Table 2.31 Values for VARSEL= Option

number	*string*	**Description**
–1	AUTOMATIC	Uses automatic branching variable selection
0	MAXINFEAS	Chooses the variable that has maximum infeasibility
1	MININFEAS	Chooses the variable that has minimum infeasibility
2	PSEUDO	Chooses a branching variable based on pseudocost
3	STRONG	Uses the strong branching variable selection strategy

By default, VARSEL=AUTOMATIC. For more information about variable selection, see Chapter 14, "The OPTMILP Procedure" (*SAS/OR User's Guide: Mathematical Programming*).

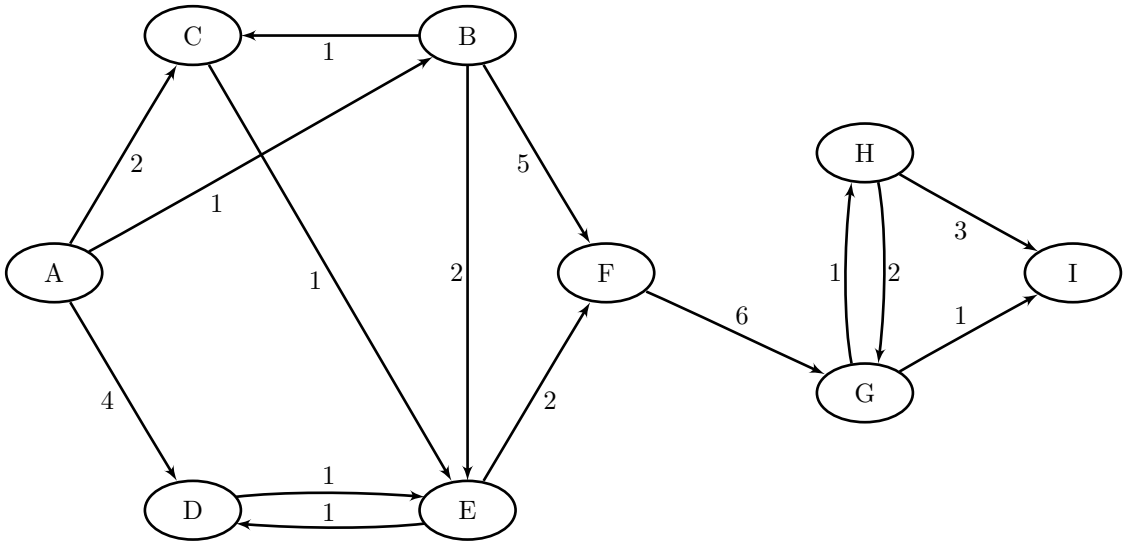
Details: OPTNET Procedure

Graph Input Data

This section describes how to input a graph for analysis by PROC OPTNET. Let $G = (N, A)$ define a graph with a set N of nodes and a set A of links.

Consider the directed graph shown in Figure 2.5.

Figure 2.5 A Simple Directed Graph



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

Link Input Data

The DATA_LINKS= option in the PROC OPTNET statement defines the data set 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 set is expected to contain some combination of the following possible variables:

- from: the *from* node (this variable can be numeric or character)
- to: the *to* node (this variable can be numeric or character)
- weight: the link weight (this variable must be numeric)
- weight2: the auxiliary link weight (this variable must be numeric)
- lower: the link flow lower bound (this variable must be numeric)
- upper: the link flow upper bound (this variable must be numeric)

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

The data set variable names can have any values that you want. If you use nonstandard names, you must identify the variables by using the DATA_LINKS_VAR statement, as described in the section "DATA_LINKS_VAR Statement" on page 21.

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

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

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

These data sets can be presented to PROC OPTNET by using the following equivalent statements:

```
proc optnet
   data_links = LinkSetInA;
run;

proc optnet
   data_links = LinkSetInB;
   data_links_var
      from    = source_node
      to      = sink_node
      weight  = value;
run;
```

The directed graph *G* shown in Figure 2.5 can be represented by the following links data set LinkSetIn:

```
data LinkSetIn;
   input from $ to $ weight @@;
   datalines;
A B 1  A C 2  A D 4  B C 1  B E 2
B F 5  C E 1  D E 1  E D 1  E F 2
F G 6  G H 1  G I 1  H G 2  H I 3
;
```

The following statements read in this graph, declare it as a directed graph, and output the resulting links and nodes data sets. These statements do not run any algorithms, so the resulting output contains only the input graph.

```
proc optnet
   graph_direction = directed
   data_links      = LinkSetIn
   out_nodes       = NodeSetOut
```

```
   out_links       = LinkSetOut;
run;
```

The data set NodeSetOut, shown in Figure 2.6, now contains the nodes that are read from the input link data set. The variable node shows the label associated with each node.

Figure 2.6 Node Data Set of a Simple Directed Graph

node
A
B
C
D
E
F
G
H
I

The data set LinkSetOut, shown in Figure 2.7, contains the links that were read from the input link data set. The variables from and to show the associated node labels.

Figure 2.7 Link Data Set of a Simple Directed Graph

Obs	from	to	weight
1	A	B	1
2	A	C	2
3	A	D	4
4	B	C	1
5	B	E	2
6	B	F	5
7	C	E	1
8	D	E	1
9	E	D	1
10	E	F	2
11	F	G	6
12	G	H	1
13	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 duplicates are removed. PROC OPTNET takes the first occurrence of the link and ignores the others. By default, GRAPH_DIRECTION=UNDIRECTED, so you can just remove this option to declare the graph as undirected.

```
proc optnet
   data_links = LinkSetIn
   out_nodes  = NodeSetOut
   out_links  = LinkSetOut;
run;
```

The progress of the procedure is shown in Figure 2.8. The log now shows the links (and their observation identifiers) that were declared as duplicates and removed.

Figure 2.8 PROC OPTNET Log: Link Data Set of a Simple Undirected Graph

```
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Running OPTNET version 15.1.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The OPTNET procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
WARNING: Link (E,D) in observation 9 of the DATA_LINKS= data set is a duplicate and is ignored.
WARNING: Link (H,G) in observation 14 of the DATA_LINKS= data set is a duplicate and is ignored.
NOTE: The number of nodes in the input graph is 9.
NOTE: The number of links in the input graph is 13.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The data set WORK.NODESETOUT has 9 observations and 1 variables.
NOTE: The data set WORK.LINKSETOUT has 13 observations and 3 variables.
```

The data set NodeSetOut is equivalent to the one shown in Figure 2.6. However, the new links data set LinkSetOut shown in Figure 2.9 contains two fewer links than before, because duplicates are removed.

Figure 2.9 Link Data Set of a Simple Undirected Graph

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

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

Node Input Data

The DATA_NODES= option in the PROC OPTNET statement defines the data set that contains the list of nodes in the graph. This data set is used to assign node weights.

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

- node: the node label (this variable can be numeric or character)
- weight: the node weight (this variable must be numeric)
- weight2: the auxiliary node weight (this variable must be numeric)

You can specify any values that you want for the data set variable names. If you use nonstandard names, you must identify the variables by using the DATA_NODES_VAR statement, as described in the section "DATA_NODES_VAR Statement" on page 22.

The data set that is specified in the DATA_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 this node must be defined in the DATA_NODES data set. The following is an example of a graph with three links but four nodes, including a singleton node D:

```
data NodeSetIn;
   input label $ @@;
   datalines;
A B C D
;

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

If you specify duplicate entries in the node data set, PROC OPTNET takes the first occurrence of the node and ignores the others. A warning is printed to the log.

Node 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 DATA_NODES_SUB= option in the PROC OPTNET statement. You can use the node subset data set in conjunction with the SHORTPATH statement (see the section "Shortest Path" on page 76. The node subset data set is expected to contain some combination of the following variables:

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

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

A representative example of a node subset data set that might be used with the graph in Figure 2.5 is as follows:

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

The data set NodeSubSetIn indicates that you want to process the shortest paths for the source-sink pairs in $\{A, E\} \times \{F\}$.

Standardized Labels

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

1. The link data set variables from and to are numeric type.
2. The node and node subset data set variable node is numeric type.
3. The node labels start from 0 and are consecutive nonnegative integers.

Consider the following links data set that uses numeric labels:

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

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

```
proc optnet
   data_links = LinkSetIn
   out_nodes  = NodeSetOut
   out_links  = LinkSetOut;
run;
```

The log is shown in Figure 2.10.

Figure 2.10 PROC OPTNET Log: A Simple Undirected Graph

```
NOTE: ---------------------------------------------------------------------------------------
NOTE: Running OPTNET version 15.1.
NOTE: ---------------------------------------------------------------------------------------
NOTE: The OPTNET procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------------------
NOTE: Data input 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 3.
NOTE: ---------------------------------------------------------------------------------------
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------
NOTE: The data set WORK.NODESETOUT has 4 observations and 1 variables.
NOTE: The data set WORK.LINKSETOUT has 3 observations and 3 variables.
```

The data set NodeSetOut, shown in Figure 2.11, contains the unique numeric node labels, $\{0, 1, 3, 5\}$.

Figure 2.11 Node Data Set of a Simple Directed Graph

Obs	node
1	0
2	1
3	3
4	5

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

```
proc optnet
   standardized_labels
   data_links = LinkSetIn
   out_nodes  = NodeSetOut
   out_links  = LinkSetOut;
run;
```

The log that results from using standardized labels is shown in Figure 2.12.

Figure 2.12 PROC OPTNET Log: A Simple Undirected Graph Using Standardized Labels

```
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Running OPTNET version 15.1.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The OPTNET procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 6.
NOTE: The number of links in the input graph is 3.
NOTE: The number of singleton nodes in the input graph is 2.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The data set WORK.NODESETOUT has 6 observations and 1 variables.
NOTE: The data set WORK.LINKSETOUT has 3 observations and 3 variables.
```

The data set NodeSetOut, shown in Figure 2.13, now contains all node labels from 0 to 5, based on the assumptions when you use the STANDARDIZED_LABELS option.

Figure 2.13 Node Data Set of a Simple Directed Graph

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

When you use standardized labels, the DATA_NODES= input order (which can be arbitrary) is not preserved in the OUT_NODES= output data set. Instead, the order is ascending, starting from zero.

Matrix Input Data

This section describes the matrix input format that you can use with some of the algorithms in PROC OPTNET. The DATA_MATRIX= option in the PROC OPTNET statement defines the data set that contains the matrix values. You can specify any values that you want for the data set variable names (the columns) by using the DATA_MATRIX_VAR statement, as described in the section "DATA_MATRIX_VAR Statement" on page 22. If you do not specify a DATA_MATRIX_VAR statement, then PROC OPTNET assumes that all numeric variables in the data set are to be used in defining the matrix.

The following statements solve the linear assignment problem for the cost matrix that is defined in the data set CostMatrix:

```
data CostMatrix;
   input back breast fly free;
   datalines;
35.1 36.7 28.3 36.1
34.6 32.6 26.9 26.2
31.3 33.9 27.1 31.2
28.6 34.1 29.1 30.3
32.9 32.2 26.6 24.0
27.8 32.5 27.8 27.0
26.3 27.6 23.5 22.4
29.0 24.0 27.9 25.4
27.2 33.8 25.2 24.1
27.0 29.2 23.0 21.9
;

proc optnet
   data_matrix = CostMatrix;
   data_matrix_var
      back--free;
   linear_assignment
      out      = LinearAssign;
run;
```

Data Input Order

Many algorithms are sensitive to the order in which PROC OPTNET reads the data. If the order of the nodes or links is changed, either by you or by some parameter setting, the final result might change. 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. Two parameters that could have such an effect are the STANDARDIZED_LABELS and NTHREADS= options. Both of these options can change the internal order of the nodes and links.

Parallel Processing

PROC OPTNET can take advantage of multicore chip technology by processing the graph input data in parallel. To enable PROC OPTNET to process in parallel, you can use the NTHREADS= option in the PERFORMANCE statement to specify the number of threads to use.

Numeric Limitations

Extremely large or extremely small numerical values might cause computational difficulties for some of the algorithms in PROC OPTNET. 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 OPTNET 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 2.32, where M is defined as the largest absolute value representable in your operating environment.

Table 2.32 Threshold Limits by Statement

	Matrix	Graph Links				Graph Nodes	
Statement		weight	weight2	lower	upper	weight	weight2
CYCLE		$\sqrt{M}$				$\sqrt{M}$	
LINEAR_ASSIGNMENT	$\sqrt{M}$	$\sqrt{M}$					
MINCOSTFLOW		1e15		1e15	1e15	1e15	1e15
MINCUT		$\sqrt{M}$					
MINSPANTREE		$\sqrt{M}$					
SHORTPATH		$\sqrt{M}$	$\sqrt{M}$				
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 OPTNET, there is no valid interpretation for a missing value. If the user data contain a missing value, PROC OPTNET 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 62. Another exception is the linear assignment problem when you are using the matrix input format. A missing value in this case defines an invalid assignment between a row and a column of the matrix. An example of this is shown in the section "Linear Assignment (Matching)" on page 61.

Negative Link Weights

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

- MINCUT

Size Limitations

PROC OPTNET can handle any graph whose number of nodes and links are less than or equal to 2,147,483,647 (the maximum that can be represented by a 32-bit integer). This maximum also applies to 64-bit systems. For graphs of two billion nodes (or links), memory limitations also become a limiting factor.

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

Biconnected Components and Articulation Points

A *biconnected component* of a graph $G = (N, A)$ is a connected subgraph that cannot be broken into disconnected pieces by deleting any single node (and its incident links). An *articulation point* is a node of a graph whose removal would cause an increase in the number of connected components. Articulation points can be important when you analyze any graph that represents a communications network. Consider an articulation point $i \in N$ which, if removed, disconnects the graph into two components C^1 and C^2. All paths in G between some nodes in C^1 and some nodes in C^2 must pass through node i. In this sense, articulation points are critical to communication. Examples of where articulation points are important are airline hubs, electric circuits, network wires, protein bonds, traffic routers, and numerous other industrial applications.

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

The results of the biconnected components algorithm are written to the output links data set that is specified in the OUT_LINKS= option in the PROC OPTNET statement. For each link in the links data set, the variable biconcomp identifies its component. The component identifiers are numbered sequentially starting from 1. The results of the articulation points are written to the output nodes data set that is specified in the OUT_NODES= option in the PROC OPTNET statement. For each node in the nodes data set, the variable artpoint is either 1 (if the node is an articulation point) or 0 (otherwise).

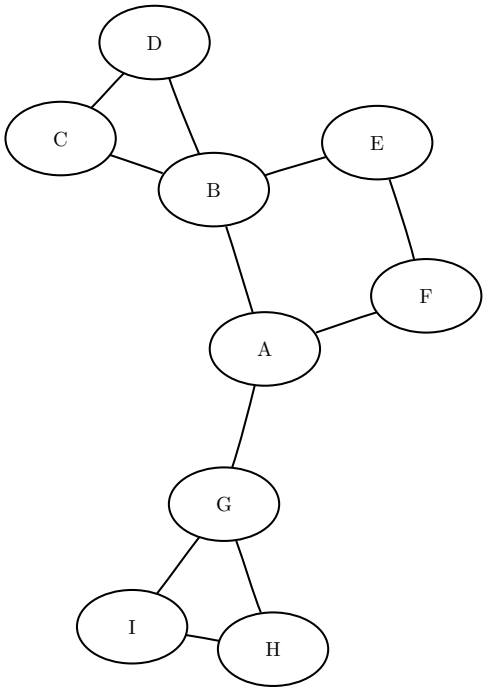
The biconnected components algorithm reports status information in a macro variable called _OROPTNET_BICONCOMP_. For more information about this macro variable see the section "Macro Variable _OROPTNET_BICONCOMP_" on page 100.

The algorithm that PROC OPTNET uses to compute biconnected components is a variant of depth-first search (Tarjan 1972). This algorithm runs in time $O(|N| + |A|)$ and therefore should scale to very large graphs.

Biconnected Components of a Simple Undirected Graph

This section illustrates the use of the biconnected components algorithm on the simple undirected graph G that is shown in Figure 2.14.

Figure 2.14 A Simple Undirected Graph G



The undirected graph G can be represented by the following links data set LinkSetInBiCC:

```
data LinkSetInBiCC;
   input from $ to $ @@;
   datalines;
A B  A F  A G  B C  B D
B E  C D  E F  G I  G H
H I
;
```

The following statements calculate the biconnected components and articulation points and output the results in the data sets LinkSetOut and NodeSetOut:

```
proc optnet
   data_links = LinkSetInBiCC
   out_links  = LinkSetOut
   out_nodes  = NodeSetOut;
   biconcomp;
run;
```

The data set LinkSetOut now contains the biconnected components of the input graph, as shown in Figure 2.15.

Figure 2.15 Biconnected Components of a Simple Undirected Graph

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

In addition, the data set NodeSetOut contains the articulation points of the input graph, as shown in Figure 2.16.

Figure 2.16 Articulation Points of a Simple Undirected Graph

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

The biconnected components are shown graphically in Figure 2.17 and Figure 2.18.

Figure 2.17 Biconnected Components C^1 and C^2

$C^1 = \{B, C, D\}$ $C^2 = \{A, B, E, F\}$

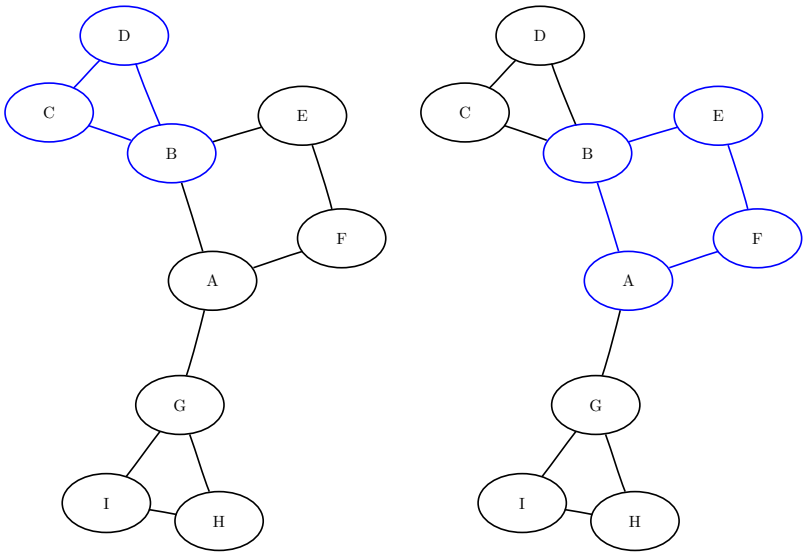
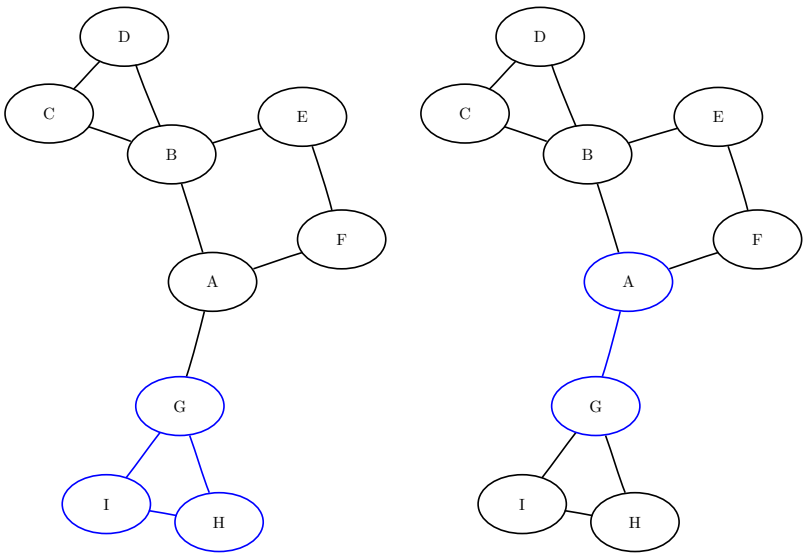


Figure 2.18 Biconnected Components C^3 and C^4

$C^3 = \{G, H, I\}$ $C^4 = \{A, G\}$



For a more detailed example, see "Example 2.1: Articulation Points in a Terrorist Network" on page 107.

Clique

A *clique* of a graph $G = (N, A)$ is an induced subgraph that is a complete graph. Every node in a clique is connected to every other node in that clique. A *maximal clique* is a clique that is not a subset of the nodes of

any larger clique. That is, it is a set *C* of nodes such that every pair of nodes in *C* is connected by a link and every node not in *C* is missing a link to at least one node in *C*. The number of maximal cliques in a particular graph can be very large and can grow exponentially with every node added. Finding cliques in graphs has applications in numerous industries including bioinformatics, social networks, electrical engineering, and chemistry.

You can find the maximal cliques of an input graph by invoking the CLIQUE statement. The options for this statement are described in the section "CLIQUE Statement" on page 17. The clique algorithm works only with undirected graphs.

The results of the clique algorithm are written to the output data set that is specified in the OUT= option in the CLIQUE statement. Each node of each clique is listed in the output data set along with the variable clique to identify the clique to which it belongs. A node can appear multiple times in this data set if it belongs to multiple cliques.

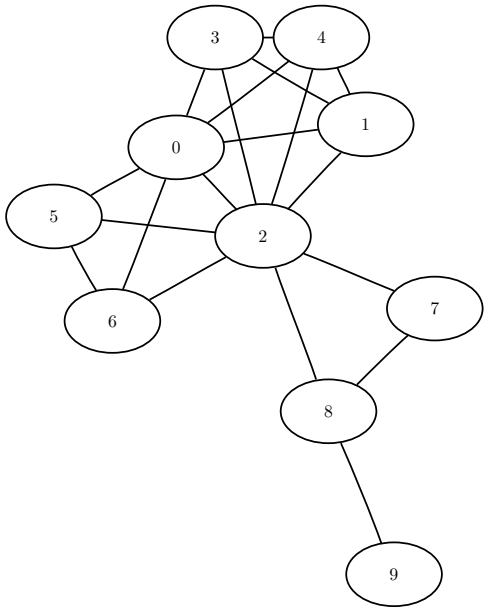
The clique algorithm reports status information in a macro variable called _OROPTNET_CLIQUE_. For more information about this macro variable, see the section "Macro Variable _OROPTNET_CLIQUE_" on page 101.

The algorithm that PROC OPTNET uses to compute maximal cliques is a variant of the Bron-Kerbosch algorithm (Bron and Kerbosch 1973; Harley 2003). Enumerating all maximal cliques is NP-hard, so this algorithm typically does not scale to very large graphs.

Maximal Cliques of a Simple Undirected Graph

This section illustrates the use of the clique algorithm on the simple undirected graph *G* that is shown in Figure 2.19.

Figure 2.19 A Simple Undirected Graph *G*



The undirected graph *G* can be represented by the following links data set LinkSetIn:

```
data LinkSetIn;
   input from to @@;
   datalines;
0 1  0 2  0 3  0 4  0 5
0 6  1 2  1 3  1 4  2 3
2 4  2 5  2 6  2 7  2 8
3 4  5 6  7 8  8 9
;
```

The following statements calculate the maximal cliques, output the results in the data set Cliques, and use the SQL procedure as a convenient way to create a table CliqueSizes of clique sizes:

```
proc optnet
   data_links = LinkSetIn;
   clique
      out     = Cliques;
run;

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

The data set Cliques now contains the maximal cliques of the input graph; it is shown in Figure 2.20.

Figure 2.20 Maximal Cliques of a Simple Undirected Graph

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

In addition, the data set CliqueSizes contains the number of nodes in each clique; it is shown in Figure 2.21.

Figure 2.21 Sizes of Maximal Cliques of a Simple Undirected Graph

clique	size
1	5
2	4
3	3
4	2

The maximal cliques are shown graphically in Figure 2.22 and Figure 2.23.

Figure 2.22 Maximal Cliques C^1 and C^2

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

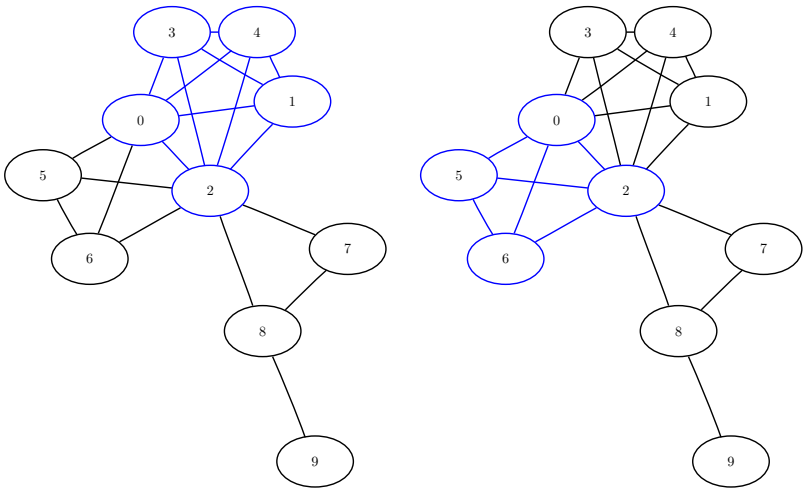
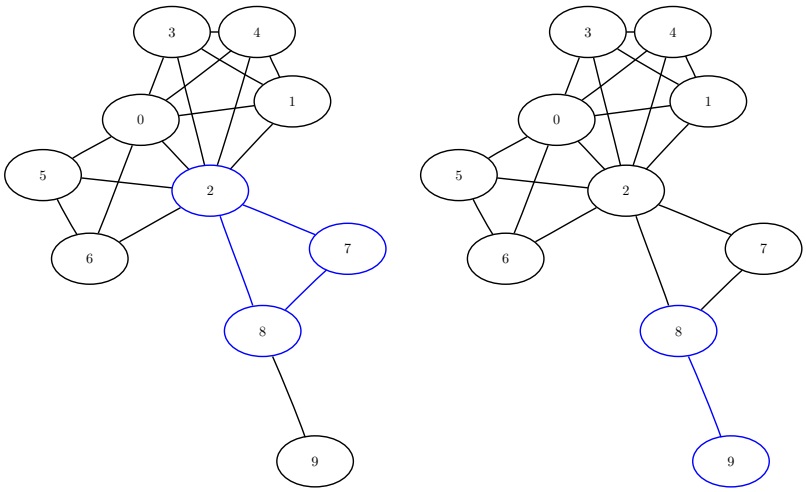


Figure 2.23 Maximal Cliques C^3 and C^4

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



Connected Components

A *connected component* of a graph is a set of nodes that are all reachable from each other. That is, if two nodes are in the same component, then there exists a path between them. For a directed graph, there are two types of components: a *strongly connected component* has a directed path between any two nodes, and a *weakly connected component* ignores direction and requires only that a path exist between any two nodes.

In PROC OPTNET, you can invoke connected components by using the CONCOMP statement. The options for this statement are described in the section "CONCOMP Statement" on page 18.

There are two main algorithms for finding connected components in an undirected graph: a depth-first search algorithm (ALGORITHM=DFS) and a union-find algorithm (ALGORITHM=UNION_FIND). For a graph $G = (N, A)$, both algorithms run in time $O(|N| + |A|)$ and can usually scale to very large graphs. The default is the union-find algorithm. For directed graphs, only the depth-first search algorithm is available.

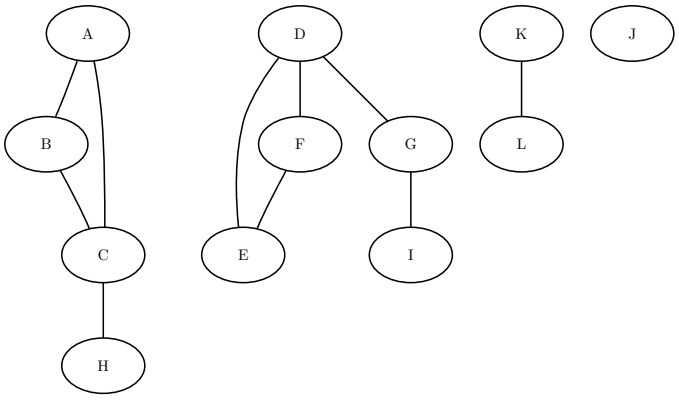
The results of the connected components algorithm are written to the output node data set that you specify in the OUT_NODES= option in the PROC OPTNET statement. For each node in the node data set, the variable concomp identifies its component. The component identifiers are numbered sequentially starting from 1.

The connected components algorithm reports status information in a macro variable called _OROPTNET_CONCOMP_. For more information about this macro variable, see the section "Macro Variable _OROPTNET_CONCOMP_" on page 101.

Connected Components of a Simple Undirected Graph

This section illustrates the use of the connected components algorithm on the simple undirected graph G that is shown in Figure 2.24.

Figure 2.24 A Simple Undirected Graph G



The undirected graph G can be represented by the following links data set, LinkSetIn:

```
data LinkSetIn;
   input from $ to $ @@;
   datalines;
A B  A C  B C  C H  D E  D F  D G  F E  G I  K L
;
```

The following statements calculate the connected components and output the results in the data set NodeSetOut:

```
proc optnet
   data_links = LinkSetIn
   out_nodes  = NodeSetOut;
   concomp;
run;
```

The data set NodeSetOut contains the connected components of the input graph and is shown in Figure 2.25.

Figure 2.25 Connected Components of a Simple Undirected Graph

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

Notice that the graph is defined by using only the links data set. As seen in Figure 2.24, this graph also contains a singleton node labeled J, which has no associated links. By definition, this node defines its own component. But because the input graph was defined by using only the links data set, it did not show up in the results data set. To define a graph by using nodes that have no associated links, you should also define the input nodes data set. In this case, define the nodes data set NodeSetIn as follows:

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

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

```
proc optnet
   data_nodes = NodeSetIn
   data_links = LinkSetIn
   out_nodes  = NodeSetOut;
   concomp;
run;
```

The resulting data set, NodeSetOut, includes the singleton node J as its own component, as shown in Figure 2.26.

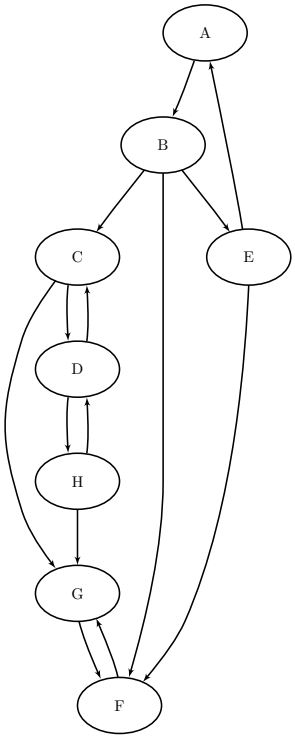
Figure 2.26 Connected Components of a Simple Undirected Graph

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

Connected Components of a Simple Directed Graph

This section illustrates the use of the connected components algorithm on the simple directed graph G that is shown in Figure 2.27.

Figure 2.27 A Simple Directed Graph G



The directed graph G can be represented by the following links data set, LinkSetIn:

```
data LinkSetIn;
   input from $ to $ @@;
   datalines;
A B  B C  B E  B F  C G
C D  D C  D H  E A  E F
F G  G F  H G  H D
;
```

The following statements calculate the connected components and output the results in the data set NodeSetOut:

```
proc optnet
   graph_direction = directed
   data_links      = LinkSetIn
   out_nodes       = NodeSetOut;
   concomp;
run;
```

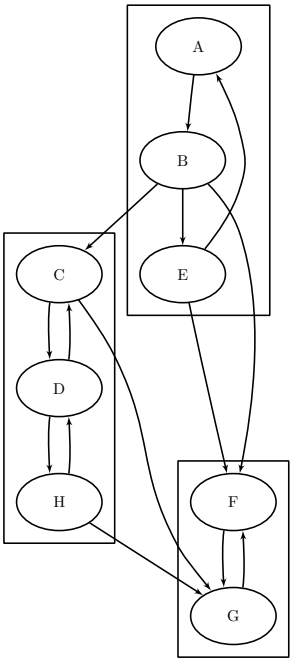
The data set NodeSetOut, shown in Figure 2.28, now contains the connected components of the input graph.

Figure 2.28 Connected Components of a Simple Directed Graph

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

The connected components are represented graphically in Figure 2.29.

Figure 2.29 Strongly Connected Components of Graph G



Cycle

A *path* in a graph is a sequence of nodes, each of which has a link to the next node in the sequence. An *elementary cycle* is a path in which the start node and end node are the same and otherwise no node appears more than once in the sequence.

In PROC OPTNET, you can find (or just count) the elementary cycles of an input graph by invoking the CYCLE statement. The options for this statement are described in the section "CYCLE Statement" on page 19. To find the cycles and report them in an output data set, use the OUT= option. To simply count the cycles, do not use the OUT= option.

For undirected graphs, each link represents two directed links. For this reason, the following cycles are filtered out: trivial cycles ($A \rightarrow B \rightarrow A$) and duplicate cycles that are found by traversing a cycle in both directions ($A \rightarrow B \rightarrow C \rightarrow A$ and $A \rightarrow C \rightarrow B \rightarrow A$).

The results of the cycle detection algorithm are written to the output data set that you specify in the OUT= option in the CYCLE statement. Each node of each cycle is listed in the OUT= data set along with the variable cycle to identify the cycle to which it belongs. The variable order defines the order (sequence) of the node in the cycle.

The cycle detection algorithm reports status information in a macro variable called _OROPTNET_CYCLE_. For more information about this macro variable, see the section "Macro Variable _OROPTNET_CYCLE_" on page 102.

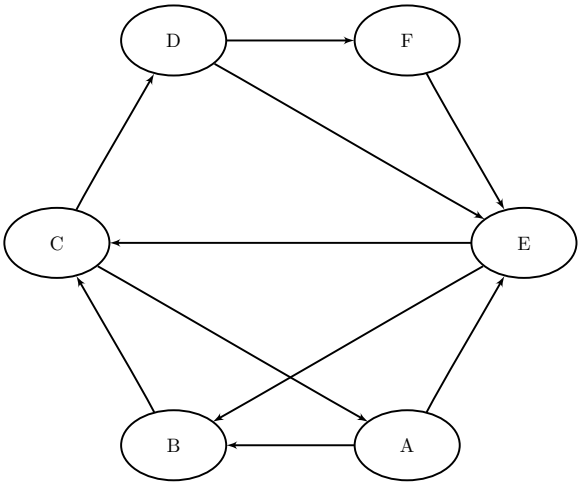
The algorithm that PROC OPTNET uses to compute all cycles is a variant of the algorithm in Johnson (1975). This algorithm runs in time $O((|N| + |A|)(c + 1))$, where c is the number of elementary cycles in the graph. So the algorithm should scale to large graphs that contain few cycles. However, some graphs can have a very large number of cycles, so the algorithm might not scale.

If MODE=ALL_CYCLES and there are many cycles, the OUT= data set can become very large. It might be beneficial to check the number of cycles before you try to create the OUT= data set. When you specify MODE=FIRST_CYCLE, the algorithm returns the first cycle that it finds and stops processing. This should run relatively quickly. For large-scale graphs, the MINLINKWEIGHT= and MAXLINKWEIGHT= options might increase the computation time. For more information about these options, see the section "CYCLE Statement" on page 19.

Cycle Detection of a Simple Directed Graph

This section provides a simple example of using the cycle detection algorithm on the simple directed graph *G* that is shown in Figure 2.30. Two other examples are "Example 2.2: Cycle Detection for Kidney Donor Exchange" on page 109, which shows the use of cycle detection for optimizing a kidney donor exchange, and "Example 2.6: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System" on page 122, which shows another application of cycle detection.

Figure 2.30 A Simple Directed Graph *G*



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

```
data LinkSetIn;
   input from $ to $ @@;
   datalines;
A B  A E  B C  C A  C D
D E  D F  E B  E C  F E
;
```

The following statements check whether the graph has a cycle:

```
proc optnet
   graph_direction = directed
   data_links      = LinkSetIn;
   cycle
      mode         = first_cycle;
run;
%put &_OROPTNET_;
%put &_OROPTNET_CYCLE_;
```

The result is written to the log of the OPTNET procedure, as shown in Figure 2.31.

Figure 2.31 PROC OPTNET Log: Check the Existence of a Cycle in a Simple Directed Graph

```
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Running OPTNET version 15.1.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The OPTNET procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 6.
NOTE: The number of links in the input graph is 10.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Processing cycle detection.
NOTE: The graph does have a cycle.
NOTE: Processing cycle detection used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------------
STATUS=OK  CYCLE=OK
STATUS=OK  NUM_CYCLES=1  CPU_TIME=0.00  REAL_TIME=0.00
```

The following statements count the number of cycles in the graph:

```
proc optnet
   graph_direction = directed
   data_links      = LinkSetIn;
   cycle
      mode         = all_cycles;
run;
%put &_OROPTNET_;
%put &_OROPTNET_CYCLE_;
```

The result is written to the log of the OPTNET procedure, as shown in Figure 2.32.

Figure 2.32 PROC OPTNET Log: Count the Number of Cycles in a Simple Directed Graph

```
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Running OPTNET version 15.1.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The OPTNET procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Data input used 0.00 (cpu: 0.01) seconds.
NOTE: The number of nodes in the input graph is 6.
NOTE: The number of links in the input graph is 10.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Processing cycle detection.
NOTE: The graph has 7 cycles.
NOTE: Processing cycle detection used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------------
STATUS=OK  CYCLE=OK
STATUS=OK  NUM_CYCLES=7  CPU_TIME=0.00  REAL_TIME=0.00
```

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

```
proc optnet
   graph_direction = directed
   data_links      = LinkSetIn;
   cycle
      out          = Cycles
      mode         = first_cycle;
run;
```

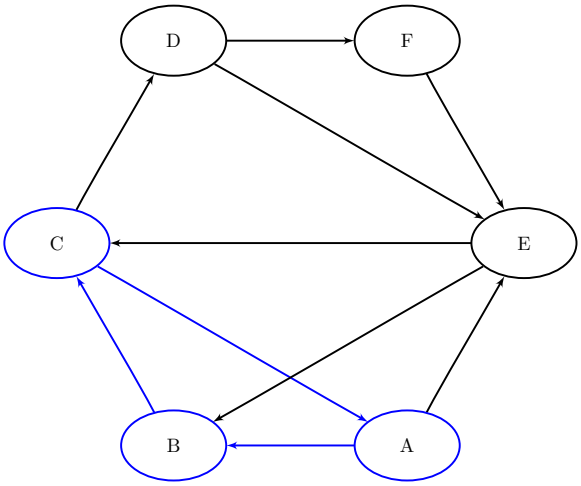
The data set Cycles now contains the first cycle found in the input graph; it is shown in Figure 2.33.

Figure 2.33 First Cycle Found in a Simple Directed Graph

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

The first cycle that is found in the input graph is shown graphically in Figure 2.34.

Figure 2.34 $A \rightarrow B \rightarrow C \rightarrow A$



The following statements return all the cycles in the graph:

```
proc optnet
   graph_direction = directed
   data_links      = LinkSetIn;
   cycle
      out          = Cycles
      mode         = all_cycles;
run;
```

The data set Cycles now contains all the cycles in the input graph; it is shown in Figure 2.35.

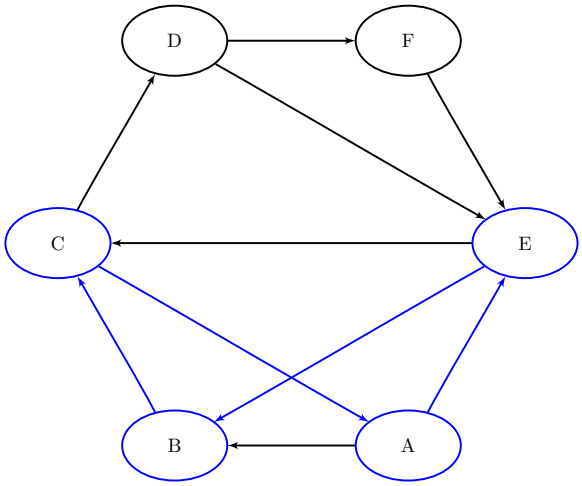
Figure 2.35 All Cycles in a Simple Directed Graph

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

The six additional cycles are shown graphically in Figure 2.36 through Figure 2.38.

Figure 2.36 Cycles

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



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

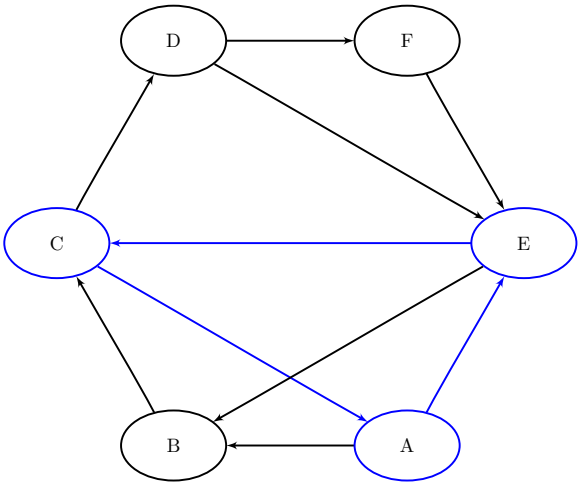
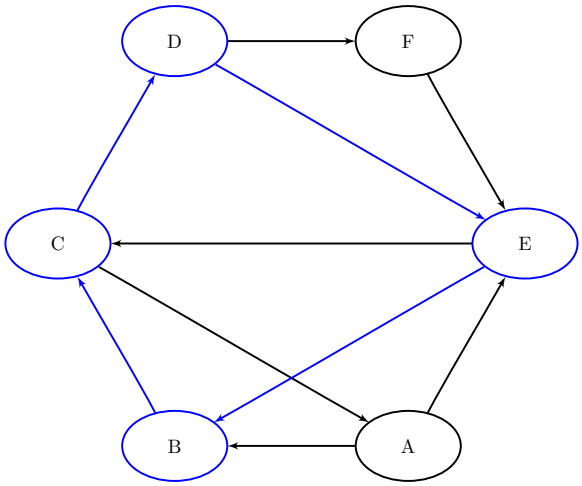


Figure 2.37 Cycles

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



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

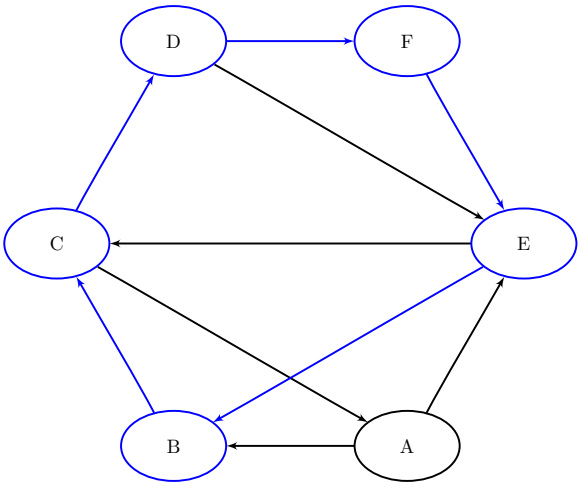
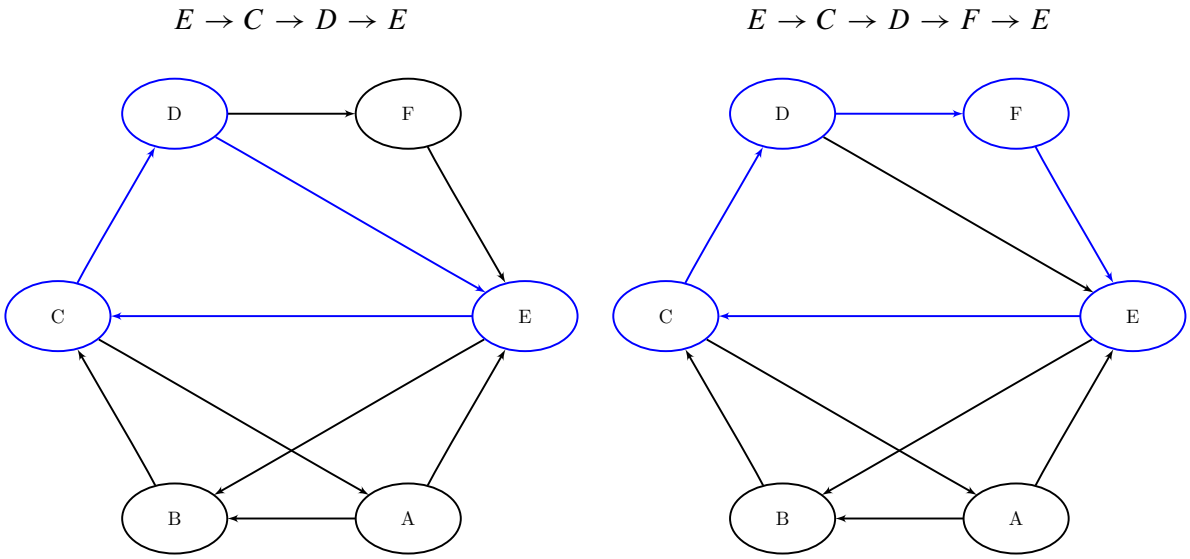


Figure 2.38 Cycles



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 from one group to some abstract object from a second group.

The linear assignment problem can be formulated as an integer programming optimization problem. The form of the problem depends on the sizes of the two input sets, S and T. Let A represent the set of possible assignments between sets S and T. In the bipartite graph, these assignments are the links. If $|S| \geq |T|$, then the following optimization problem is solved:

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

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

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

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

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

In PROC OPTNET, you can invoke the linear assignment problem solver by using the LINEAR_ASSIGNMENT statement. The options for this statement are described in the section "LINEAR_ASSIGNMENT Statement" on page 22. The algorithm that the PROC OPTNET uses for solving a LAP is based on augmentation of shortest paths (Jonker and Volgenant 1987). This algorithm can be applied to either matrix data input (see the section "Matrix Input Data" on page 41) or graph data input (see the section "Graph Input Data" on page 34) as long as the graph is bipartite.

The resulting assignment (or matching) is contained in the output data set that is specified in the OUT= option in the LINEAR_ASSIGNMENT statement.

The linear assignment problem solver reports status information in a macro variable called _OROPTNET_LAP_. For more information about this macro variable, see the section "Macro Variable _OROPTNET_LAP_" on page 102.

For a detailed example, see "Example 2.3: Linear Assignment Problem for Minimizing Swim Times" on page 115.

Minimum-Cost Network Flow

The *minimum-cost network flow* (MCF) problem is a fundamental problem in network analysis that involves sending flow over a network at minimal cost. Let $G = (N, A)$ be a directed graph. For each link $(i, j) \in A$, associate a cost per unit of flow, designated as c_{ij}. The demand (or supply) at each node $i \in N$ is designated as b_i, where $b_i \geq 0$ 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_{ij} that denote the amount of flow sent from node i to node j. The amount of flow that can be sent across each link is bounded to be within $[l_{ij}, u_{ij}]$. The problem can be modeled as a linear programming problem as follows:

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

When $b_i = b_i^l = b_i^u$ for all nodes $i \in N$, the problem is called a *pure network flow problem*. For these problems, the sum of the supplies and demands must be equal to 0 to ensure that a feasible solution exists.

In PROC OPTNET, you can invoke the minimum-cost network flow solver by using the MINCOSTFLOW statement. The options for this statement are described in the section "MINCOSTFLOW Statement" on page 23.

The minimum-cost network flow solver reports status information in a macro variable called _OROPTNET_MCF_. For more information about this macro variable, see the section "Macro Variable _OROPTNET_MCF_" on page 102.

The algorithm that PROC OPTNET 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 34. The links data set, which is specified in the DATA_LINKS= option in the PROC OPTNET statement, contains the following columns:

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

The nodes data set, which is specified in the DATA_NODES= option in the PROC OPTNET statement, can contain the following columns:

- weight, which defines the node supply lower bound b_i^l. The default is 0.
- weight2, which defines the node supply upper bound b_i^u. The default is ∞.

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

To explicitly define an upper bound of ∞, use the special missing value, (.I). To explicitly define a lower bound of $-\infty$, use the special missing value, (.M). Related to infinite bounds, the following scenarios are not supported:

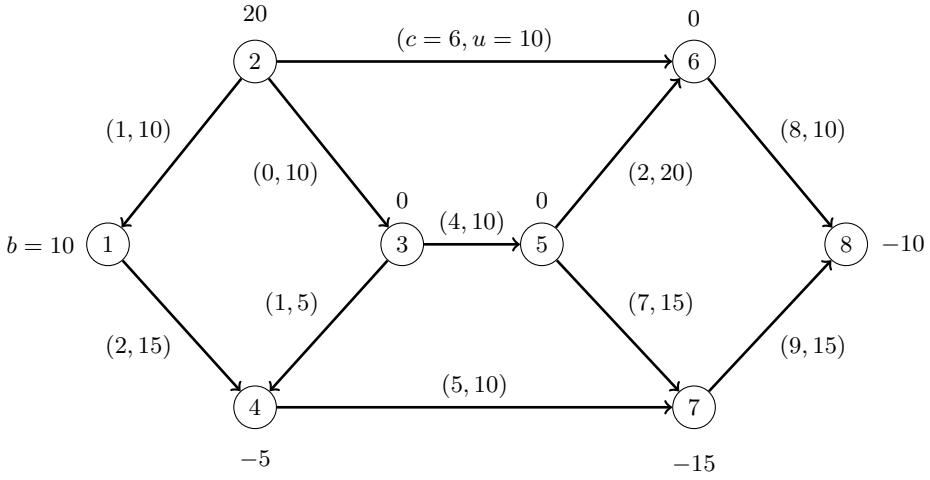
- The flow on a link must be bounded from below ($l_{ij} = -\infty$ is not allowed).
- Flow balance constraints cannot be *free* ($b_i^l = -\infty$ and $b_i^u = \infty$ is not allowed).

The resulting optimal flow through the network is written to the links output data set, which is specified in the OUT_LINKS= option in the PROC OPTNET statement.

Minimum-Cost Network Flow for a Simple Directed Graph

This example demonstrates how to use the network simplex algorithm to find a minimum-cost flow in a directed graph. Consider the directed graph in Figure 2.39, which appears in Ahuja, Magnanti, and Orlin (1993).

Figure 2.39 Minimum-Cost Network Flow Problem: Data



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

```
data LinkSetIn;
   input from to weight upper;
   datalines;
1 4 2 15
2 1 1 10
2 3 0 10
2 6 6 10
3 4 1  5
3 5 4 10
4 7 5 10
5 6 2 20
5 7 7 15
6 8 8 10
7 8 9 15
;

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

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

```
proc optnet
   loglevel        = moderate
   graph_direction = directed
   data_links      = LinkSetIn
   data_nodes      = NodeSetIn
   out_links       = LinkSetOut;
   mincostflow
      logfreq      = 1;
run;
%put &_OROPTNET_;
%put &_OROPTNET_MCF_;
```

The progress of the procedure is shown in Figure 2.40.

Figure 2.40 PROC OPTNET Log for Minimum-Cost Network Flow

```
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Running OPTNET version 15.1.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The OPTNET procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Reading the nodes data set.
NOTE: There were 5 observations read from the data set WORK.NODESETIN.
NOTE: Reading the links data set.
NOTE: There were 11 observations read from the data set WORK.LINKSETIN.
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 input graph storage is using 0.0 MBs (peak: 0.0 MBs) of memory.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 11.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Processing the minimum-cost network flow problem.
NOTE: The network has 1 connected component.
                      Primal         Primal           Dual
      Iteration      Objective  Infeasibility  Infeasibility     Time
              1   0.000000E+00   2.000000E+01   8.900000E+01     0.00
              2   0.000000E+00   2.000000E+01   8.900000E+01     0.00
              3   5.000000E+00   1.500000E+01   8.400000E+01     0.00
              4   5.000000E+00   1.500000E+01   8.300000E+01     0.00
              5   5.000000E+00   1.500000E+01   8.300000E+01     0.00
              6   7.500000E+01   1.500000E+01   7.900000E+01     0.00
              7   1.300000E+02   1.000000E+01   7.600000E+01     0.00
              8   1.300000E+02   1.000000E+01   7.600000E+01     0.00
              9   1.300000E+02   1.000000E+01   7.600000E+01     0.00
             10   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: ---------------------------------------------------------------------------------------------
NOTE: Creating links data set output.
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The data set WORK.LINKSETOUT has 11 observations and 5 variables.
STATUS=OK  MCF=OPTIMAL
STATUS=OPTIMAL  OBJECTIVE=270  CPU_TIME=0.00  REAL_TIME=0.00
```

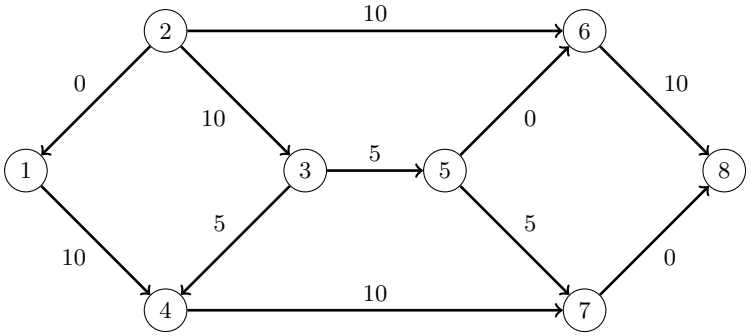
The optimal solution is displayed in Figure 2.41.

Figure 2.41 Minimum-Cost Network Flow Problem: Optimal Solution

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

The optimal solution is represented graphically in Figure 2.42.

Figure 2.42 Minimum-Cost Network Flow Problem: Optimal Solution



Minimum-Cost Network Flow with Flexible Supply and Demand

Using the same directed graph shown in Figure 2.39, this example demonstrates a network that has a flexible supply and demand. Consider the following adjustments to the node bounds:

- Node 1 has an infinite supply, but it still requires at least 10 units to be sent.
- Node 4 is a throughput node that can now handle an infinite amount of demand.
- Node 8 has a flexible demand. It requires between 6 and 10 units.

You use the special missing values (.I) to represent infinity and (.M) to represent minus infinity. The adjusted node bounds can be represented by the following nodes data set:

```
data NodeSetIn;
   input node weight weight2;
   datalines;
1  10  .I
2  20  20
4  .M  -5
7 -15 -15
8 -10  -6
;
```

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

```
proc optnet
   loglevel        = moderate
   graph_direction = directed
   data_links      = LinkSetIn
   data_nodes      = NodeSetIn
   out_links       = LinkSetOut;
   mincostflow
      logfreq      = 1;
run;
%put &_OROPTNET_;
%put &_OROPTNET_MCF_;
```

The progress of the procedure is shown in Figure 2.43.

Figure 2.43 PROC OPTNET Log for Minimum-Cost Network Flow

```
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Running OPTNET version 15.1.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The OPTNET procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Reading the nodes data set.
NOTE: There were 5 observations read from the data set WORK.NODESETIN.
NOTE: Reading the links data set.
NOTE: There were 11 observations read from the data set WORK.LINKSETIN.
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 input graph storage is using 0.0 MBs (peak: 0.0 MBs) of memory.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 11.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Processing the minimum-cost network flow problem.
NOTE: The network has 1 connected component.
                      Primal         Primal           Dual
      Iteration      Objective  Infeasibility  Infeasibility     Time
              1   1.000000E+01   2.000000E+01   1.730000E+02     0.00
              2   1.000000E+01   2.000000E+01   1.730000E+02     0.00
              3   4.500000E+01   2.000000E+01   1.740000E+02     0.00
              4   4.500000E+01   2.000000E+01   1.740000E+02     0.00
              5   7.500000E+01   1.500000E+01   1.660000E+02     0.00
              6   7.500000E+01   1.500000E+01   1.660000E+02     0.00
              7   1.590000E+02   9.000000E+00   8.700000E+01     0.00
              8   1.590000E+02   9.000000E+00   1.690000E+02     0.00
              9   2.140000E+02   4.000000E+00   8.700000E+01     0.00
             10   2.140000E+02   4.000000E+00   8.700000E+01     0.00
             11   2.260000E+02   0.000000E+00   0.000000E+00     0.00
             12   2.260000E+02   0.000000E+00   0.000000E+00     0.00
             13   2.260000E+02   0.000000E+00   0.000000E+00     0.00
             14   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: ---------------------------------------------------------------------------------------------
NOTE: Creating links data set output.
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The data set WORK.LINKSETOUT has 11 observations and 5 variables.
STATUS=OK  MCF=OPTIMAL
STATUS=OPTIMAL  OBJECTIVE=226  CPU_TIME=0.00  REAL_TIME=0.00
```

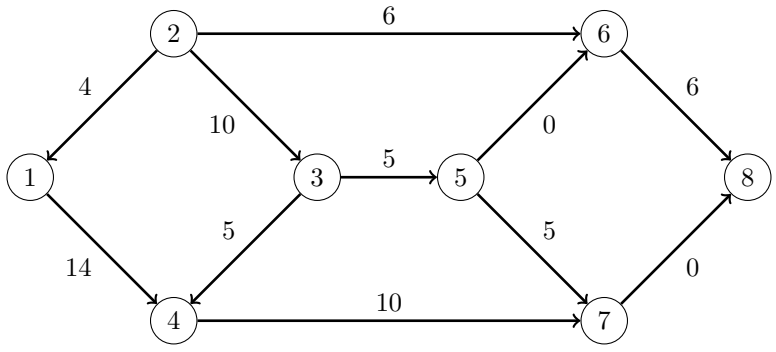
The optimal solution is displayed in Figure 2.44.

Figure 2.44 Minimum-Cost Network Flow Problem: Optimal Solution

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

The optimal solution is represented graphically in Figure 2.45.

Figure 2.45 Minimum-Cost Network Flow Problem: Optimal Solution



Minimum Cut

A *cut* is a partition of the nodes of a graph into two disjoint subsets. The *cut-set* is the set of links whose *from* and *to* nodes are in different subsets of the partition. A *minimum cut* of an undirected graph is a cut whose cut-set has the smallest link metric, which is measured as follows: For an unweighted graph, the link metric is the number of links in the cut-set. For a weighted graph, the link metric is the sum of the link weights in the cut-set.

In PROC OPTNET, you can invoke the minimum-cut algorithm by using the MINCUT statement. The options for this statement are described in the section "MINCUT Statement" on page 24. This algorithm can be used only on undirected graphs.

If the value of the MAXNUMCUTS= option is greater than 1, then the algorithm can return more than one set of cuts. The resulting cuts can be described in terms of partitions of the nodes of the graph or the links in the cut-sets. The node partition is specified by the mincut_*i* variable, for each cut *i*, in the data set that is specified in the OUT_NODES= option in the PROC OPTNET statement. Each node is assigned the value 0 or 1, which defines the side of the partition to which it belongs. The cut-set is defined in the output data set

that is specified in the OUT= option in the MINCUT statement. This data set lists the links and their weights for each cut.

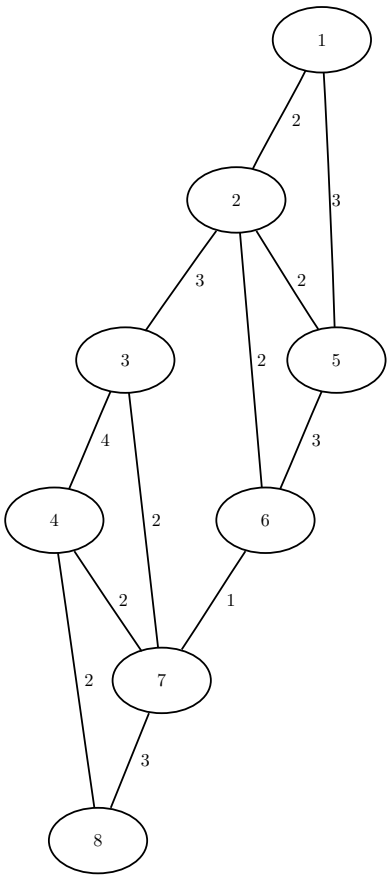
The minimum-cut algorithm reports status information in a macro variable called _OROPTNET_MINCUT_. For more information about this macro variable, see the section "Macro Variable _OROPTNET_MINCUT_" on page 103.

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

Minimum Cut for a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 2.46.

Figure 2.46 A Simple Undirected Graph



The links data set can be represented as follows:

```
data LinkSetIn;
   input from to weight @@;
   datalines;
1 2 2  1 5 3  2 3 3  2 5 2  2 6 2
3 4 4  3 7 2  4 7 2  4 8 2  5 6 3
6 7 1  7 8 3
;
```

The following statements calculate minimum cuts in the graph and output the results in the data set MinCut:

```
proc optnet
   loglevel     = moderate
   out_nodes    = NodeSetOut
   data_links   = LinkSetIn;
   mincut
     out        = MinCut
     maxnumcuts = 3;
run;
%put &_OROPTNET_;
%put &_OROPTNET_MINCUT_;
```

The progress of the procedure is shown in Figure 2.47.

Figure 2.47 PROC OPTNET Log for Minimum Cut

```
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Running OPTNET version 15.1.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The OPTNET procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: There were 12 observations read from the data set WORK.LINKSETIN.
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 input graph storage is using 0.0 MBs (peak: 0.0 MBs) of memory.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 12.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Processing the minimum-cut problem.
NOTE: The minimum-cut algorithm found 3 cuts.
NOTE: The cut 1 has weight 4.
NOTE: The cut 2 has weight 5.
NOTE: The cut 3 has weight 5.
NOTE: Processing the minimum-cut problem used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Creating nodes data set output.
NOTE: Creating minimum-cut data set output.
NOTE: Data output used 0.01 (cpu: 0.01) seconds.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The data set WORK.NODESETOUT has 8 observations and 4 variables.
NOTE: The data set WORK.MINCUT has 6 observations and 4 variables.
STATUS=OK  MINCUT=OPTIMAL
STATUS=OPTIMAL  OBJECTIVE=4  CPU_TIME=0.00  REAL_TIME=0.00
```

The data set NodeSetOut now contains the partition of the nodes for each cut, shown in Figure 2.48.

Figure 2.48 Minimum Cut Node Partition

node	mincut_1	mincut_2	mincut_3
1	1	1	1
2	1	1	0
5	1	1	0
3	0	1	0
6	1	1	0
4	0	1	0
7	0	1	0
8	0	0	0

The data set MinCut contains the links in the cut-sets for each cut. This data set is shown in Figure 2.49, which also shows each cut separately.

Figure 2.49 Minimum Cut Sets

mincut	from	to	weight
1	2	3	3
1	6	7	1
2	4	8	2
2	7	8	3
3	1	2	2
3	1	5	3

mincut=1

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

mincut=2

from	to	weight
4	8	2
7	8	3
mincut		**5**

mincut=3

from	to	weight
1	2	2
1	5	3
mincut		**5**
		14

Minimum Spanning Tree

A *spanning tree* of a connected undirected graph is a subgraph that is a tree that connects all the nodes together. When weights have been assigned to the links, a *minimum spanning tree* (MST) is a spanning tree whose sum of link weights is less than or equal to the sum of link weights of every other spanning tree. More generally, any undirected graph (not necessarily connected) has a *minimum spanning forest*, which is a union of minimum spanning trees of its connected components.

In PROC OPTNET, 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 25. This algorithm can be used only on undirected graphs.

The resulting minimum spanning tree is contained in the output data set that is specified in the OUT= option in the MINSPANTREE statement.

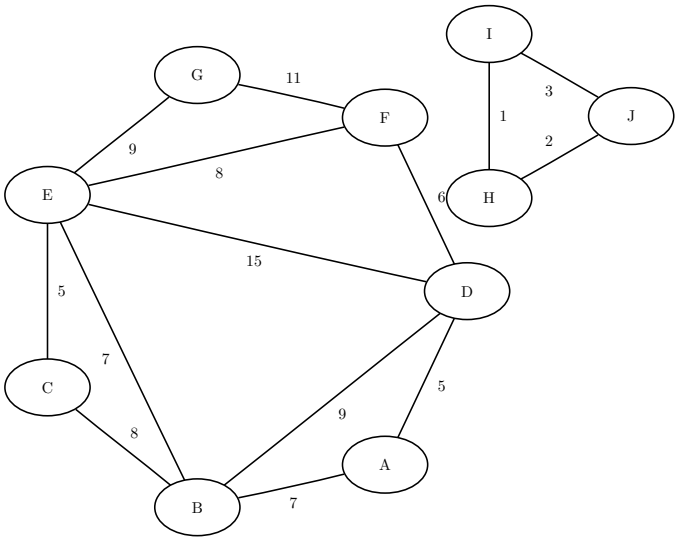
The minimum spanning tree algorithm reports status information in a macro variable called _OROPTNET_MST_. For more information about this macro variable, see the section "Macro Variable _OROPTNET_MST_" on page 103.

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

Minimum Spanning Tree for a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 2.50.

Figure 2.50 A Simple Undirected Graph



The links data set can be represented as follows:

```
data LinkSetIn;
   input from $ to $ weight @@;
   datalines;
A B  7  A D  5  B C 8  B D 9  B E 7
```

```
C E  5  D E 15  D F 6  E F 8  E G 9
F G 11  H I  1  I J 3  H J 2
;
```

The following statements calculate a minimum spanning forest and output the results in the data set MinSpanForest:

```
proc optnet
   data_links = LinkSetIn;
   minspantree
      out     = MinSpanForest;
run;
```

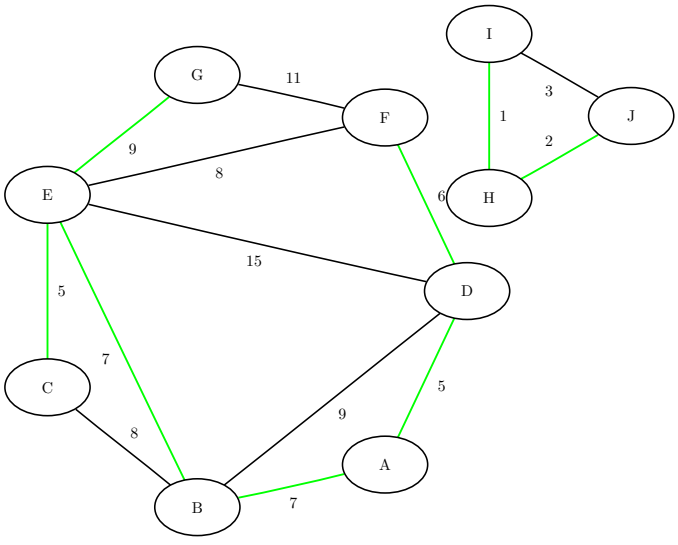
The data set MinSpanForest now contains the links that belong to a minimum spanning forest, which is shown in Figure 2.51.

Figure 2.51 Minimum Spanning Forest

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

The minimal cost links are shown in green in Figure 2.52.

Figure 2.52 Minimum Spanning Forest



For a more detailed example, see "Example 2.5: Minimum Spanning Tree for Computer Network Topology" on page 120.

Shortest Path

A *shortest path* between two nodes *u* and *v* in a graph is a path that starts at *u* and ends at *v* and has the lowest total link weight. The starting node is called the *source node*, and the ending node is called the *sink node*.

In PROC OPTNET, you can calculate shortest paths by using the SHORTPATH statement. The options for this statement are described in the section "SHORTPATH Statement" on page 27.

The shortest path algorithm reports status information in a macro variable called _OROPTNET_SHORTPATH_. For more information about this macro variable, see the section "Macro Variable _OROPTNET_SHORTPATH_" on page 104.

By default, PROC OPTNET finds shortest paths for all pairs. That is, it finds a shortest path for each possible combination of source and sink nodes. Alternatively, you can use the SOURCE= 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 DATA_NODES_SUB= option to define a list of source-sink pairs to process, as described in the section "Node Subset Input Data" on page 38. The following sections show examples of these options.

Which algorithm PROC OPTNET uses to find shortest paths depends on the data. The algorithm and run-time complexity for each link type are shown in Table 2.33.

Table 2.33 Algorithms for Shortest Paths

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

Details for each algorithm can be found in Ahuja, Magnanti, and Orlin (1993).

For weighted graphs, the algorithm uses the weight variable that is defined in the links data set to evaluate a path's total weight (cost). You can also use the WEIGHT2= option in the SHORTPATH 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 Sets

The shortest path algorithm produces up to two output data sets. The output data set that you specify in the OUT_PATHS= option contains the links of a shortest path for each source-sink pair combination. The output data set that you specify in the OUT_WEIGHTS= option contains the total weight for the shortest path for each source-sink pair combination.

OUT_PATHS= Data Set

The OUT_PATHS= data set contains the links present in the shortest path for each source-sink pair. For large graphs and a large requested number of source-sink pairs, this output data set can be extremely large. For extremely large sets, 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 set of size 11 GB. Finding shortest paths for all pairs from all nodes would produce an enormous output data set.

The OUT_PATHS= data set 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
- weight2: the auxiliary weight of this link

OUT_WEIGHTS= Data Set

The OUT_WEIGHTS= data set contains the total weight (and total auxiliary weight) for the shortest path for each source-sink pair.

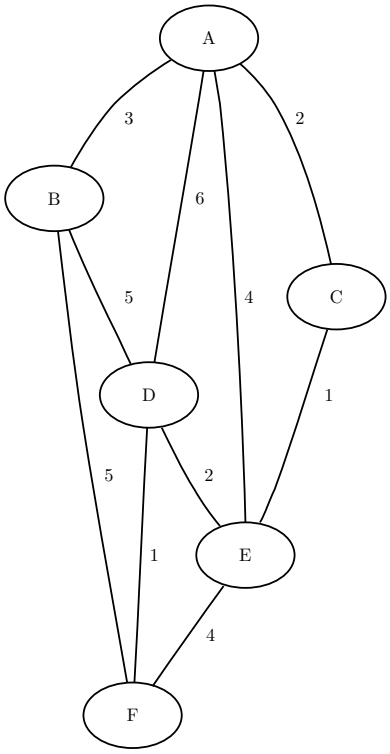
This data set contains the following columns:

- source: the source node label of this shortest path
- sink: the sink node label of this shortest path
- path_weight: the total weight of the shortest path for this source-sink pair
- path_weight2: the total auxiliary weight of the shortest path for this source-sink pair

Shortest Paths for All Pairs

This example illustrates the use of the shortest path algorithm for all source-sink pairs on the simple undirected graph *G* shown in Figure 2.53.

Figure 2.53 A Simple Undirected Graph G



The undirected graph G can be represented by the following links data set, LinkSetIn:

```
data LinkSetIn;
   input from $ to $ weight @@;
   datalines;
A B 3  A C 2  A D 6  A E 4  B D 5
B F 5  C E 1  D E 2  D F 1  E F 4
;
```

The following statements calculate shortest paths for all source-sink pairs:

```
proc optnet
   data_links     = LinkSetIn;
   shortpath
      out_weights = ShortPathW
      out_paths   = ShortPathP;
run;
```

The data set ShortPathP contains the shortest paths and is shown in Figure 2.54.

Figure 2.54 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 data set ShortPathW contains the path weight for the shortest paths of each source-sink pair and is shown in Figure 2.55.

Figure 2.55 All-Pairs Shortest Paths Summary

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

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

When you are interested only in the source-sink pair that has the longest shortest path, you can use the PATHS= option. This option affects only the output processing; it does not affect the computation. All the designated source-sink shortest paths are calculated, but only the longest ones are written to the output data set.

The following statements display only the longest shortest paths:

```
proc optnet
   data_links   = LinkSetIn;
   shortpath
      paths     = longest
      out_paths = ShortPathLong;
run;
```

The data set ShortPathLong now contains the longest shortest paths and is shown in Figure 2.56.

Figure 2.56 Longest Shortest Paths

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

Shortest Paths for a Subset of Source-Sink Pairs

This section illustrates the use of a node subset data set, the DATA_NODES_SUB= option, and the shortest path algorithm to calculate shortest paths for a subset of source-sink pairs. The data set 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 OPTNET then calculates all the source-sink pairs in the crossproduct of these two sets.

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

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

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

```
proc optnet
   data_nodes_sub = NodeSubSetIn
   data_links     = LinkSetIn;
   shortpath
      out_paths   = ShortPath;
run;
```

The data set ShortPath contains the shortest paths and is shown in Figure 2.57.

Figure 2.57 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 calculate shortest paths between a subset of source (or sink) nodes and all other sink (or source) nodes.

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

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

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

You can use the same PROC OPTNET call as is used in the section "Shortest Paths for a Subset of Source-Sink Pairs" on page 81 to calculate all the shortest paths from nodes *B* and *E*. The data set ShortPath contains the shortest paths and is shown in Figure 2.58.

Figure 2.58 Shortest Paths for a Subset of Source Pairs

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

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

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

You can use the same PROC OPTNET call again to calculate all the shortest paths to nodes *B* and *E*. The data set ShortPath contains the shortest paths and is shown in Figure 2.59.

Figure 2.59 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
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
F	B	1	F	B	5
F	E	1	F	D	1
F	E	2	D	E	2

Shortest Paths for One Source-Sink Pair

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

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

```
proc optnet
   data_links   = LinkSetIn;
   shortpath
      source    = C
      sink      = F
      out_paths = ShortPath;
run;
```

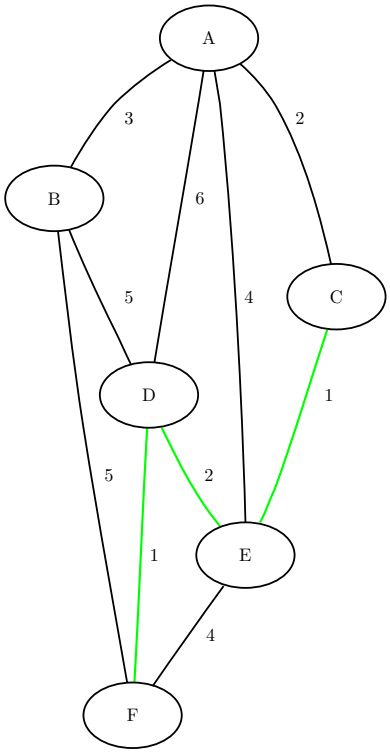
The data set ShortPath contains this shortest path and is shown in Figure 2.60.

Figure 2.60 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 2.61.

Figure 2.61 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 calculate the shortest paths between all source-sink pairs.

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

```
data LinkSetIn;
   input from $ to $ weight count @@;
   datalines;
A B 3 1  A C 2 1  A D 6 1  A E 4 1  B D 5 1
B F 5 1  C E 1 1  D E 2 1  D F 1 1  E F 4 1
;
```

The following statements calculate shortest paths for all source-sink pairs:

```
proc optnet
   data_links     = LinkSetIn;
   shortpath
      weight2     = count
      out_weights = ShortPathW;
run;
```

The data set ShortPathW contains the total path weight for shortest paths in each source-sink pair and is shown in Figure 2.62. Because the variable count in LinkSetIn has a value of 1 for all links, the value in the output data set variable path_weights2 contains the number of links in each shortest path.

Figure 2.62 Shortest Paths Including Auxiliary Weights in Calculation

source	sink	path_weight	path_weight2
A	B	3	1
A	C	2	1
A	D	5	3
A	E	3	2
A	F	6	4
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

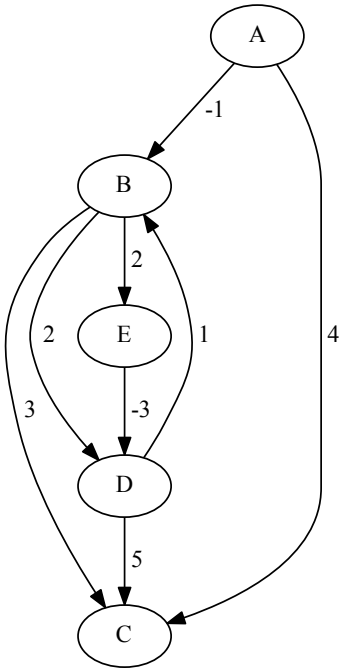
source	sink	path_weight	path_weight2
D	A	5	3
D	B	5	1
D	C	3	2
D	E	2	1
D	F	1	1
E	A	3	2
E	B	6	3
E	C	1	1
E	D	2	1
E	F	3	2
F	A	6	4
F	B	5	1
F	C	4	3
F	D	1	1
F	E	3	2

The section "Road Network Shortest Path" on page 7 shows an example of using the shortest path algorithm to minimize travel to and from work based on traffic conditions.

Shortest Paths with Negative Link Weights

This section illustrates the use of the shortest path algorithm on a simple directed graph *G* with negative link weights, shown in Figure 2.63.

Figure 2.63 A Simple Directed Graph *G* with Negative Link Weights



You can represent the directed graph *G* by using the following links data set LinkSetIn:

```
data 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 calculate the shortest paths between source node *E* and sink node *B*:

```
proc optnet
   direction      = directed
   data_links     = LinkSetIn;
   shortpath
      source      = E
      sink        = B
      out_paths   = ShortPathP;
run;
```

The data set ShortPathP contains the shortest path from node *E* to node *B* and is shown in Figure 2.64.

Figure 2.64 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 LinkSetIn;
   set 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 this and produces an error, as shown in Figure 2.65.

Figure 2.65 PROC OPTNET Log: Negative Weight Cycle

```
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Running OPTNET version 15.1.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The OPTNET procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Data input 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 8.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Processing the shortest paths problem.
ERROR: The graph contains a negative weight cycle.
NOTE: Processing the shortest paths problem used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The data set WORK.SHORTPATHP has 0 observations and 6 variables.
STATUS=OK  SHORTPATH=ERROR
```

Transitive Closure

The *transitive closure* of a graph G is a graph $G^T = (N, A^T)$ such that for all $i, j \in N$ there is a link $(i, j) \in A^T$ if and only if there exists a path from i to j in G.

The transitive closure of a graph can help to efficiently answer questions about reachability. Suppose you want to answer the question of whether you can get from node i to node j in the original graph G. Given the transitive closure G^T of G, you can simply check for the existence of link (i, j) to answer the question. Transitive closure has many applications, including speeding up the processing of structured query languages, which are often used in databases.

In PROC OPTNET, you can invoke the transitive closure algorithm by using the TRANSITIVE_CLOSURE statement. The options for this statement are described in the section "TRANSITIVE_CLOSURE Statement" on page 28.

The results of the transitive closure algorithm are written to the output data set that is specified in the OUT= option in the TRANSITIVE_CLOSURE statement. The links that define the transitive closure are listed in the output data set with variable names from and to.

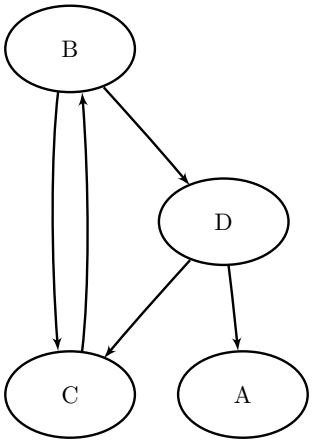
The transitive closure algorithm reports status information in a macro variable called _OROPTNET_TRANSCL_. For more information about this macro variable, see the section "Macro Variable _OROPTNET_TRANSCL_" on page 104.

The algorithm that the PROC OPTNET uses to compute transitive closure is a sparse version of the Floyd-Warshall algorithm (Cormen, Leiserson, and Rivest 1990). This algorithm runs in time $O(|N|^3)$ and therefore might not scale to very large graphs.

Transitive Closure of a Simple Directed Graph

This example illustrates the use of the transitive closure algorithm on the simple directed graph G, which is shown in Figure 2.66.

Figure 2.66 A Simple Directed Graph G



The directed graph G can be represented by the following links data set LinkSetIn:

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

The following statements calculate the transitive closure and output the results in the data set TransClosure:

```
proc optnet
   graph_direction = directed
   data_links      = LinkSetIn;
   transitive_closure
      out          = TransClosure;
run;
```

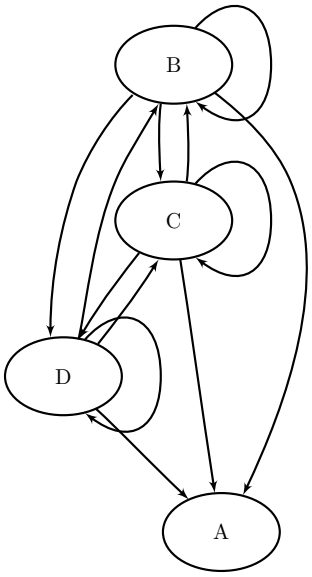
The data set TransClosure contains the transitive closure of G and is shown in Figure 2.67.

Figure 2.67 Transitive Closure of a Simple Directed Graph

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

The transitive closure of G is shown graphically in Figure 2.68.

Figure 2.68 Transitive Closure of G



For a more detailed example, see "Example 2.6: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System" on page 122.

Traveling Salesman Problem

The *traveling salesman problem* (TSP) finds a minimum-cost tour in a graph, G, that has a node set, N, and a link set, A. A *path* in a graph is a sequence of nodes, each of which has a link to the next node in the sequence. An *elementary cycle* is a path in which the start node and end node are the same and no node appears more than once in the sequence. A *Hamiltonian cycle* (or *tour*) is an elementary cycle that visits every node. In solving the TSP, then, the goal is to find a Hamiltonian cycle of minimum total cost, where the total cost is

the sum of the costs of the links in the tour. Associated with each link $(i, j) \in A$ are a binary variable x_{ij}, which indicates whether link x_{ij} is part of the tour, and a cost c_{ij}. Let $\delta(S) = \{(i, j) \in A \mid i \in S,\ j \notin S\}$. Then an integer linear programming formulation of the TSP (for an undirected graph G) is as follows:

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

The equations (two_match) are the *matching constraints*, which ensure that each node has degree two in the subgraph. The inequalities (subtour_elim) are the *subtour elimination constraints* (SECs), which enforce connectivity.

For a directed graph, G, the same formulation and solution approach is used on an expanded graph G', as described in Kumar and Li (1994). PROC OPTNET 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 OPTNET, 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 29.

The traveling salesman problem solver reports status information in a macro variable called _OROPTNET_TSP_. For more information about this macro variable, see the section "Macro Variable _OROPTNET_TSP_" on page 104.

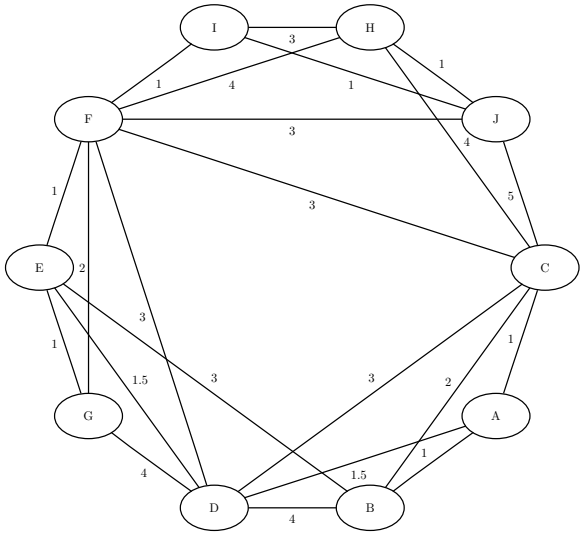
The algorithm that PROC OPTNET uses for solving the TSP is based on a variant of the branch-and-cut process described in Applegate et al. (2006).

The resulting tour is represented in two ways: in the data set that you specify in the OUT_NODES= option in the PROC OPTNET statement, the tour is specified as a sequence of nodes; in the data set that you specify in the OUT= option of the TSP statement, the tour is specified as a list of links in the optimal tour.

Traveling Salesman Problem Applied to a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 2.69.

Figure 2.69 A Simple Undirected Graph



You can represent the links data set as follows:

```
data LinkSetIn;
   input from $ to $ weight @@;
   datalines;
A B 1.0 A C 1.0 A D 1.5 B C 2.0 B D 4.0
B E 3.0 C D 3.0 C F 3.0 C H 4.0 D E 1.5
D F 3.0 D G 4.0 E F 1.0 E G 1.0 F G 2.0
F H 4.0 H I 3.0 I J 1.0 C J 5.0 F J 3.0
F I 1.0 H J 1.0
;
```

The following statements calculate an optimal traveling salesman tour and output the results in the data sets TSPTour and NodeSetOut:

```
proc optnet
   loglevel   = moderate
   data_links = LinkSetIn
   out_nodes  = NodeSetOut;
   tsp
      out     = TSPTour;
run;
%put &_OROPTNET_;
%put &_OROPTNET_TSP_;
```

The progress of the OPTNET procedure is shown in Figure 2.70.

Figure 2.70 PROC OPTNET Log: Optimal Traveling Salesman Tour of a Simple Undirected Graph

```
NOTE: ------------------------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------------------------
NOTE: Running OPTNET version 15.1.
NOTE: ------------------------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------------------------
NOTE: The OPTNET procedure is executing in single-machine mode.
NOTE: ------------------------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: There were 22 observations read from the data set WORK.LINKSETIN.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: Building the input graph storage used 0.00 (cpu: 0.01) seconds.
NOTE: The input graph storage is using 0.0 MBs (peak: 0.0 MBs) of memory.
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 22.
NOTE: ------------------------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------------------------
NOTE: Processing the traveling salesman problem.
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: Objective = 16.
NOTE: Processing the traveling salesman problem used 0.01 (cpu: 0.00) seconds.
NOTE: ------------------------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------------------------
NOTE: Creating nodes data set output.
NOTE: Creating traveling salesman data set output.
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ------------------------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------------------------
NOTE: The data set WORK.NODESETOUT has 10 observations and 2 variables.
NOTE: The data set WORK.TSPTOUR has 10 observations and 3 variables.
STATUS=OK  TSP=OPTIMAL
STATUS=OPTIMAL  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=0.01  REAL_TIME=0.01
```

The data set NodeSetOut now contains a sequence of nodes in the optimal tour and is shown in Figure 2.71.

Figure 2.71 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

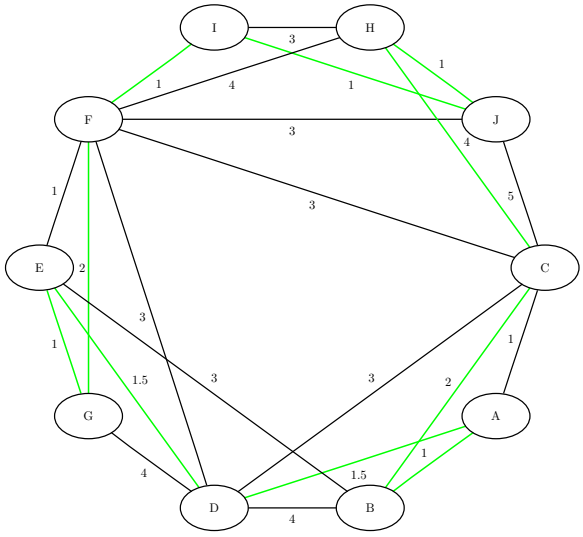
The data set TSPTour now contains the links in the optimal tour and is shown in Figure 2.72.

Figure 2.72 Links in the Optimal Traveling Salesman Tour

from	to	weight
A	B	1.0
B	C	2.0
C	H	4.0
H	J	1.0
I	J	1.0
F	I	1.0
F	G	2.0
E	G	1.0
D	E	1.5
A	D	1.5
		16.0

The minimum-cost links are shown in green in Figure 2.73.

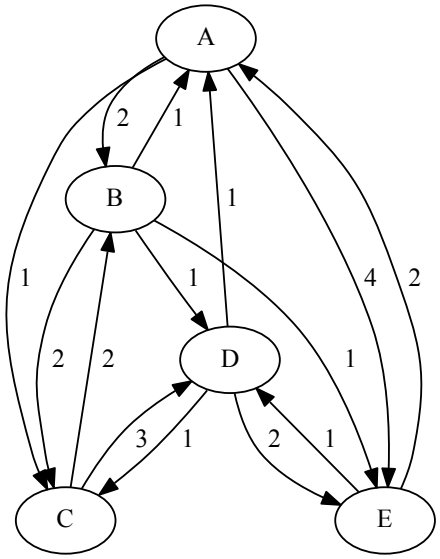
Figure 2.73 Optimal Traveling Salesman Tour



Traveling Salesman Problem Applied to a Simple Directed Graph

As another simple example, consider the weighted directed graph in Figure 2.74.

Figure 2.74 A Simple Directed Graph



You can represent the links data set as follows:

```
data LinkSetIn;
   input from $ to $ weight @@;
   datalines;
A B 2.0 A C 1.0 A E 4.0 B A 1.0 B C 2.0
B D 1.0 B E 1.0 C B 2.0 C D 3.0 D A 1.0
D C 1.0 D E 2.0 E A 2.0 E D 1.0
;
```

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

```
proc optnet
   direction  = directed
   loglevel   = moderate
   data_links = LinkSetIn
   out_nodes  = NodeSetOut;
   tsp
      out     = TSPTour;
run;
%put &_OROPTNET_;
%put &_OROPTNET_TSP_;
```

The progress of the OPTNET procedure is shown in Figure 2.75.

Figure 2.75 PROC OPTNET Log: Optimal Traveling Salesman Tour of a Simple Directed Graph

```
NOTE: ----------------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------------
NOTE: Running OPTNET version 15.1.
NOTE: ----------------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------------
NOTE: The OPTNET procedure is executing in single-machine mode.
NOTE: ----------------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: There were 14 observations read from the data set WORK.LINKSETIN.
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 input graph storage is using 0.0 MBs (peak: 0.0 MBs) of memory.
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 14.
NOTE: ----------------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------------
NOTE: The TSP solver is starting using an augmented symmetric graph with 10 nodes and 19 links.
NOTE: ----------------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------------
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 6 using 0.00 (cpu: 0.00) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Branch and Cut algorithm is used.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       1      6.0000000      5.9001000    1.69%       0
             0       0       1      6.0000000      6.0000000    0.00%       0
NOTE: Objective = 6.
NOTE: Processing the traveling salesman problem used 0.01 (cpu: 0.00) seconds.
NOTE: ----------------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------------
NOTE: Creating nodes data set output.
NOTE: Creating traveling salesman data set output.
NOTE: Data output used 0.01 (cpu: 0.00) seconds.
NOTE: ----------------------------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------------------------
NOTE: The data set WORK.NODESETOUT has 5 observations and 2 variables.
NOTE: The data set WORK.TSPTOUR has 5 observations and 3 variables.
STATUS=OK  TSP=OPTIMAL
STATUS=OPTIMAL  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=0.00  REAL_TIME=0.01
```

The data set NodeSetOut now contains a sequence of nodes in the optimal tour and is shown in Figure 2.76.

Figure 2.76 Nodes in the Optimal Traveling Salesman Tour

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

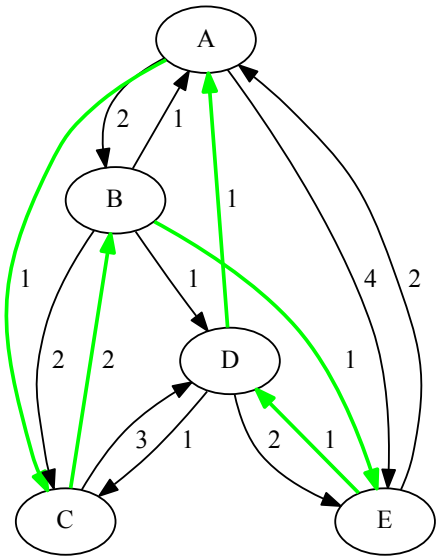
The data set TSPTour now contains the links in the optimal tour and is shown in Figure 2.77.

Figure 2.77 Links in the Optimal Traveling Salesman Tour

from	to	weight
A	C	1
C	B	2
B	E	1
E	D	1
D	A	1
		6

The minimum-cost links are shown in green in Figure 2.78.

Figure 2.78 Optimal Traveling Salesman Tour



Macro Variables

The OPTNET procedure defines one summary macro variable that reports the overall status and one detailed macro variable for each completed algorithm.

Macro Variable _OROPTNET_

The OPTNET procedure defines a macro variable named _OROPTNET_. This variable contains a character string that indicates the status of the OPTNET procedure upon termination. 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 during the input or setup phase.
ERROR	The procedure encountered an error.

BICONCOMP
indicates the status of the biconnected components algorithm at termination. This algorithm is described in the section "Biconnected Components and Articulation Points" on page 44. The BICONCOMP term can take one of the following values:

OK	The algorithm terminated normally.
ERROR	The algorithm encountered an error.

CLIQUE
indicates the status of the clique-finding algorithms at termination. These algorithms are described in the section "Clique" on page 47. The CLIQUE term can take one of the following values:

OK	The algorithm terminated normally.
INTERRUPTED	The algorithm was interrupted by the user.
TIMELIMIT	The algorithm reached its execution time limit, which is specified in the MAXTIME= option in the CLIQUE statement.
SOLUTION_LIM	The algorithm reached its limit on the number of cliques found, which is specified in the MAXCLIQUES= option in the CLIQUE statement.
ERROR	The algorithm encountered an error.

CONCOMP
indicates the status of the connected components algorithm at termination. This algorithm is described in the section "Connected Components" on page 51. The CONCOMP term can take one of the following values:

OK	The algorithm terminated normally.
ERROR	The algorithm encountered an error.

CYCLE

indicates the status of the cycle detection algorithm at termination. This algorithm is described in the section "Cycle" on page 55. The CYCLE term can take one of the following values:

OK	The algorithm terminated normally.
TIMELIMIT	The algorithm reached its execution time limit, which is specified in the MAXTIME= option in the CYCLE statement.
SOLUTION_LIM	The algorithm reached its limit on the number of cycles found, which is specified in the MAXCYCLES= option in the CYCLE statement.
ERROR	The algorithm encountered an error.

LAP

indicates the status of the linear assignment solver at termination. This solver is described in the section "Linear Assignment (Matching)" on page 61. The LAP term can take one of the following values:

OPTIMAL	The solution is optimal.
INFEASIBLE	The problem is infeasible.
ERROR	The solver encountered an error.

MCF

indicates the status of the minimum-cost network flow solver at termination. This solver is described in the section "Minimum-Cost Network Flow" on page 62. The MCF term can take one of the following values:

OPTIMAL	The solution is optimal.
OPTIMAL_COND	The solution is optimal, but some infeasibilities (primal or bound) exceed tolerances because of scaling.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
TIMELIMIT	The solver reached its execution time limit, which is specified in the MAXTIME= option in the MINCOSTFLOW statement.
FAIL_NOSOL	The solver stopped because of errors and did not find a solution.
ERROR	The solver encountered an error.

MINCUT

indicates the status of the minimum-cut solver at termination. This solver is described in the section "Minimum Cut" on page 70. The MINCUT term can take one of the following values:

OPTIMAL	The solution is optimal.
INFEASIBLE	The problem is infeasible.
INTERRUPTED	The solver was interrupted by the user.
ERROR	The solver encountered an error.

MST

indicates the status of the minimum spanning tree solver at termination. This solver is described in the section "Minimum Spanning Tree" on page 74. The MST term can take one of the following values:

OPTIMAL	The solution is optimal.
INTERRUPTED	The algorithm was interrupted by the user.
ERROR	The solver encountered an error.

SHORTPATH

indicates the status of the shortest path algorithms at termination. These algorithms are described in the section "Shortest Path" on page 76. The SHORTPATH term can take one of the following values:

OK	The algorithm terminated normally.
INTERRUPTED	The algorithm was interrupted by the user.
ERROR	The algorithm encountered an error.

TRANSCL

indicates the status of the transitive closure algorithm at termination. This algorithm is described in the section "Transitive Closure" on page 87. The TRANSCL term can take one of the following values:

OK	The algorithm terminated normally.
INTERRUPTED	The algorithm was interrupted by the user.
ERROR	The algorithm encountered an error.

TSP

indicates the status of the traveling salesman problem solver at termination. This algorithm is described in the section "Traveling Salesman Problem" on page 89. The TSP term can take one of the following values:

OPTIMAL	The solution is optimal.
OPTIMAL_AGAP	The solution is optimal within the absolute gap that is specified in the ABSOBJGAP= option.
OPTIMAL_RGAP	The solution is optimal within the relative gap that is 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 is specified in the TARGET= option.

INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
SOLUTION_LIM	The solver reached the maximum number of solutions specified in the MAXSOLS= option.
NODE_LIM_SOL	The solver reached the maximum number of nodes specified in the MAXNODES= option and found a solution.
NODE_LIM_NOSOL	The solver reached the maximum number of nodes specified in the MAXNODES= option and did not find a solution.
TIME_LIM_SOL	The solver reached the execution time limit specified in the MAXTIME= option and found a solution.
TIME_LIM_NOSOL	The solver reached the execution time limit 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.
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.

Each algorithm reports its own status information in an additional macro variable. The following sections provide more information about these macro variables.

Macro Variable _OROPTNET_BICONCOMP_

The OPTNET procedure defines a macro variable named _OROPTNET_BICONCOMP_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm that PROC OPTNET uses to calculate biconnected components. The various terms of the variable are interpreted as follows:

STATUS
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term BICONCOMP in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 97.

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

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

CPU_TIME
indicates the CPU time (in seconds) taken by the algorithm.

REAL_TIME
indicates the real time (in seconds) taken by the algorithm.

Macro Variable _OROPTNET_CLIQUE_

The OPTNET procedure defines a macro variable named _OROPTNET_CLIQUE_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm that PROC OPTNET uses to calculate cliques. The various terms of the variable are interpreted as follows:

STATUS
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term CLIQUE in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 97.

NUM_CLIQUES
indicates the number of cliques found by the algorithm.

CPU_TIME
indicates the CPU time (in seconds) taken by the algorithm.

REAL_TIME
indicates the real time (in seconds) taken by the algorithm.

Macro Variable _OROPTNET_CONCOMP_

The OPTNET procedure defines a macro variable named _OROPTNET_CONCOMP_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm that PROC OPTNET uses to calculate connected components. The various terms of the variable are interpreted as follows:

STATUS
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term CONCOMP in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 97.

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

CPU_TIME
indicates the CPU time (in seconds) taken by the algorithm.

REAL_TIME
indicates the real time (in seconds) taken by the algorithm.

Macro Variable _OROPTNET_CYCLE_

The OPTNET procedure defines a macro variable named _OROPTNET_CYCLE_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm that PROC OPTNET uses to calculate cycles. The various terms of the variable are interpreted as follows:

STATUS
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term CYCLE in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 97.

NUM_CYCLES
indicates the number of cycles found by the algorithm.

CPU_TIME
indicates the CPU time (in seconds) taken by the algorithm.

REAL_TIME
indicates the real time (in seconds) taken by the algorithm.

Macro Variable _OROPTNET_LAP_

The OPTNET procedure defines a macro variable named _OROPTNET_LAP_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm that PROC OPTNET uses to solve the linear assignment problem. The various terms of the variable are interpreted as follows:

STATUS
indicates the status of the solver at termination. The STATUS term takes the same value as the term LAP in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 97.

OBJECTIVE
indicates the total weight of the minimum linear assignment.

CPU_TIME
indicates the CPU time (in seconds) taken by the solver.

REAL_TIME
indicates the real time (in seconds) taken by the solver.

Macro Variable _OROPTNET_MCF_

The OPTNET procedure defines a macro variable named _OROPTNET_MCF_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm that PROC OPTNET uses to solve the minimum-cost network flow problem. The various terms of the variable are interpreted as follows:

STATUS
indicates the status of the solver at termination. The STATUS term takes the same value as the term MCF in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 97.

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

CPU_TIME
indicates the CPU time (in seconds) taken by the solver.

REAL_TIME
indicates the real time (in seconds) taken by the solver.

Macro Variable _OROPTNET_MINCUT_

The OPTNET procedure defines a macro variable named _OROPTNET_MINCUT_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm that PROC OPTNET uses to find the minimum cut. The various terms of the variable are interpreted as follows:

STATUS
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term MINCUT in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 97.

OBJECTIVE
indicates the total link weight of the minimum cut.

CPU_TIME
indicates the CPU time (in seconds) taken by the algorithm.

REAL_TIME
indicates the real time (in seconds) taken by the algorithm.

Macro Variable _OROPTNET_MST_

The OPTNET procedure defines a macro variable named _OROPTNET_MST_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm that PROC OPTNET uses to solve the minimum spanning tree problem. The various terms of the variable are interpreted as follows:

STATUS
indicates the status of the solver at termination. The STATUS term takes the same value as the term MST in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 97.

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

CPU_TIME
indicates the CPU time (in seconds) taken by the solver.

REAL_TIME
indicates the real time (in seconds) taken by the solver.

Macro Variable _OROPTNET_SHORTPATH_

The OPTNET procedure defines a macro variable named _OROPTNET_SHORTPATH_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm that PROC OPTNET uses to calculate shortest paths. The various terms of the variable are interpreted as follows:

STATUS
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term SHORTPATH in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 97.

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

CPU_TIME
indicates the CPU time (in seconds) taken by the algorithm.

REAL_TIME
indicates the real time (in seconds) taken by the algorithm.

Macro Variable _OROPTNET_TRANSCL_

The OPTNET procedure defines a macro variable named _OROPTNET_TRANSCL_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm that PROC OPTNET uses to calculate transitive closure. The various terms of the variable are interpreted as follows:

STATUS
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term TRANSCL in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 97.

CPU_TIME
indicates the CPU time (in seconds) taken by the algorithm.

REAL_TIME
indicates the real time (in seconds) taken by the algorithm.

Macro Variable _OROPTNET_TSP_

The OPTNET procedure defines a macro variable named _OROPTNET_TSP_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm that PROC OPTNET uses to solve the traveling salesman problem. The various terms of the variable are interpreted as follows:

STATUS
indicates the status of the solver at termination. The STATUS term takes the same value as the term TSP in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 97.

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

RELATIVE_GAP
specifies 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

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

ABSOLUTE_GAP
specifies 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 are returned by the solver.

BEST_BOUND
specifies 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
specifies the number of nodes enumerated by the solver by using the branch-and-bound algorithm.

ITERATIONS
indicates the number of simplex iterations taken to solve the problem.

CPU_TIME
indicates the CPU time (in seconds) taken by the algorithm.

REAL_TIME
indicates the real time (in seconds) taken by the algorithm.

NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time (default) or real time. The type is determined by the TIMETYPE= option.

ODS Table Names

Each table that the OPTNET procedure creates has a name associated with it, and you must use this name to refer to the table when you use ODS statements. These names are listed in Table 2.34.

Table 2.34 ODS Tables Produced by PROC OPTNET

Table Name	Description	Required Statement or Option
PerformanceInfo	Information about the computing environment	Default output
ProblemSummary	Summary of the graph (or matrix) input	Default output
SolutionSummary	For each algorithm, summary of the solution status	Default output
Timing	Detailed real times for each phase of the procedure	PERFORMANCE with DETAILS option

The following code uses the example in the section "Traveling Salesman Problem Applied to a Simple Undirected Graph" on page 90 and calculates both an optimal traveling salesman tour and a minimum spanning tree. This code produces all four ODS output tables listed in Table 2.34.

```
data LinkSetIn;
   input from $ to $ weight @@;
   datalines;
A B 1.0 A C 1.0 A D 1.5 B C 2.0 B D 4.0
B E 3.0 C D 3.0 C F 3.0 C H 4.0 D E 1.5
D F 3.0 D G 4.0 E F 1.0 E G 1.0 F G 2.0
F H 4.0 H I 3.0 I J 1.0 C J 5.0 F J 3.0
F I 1.0 H J 1.0
;

proc optnet
   loglevel   = moderate
   data_links = LinkSetIn
   out_nodes  = NodeSetOut;
   mst
     out      = MST;
   tsp
     out      = TSP;
run;
%put &_OROPTNET_;
%put &_OROPTNET_TSP_;
%put &_OROPTNET_MST_;
```

The problem summary table in Figure 2.79 provides a basic summary of the graph (or matrix) input.

Figure 2.79 Problem Summary Table

The OPTNET Procedure

Problem Summary	
Input Type	Graph
Number of Nodes	10
Number of Links	22
Graph Direction	Undirected

The solution summary tables in Figure 2.80 and Figure 2.81 provide a basic solution summary for each

algorithm that is processed. The information in these tables matches the information that is provided in the macro variables for each algorithm, described in the section "Macro Variables" on page 97.

Figure 2.80 Solution Summary Table for MST

Solution Summary	
Problem Type	Minimum Spanning Tree
Solution Status	Optimal
Objective Value	10
CPU Time	0.00
Real Time	0.00

Figure 2.81 Solution Summary Table for TSP

Solution Summary	
Problem Type	Traveling Salesman Problem
Solution Status	Optimal
Objective Value	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	0.00
Real Time	0.01

Examples: OPTNET Procedure

Example 2.1: 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 2.82 shows this network, which was constructed after the attacks, based on collected intelligence information.

Figure 2.82 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 BICONCOMP statement in this context:

```
data 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 optnet
   data_links = LinkSetInTerror911
   out_nodes  = NodeSetOut;
   biconcomp;
run;

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

The data set ArtPoints contains members of the network who are articulation points. Focusing investigations on cutting off these particular members could have caused a great deal of disruption in the terrorists' ability to communicate when formulating the attack.

Output 2.1.1 Articulation Points of Terrorist Communications Network from 9/11

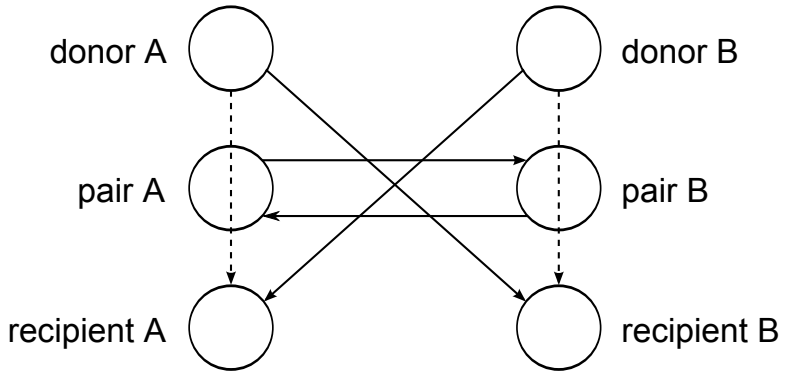
node	artpoint
Djamal Beghal	1
Zacarias Moussaoui	1
Essid Sami Ben Khemais	1
Mohamed Atta	1
Mamoun Darkazanli	1
Nawaf Alhazmi	1

Example 2.2: Cycle Detection for Kidney Donor Exchange

This example looks at an application of cycle detection to help create a kidney donor exchange. Suppose someone needs a kidney transplant and a family member is willing to donate one. If the donor and recipient are incompatible (because of blood types, tissue mismatch, and so on), the transplant cannot happen. Now suppose two donor-recipient pairs A and B are in this situation, but donor A is compatible with recipient B and donor B is compatible with recipient A. Then two transplants can take place in a two-way swap, shown

graphically in Figure 2.83. More generally, an n-way swap can be performed involving n donors and n recipients (Willingham 2009).

Figure 2.83 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. The link weight is a measure of the quality of the match. By introducing dummy links whose weight is 0, you can also include altruistic donors who have no recipients or recipients who have no donors. The idea is to find a maximum-weight node-disjoint union of directed cycles. You want the union to be node-disjoint so that no kidney is donated more than once, and you want cycles so that the donor from node i gives up 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 61. But doing so would allow arbitrarily long cycles in the solution. Because of practical considerations (such as travel) and to mitigate risk, each cycle must have no more than L links. The kidney exchange problem is to find a maximum-weight node-disjoint union of short directed cycles.

One way to solve this problem is to explicitly generate all cycles whose length is at most L and then solve a set packing problem. You can use PROC OPTNET to generate the cycles and then PROC OPTMODEL (see *SAS/OR User's Guide: Mathematical Programming*) to read the PROC OPTNET output, formulate the set packing problem, call the mixed integer linear programming solver, and output the optimal solution.

The following DATA step sets up the problem, first creating a random graph on n nodes with link probability p and Uniform(0,1) weight:

```
/* create random graph on n nodes with arc probability p
   and uniform(0,1) weight */
%let n = 100;
%let p = 0.02;
data LinkSetIn;
   do from = 0 to &n - 1;
      do to = 0 to &n - 1;
         if from eq to then continue;
         else if ranuni(1) < &p then do;
            weight = ranuni(2);
            output;
         end;
      end;
   end;
run;
```

The following statements use PROC OPTNET 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 optnet
   loglevel        = moderate
   graph_direction = directed
   data_links      = LinkSetIn;
   cycle
      minLength    = 2
      maxLength    = &max_length
      out          = Cycles
      mode         = all_cycles;
run;
%put &_OROPTNET_;
%put &_OROPTNET_CYCLE_;
```

PROC OPTNET finds 224 cycles of the appropriate length, as shown in Output 2.2.1.

Output 2.2.1 Cycles for Kidney Donor Exchange PROC OPTNET Log

```
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Running OPTNET version 15.1.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The OPTNET procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: There were 194 observations read from the data set WORK.LINKSETIN.
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 input graph storage is using 0.1 MBs (peak: 0.1 MBs) of memory.
NOTE: The number of nodes in the input graph is 97.
NOTE: The number of links in the input graph is 194.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Processing cycle detection.
NOTE: The graph has 224 cycles.
NOTE: Processing cycle detection used 0.44 (cpu: 0.44) seconds.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Creating cycle data set output.
NOTE: Data output used 0.01 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The data set WORK.CYCLES has 2124 observations and 3 variables.
STATUS=OK  CYCLE=OK
STATUS=OK  NUM_CYCLES=224  CPU_TIME=0.44  REAL_TIME=0.44
```

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

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

Given the set of cycles, you can now formulate a mixed integer linear program (MILP) to maximize the total cycle weight. Let C define the set of cycles of appropriate length, N_c define the set of nodes in cycle c, A_c define the set of links in cycle c, and w_{ij} denote the link weight for link (i, j). Define a binary decision variable x_c. Set x_c to 1 if cycle c is used in the solution; otherwise, set it to 0. Then, the following MILP

defines the problem that you want to solve to maximize the quality of the kidney exchange:

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

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> ARCS;
   num weight {ARCS};
   read data LinkSetIn into ARCS=[from to] weight;
   set <num,num,num> TRIPLES;
   read data Cycles0 into TRIPLES=[c i j];
   set CYCLES = setof {<c,i,j> in TRIPLES} c;
   set ARCS_c {c in CYCLES} = setof {<(c),i,j> in TRIPLES} <i,j>;
   set NODES_c {c in CYCLES} = union {<i,j> in ARCS_c[c]} {i,j};
   set NODES = union {c in CYCLES} NODES_c[c];
   num cycle_weight {c in CYCLES} = sum {<i,j> in ARCS_c[c]} weight[i,j];

   /* UseCycle[c] = 1 if cycle c is used, 0 otherwise */
   var UseCycle {CYCLES} binary;

   /* declare objective */
   max TotalWeight
      = sum {c in CYCLES} cycle_weight[c] * UseCycle[c];

   /* each node appears in at most one cycle */
   con node_packing {i in NODES}:
      sum {c in CYCLES: i in NODES_c[c]} UseCycle[c] <= 1;

   /* call solver */
   solve with milp;

   /* output optimal solution */
   create data Solution from
      [c]={c in CYCLES: UseCycle[c].sol > 0.5} cycle_weight;
quit;
%put &_OROPTMODEL_;
```

PROC OPTMODEL solves the problem by using the mixed integer linear programming solver. As shown in Output 2.2.2, it was able to find a total weight (quality level) of 26.02.

Output 2.2.2 Cycles for Kidney Donor Exchange PROC OPTMODEL Log

```
NOTE: There were 194 observations read from the data set WORK.LINKSETIN.
NOTE: There were 1900 observations read from the data set WORK.CYCLES0.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 224 variables (0 free, 0 fixed).
NOTE: The problem has 224 binary and 0 integer variables.
NOTE: The problem has 63 linear constraints (63 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 1900 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 46 variables and 35 constraints.
NOTE: The MILP presolver removed 901 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 178 variables, 28 constraints, and 999 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
          Node   Active   Sols    BestInteger      BestBound      Gap    Time
             0        1      4     22.7780692    868.9019355   97.38%       0
             0        1      4     22.7780692     26.7803921   14.94%       0
             0        1      5     23.2747070     26.7803921   13.09%       0
             0        1      5     23.2747070     26.0966379   10.81%       0
             0        1      6     26.0202871     26.0202871    0.00%       0
             0        0      6     26.0202871     26.0202871    0.00%       0
NOTE: The MILP solver added 12 cuts with 673 cut coefficients at the root.
NOTE: Objective = 26.020287142.
NOTE: The data set WORK.SOLUTION has 6 observations and 2 variables.
STATUS=OK ALGORITHM=BAC SOLUTION_STATUS=OPTIMAL OBJECTIVE=26.020287142 RELATIVE_GAP=0
ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=2.220446E-16 BOUND_INFEASIBILITY=2.220446E-16
INTEGER_INFEASIBILITY=1.44329E-15 BEST_BOUND=26.020287142 NODES=1 SOLUTIONS_FOUND=6
ITERATIONS=98 PRESOLVE_TIME=0.01 SOLUTION_TIME=0.03
```

The data set Solution, shown in Output 2.2.3, now contains the cycles that define the best exchange and their associated weight (quality).

Output 2.2.3 Maximum Quality Solution for Kidney Donor Exchange

c	cycle_weight
12	5.84985
43	3.90015
71	5.44467
124	7.42574
222	2.28231
224	1.11757

Example 2.3: Linear Assignment Problem for Minimizing Swim Times

A swimming coach needs to assign male and female swimmers to each stroke of a medley relay team. The swimmers' best times for each stroke are stored in a SAS data set. The LINEAR_ASSIGNMENT statement evaluates the times and matches strokes and swimmers to minimize the total relay swim time.

The data are stored in matrix format, where the row identifier is the swimmer's name (variable name) and each swimming event is a column (variables: back, breast, fly, and free). In the following DATA step, the relay times are split into two categories, male and female:

```
data RelayTimes;
   input name $ sex $ back breast fly free;
   datalines;
Sue     F 35.1 36.7 28.3 36.1
Karen   F 34.6 32.6 26.9 26.2
Jan     F 31.3 33.9 27.1 31.2
Andrea  F 28.6 34.1 29.1 30.3
Carol   F 32.9 32.2 26.6 24.0
Ellen   F 27.8 32.5 27.8 27.0
Jim     M 26.3 27.6 23.5 22.4
Mike    M 29.0 24.0 27.9 25.4
Sam     M 27.2 33.8 25.2 24.1
Clayton M 27.0 29.2 23.0 21.9
;

data RelayTimesF RelayTimesM;
   set RelayTimes;
   if      sex='F' then output RelayTimesF;
   else if sex='M' then output RelayTimesM;
run;
```

The following statements solve the linear assignment problem for both male and female relay teams:

```
proc optnet
   data_matrix = RelayTimesF;
   linear_assignment
      out      = LinearAssignF
      id       = (name sex);
run;
%put &_OROPTNET_;
%put &_OROPTNET_LAP_;

proc optnet
   data_matrix = RelayTimesM;
   linear_assignment
      out      = LinearAssignM
      id       = (name sex);
run;
%put &_OROPTNET_;
%put &_OROPTNET_LAP_;
```

The progress of the two PROC OPTNET calls is shown in Output 2.3.1 and Output 2.3.2.

Output 2.3.1 PROC OPTNET Log: Linear Assignment for Female Swim Times

```
NOTE: ---------------------------------------------------------------------------------------
NOTE: Running OPTNET version 15.1.
NOTE: ---------------------------------------------------------------------------------------
NOTE: The OPTNET procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------------------
NOTE: The number of columns in the input matrix is 4.
NOTE: The number of rows in the input matrix is 6.
NOTE: The number of nonzeros in the input matrix is 24.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------
NOTE: Processing the linear assignment problem.
NOTE: Objective = 111.5.
NOTE: Processing the linear assignment problem used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------
NOTE: The data set WORK.LINEARASSIGNF has 4 observations and 4 variables.
STATUS=OK  LAP=OPTIMAL
STATUS=OPTIMAL  OBJECTIVE=111.5  CPU_TIME=0.00  REAL_TIME=0.00
```

Output 2.3.2 PROC OPTNET Log: Linear Assignment for Male Swim Times

```
NOTE: ---------------------------------------------------------------------------------------
NOTE: Running OPTNET version 15.1.
NOTE: ---------------------------------------------------------------------------------------
NOTE: The OPTNET procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------------------
NOTE: The number of columns in the input matrix is 4.
NOTE: The number of rows in the input matrix is 4.
NOTE: The number of nonzeros in the input matrix is 16.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------
NOTE: Processing the linear assignment problem.
NOTE: Objective = 96.6.
NOTE: Processing the linear assignment problem used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------
NOTE: The data set WORK.LINEARASSIGNM has 4 observations and 4 variables.
STATUS=OK  LAP=OPTIMAL
STATUS=OPTIMAL  OBJECTIVE=96.6  CPU_TIME=0.00  REAL_TIME=0.00
```

The data sets LinearAssignF and LinearAssignM contain the optimal assignments. Note that in the case of the female data, there are more people (set S) than there are strokes (set T). Therefore, the solver allows for some members of S to remain unassigned.

Output 2.3.3 Optimal Assignments for Best Female Swim Times

name	sex	assign	cost
Karen	F	breast	32.6
Jan	F	fly	27.1
Carol	F	free	24.0
Ellen	F	back	27.8
			111.5

Output 2.3.4 Optimal Assignments for Best Male Swim Times

name	sex	assign	cost
Jim	M	free	22.4
Mike	M	breast	24.0
Sam	M	back	27.2
Clayton	M	fly	23.0
			96.6

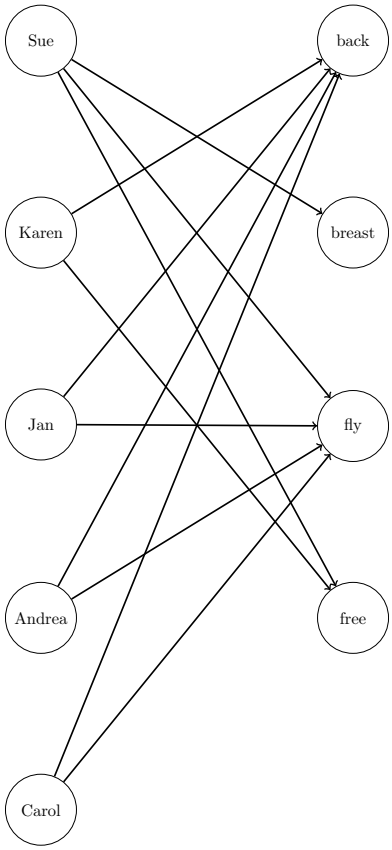
Example 2.4: Linear Assignment Problem, Sparse Format versus Dense Format

This example looks at the problem of assigning swimmers to strokes based on their best times. However, in this case certain swimmers are not eligible to perform certain 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    .
;
```

Recall that the linear assignment problem can also 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 2.84.

Figure 2.84 Bipartite Graph for Linear Assignment Problem



You can represent the same data in RelayTimesMatrix by using a links data set as follows:

```
data RelayTimesLinks;
   input name $ attr $ cost;
   datalines;
Sue     breast 36.7
Sue     fly    28.3
Sue     free   36.1
Karen   back   34.6
Karen   free   26.2
Jan     back   31.3
Jan     fly    27.1
Andrea  back   28.6
Andrea  fly    29.1
Carol   back   32.9
Carol   fly    26.6
;
```

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

Now, you can use either input format to solve the same problem, as follows:

```
proc optnet
   data_matrix = RelayTimesMatrix;
   linear_assignment
      out      = LinearAssignMatrix
      weight   = (back--free)
      id       = (name sex);
run;

proc optnet
   graph_direction = directed
   data_links      = RelayTimesLinks;
   data_links_var
      from         = name
      to           = attr
      weight       = cost;
   linear_assignment
      out          = LinearAssignLinks;
run;
```

When you use the graph input format, the LINEAR_ASSIGNMENT options WEIGHT= and ID= are not used directly.

The data sets LinearAssignMatrix and LinearAssignLinks now contain the optimal assignments, as shown in Output 2.4.1 and Output 2.4.2.

Output 2.4.1 Optimal Assignments for Swim Times (Dense Input)

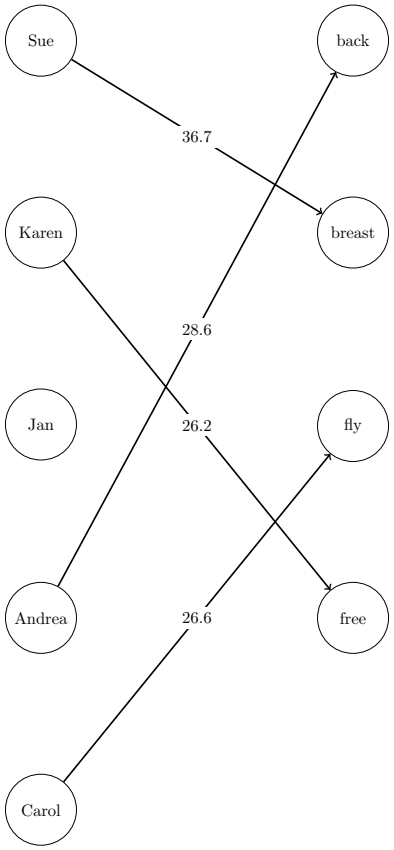
name	sex	assign	cost
Sue	F	breast	36.7
Karen	F	free	26.2
Andrea	F	back	28.6
Carol	F	fly	26.6
			118.1

Output 2.4.2 Optimal Assignments for Swim Times (Sparse Input)

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

The optimal assignments are shown graphically in Figure 2.85.

Figure 2.85 Optimal Assignments for Swim Times

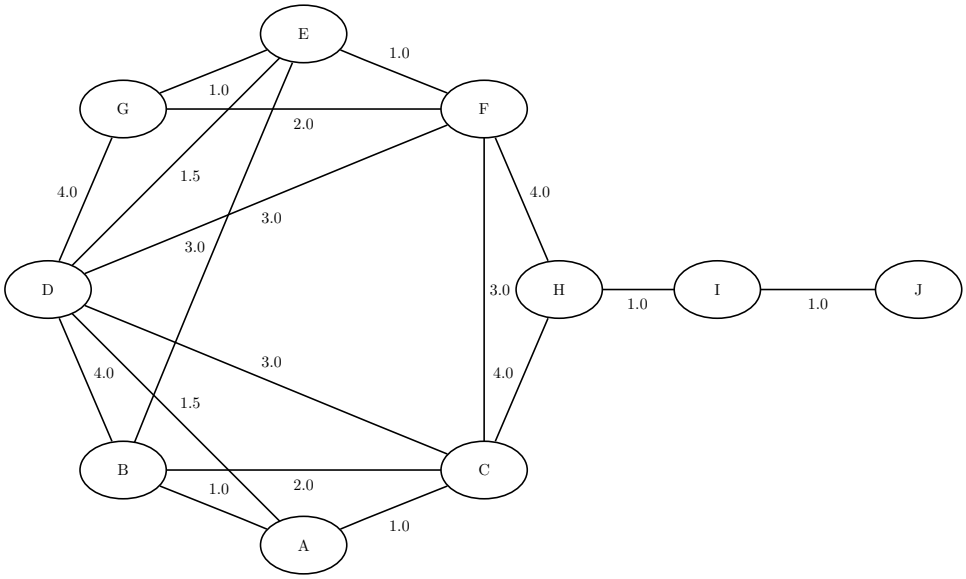


For large problems where a number of links are forbidden, the sparse format can be faster and can save a great deal of memory. Consider an example that uses the format of the DATA_MATRIX= option with 15,000 columns ($|S| = 15{,}000$) and 4,000 rows ($|T| = 4{,}000$). To store the dense matrix in memory, PROC OPTNET needs to allocate approximately $|S| \cdot |T| \cdot 8/1{,}024/1{,}024 = 457$ MB. If the data have mostly ineligible links, then the sparse (graph) format that uses the DATA_LINKS= option is much more efficient with respect to memory. For example, if the data have only 5% of the eligible links ($15{,}000 \cdot 4{,}000 \cdot 0.05 = 3{,}000{,}000$), then the dense storage would still need 457 MB. The sparse storage for the same example needs approximately $|S| \cdot |T| \cdot 0.05 \cdot 12/1{,}024/1{,}024 = 34$ MB. If the problem is fully dense (all links are eligible), then the dense format that uses the DATA_MATRIX= option is the most efficient.

Example 2.5: Minimum Spanning Tree for Computer Network Topology

Consider the problem of designing a small network of computers in an office. In designing the network, the goal is to make sure that each machine in the office can reach every other machine. To accomplish this goal, Ethernet lines must be constructed and run between the machines. The construction costs for each possible link are based approximately on distance and are shown in Figure 2.86. Besides distance, the costs also reflect some restrictions due to physical boundaries. To connect all the machines in the office at minimal cost, you need to find a minimum spanning tree on the network of possible links.

Figure 2.86 Potential Office Computer Network



Define the link data set as follows:

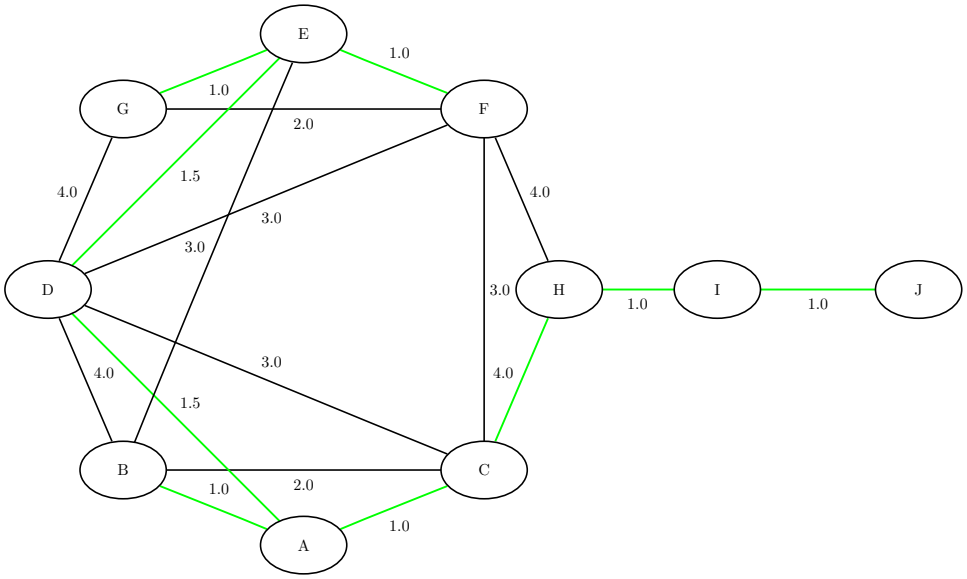
```
data LinkSetInCompNet;
   input from $ to $ weight @@;
   datalines;
A B 1.0  A C 1.0  A D 1.5  B C 2.0  B D 4.0
B E 3.0  C D 3.0  C F 3.0  C H 4.0  D E 1.5
D F 3.0  D G 4.0  E F 1.0  E G 1.0  F G 2.0
F H 4.0  H I 1.0  I J 1.0
;
```

The following statements find a minimum spanning tree:

```
proc optnet
   data_links = LinkSetInCompNet;
   minspantree
      out     = MinSpanTree;
run;
```

Output 2.5.1 shows the resulting data set MinSpanTree, which is displayed graphically in Figure 2.87 with the minimal cost links shown in green.

Figure 2.87 Minimum Spanning Tree for Office Computer Network



Output 2.5.1 Minimum Spanning Tree of a Computer Network Topology

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

Example 2.6: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System

Most software bug tracking systems have some notion of *duplicate bugs* in which one bug is declared to be the same as another bug. If bug A is considered a duplicate (DUP) of bug B, then a fix for B would also fix A. You can represent the DUPs in a bug tracking system as a directed graph where you add a link $A \rightarrow B$ if A is a DUP of B.

The bug tracking system needs to check for two situations when users declare a bug to be a DUP. The first situation is called a *circular dependence*. Consider bugs A, B, C, and D in the tracking system. The first user declares that A is a DUP of B and that C is a DUP of D. Then, a second user declares that B is a DUP of C, and a third user declares that D is a DUP of A. You now have a circular dependence, and no primary bug is defined on which the development team should focus. You can easily see this circular dependence in the graph representation, because $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$. Finding such circular dependencies can be done

using cycle detection, which is described in the section "Cycle" on page 55. However, the second situation that needs to be checked is more general. If a user declares that A is a DUP of B and another user declares that B is a DUP of C, this chain of duplicates is already an issue. The bug tracking system needs to provide one primary bug to which the rest of the bugs are duplicated. The existence of these chains can be identified by calculating the transitive closure of the directed graph that is defined by the DUP links.

Given the original directed graph G (defined by the DUP links) and its transitive closure G^T, any link in G^T that is not in G exists because of some chain that is present in G.

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

```
data DefectLinks;
   input defectId $ linkedDefect $ linkType $ when datetime16.;
   format when datetime16.;
   datalines;
D0096978 S0711218 DUPTO 20OCT10:00:00:00
S0152674 S0153280 DUPTO 30MAY02:00:00:00
S0153280 S0153307 DUPTO 30MAY02:00:00:00
S0153307 S0152674 DUPTO 30MAY02:00:00:00
S0162973 S0162978 DUPTO 29NOV10:16:13:16
S0162978 S0165405 DUPTO 29NOV10:16:13:16
S0325026 S0575748 DUPTO 01JUN10:00:00:00
S0347945 S0346582 DUPTO 03MAR06:00:00:00
S0350596 S0346582 DUPTO 21MAR06:00:00:00
S0539744 S0643230 DUPTO 10MAY10:00:00:00
S0575748 S0643230 DUPTO 15JUN10:00:00:00
S0629984 S0643230 DUPTO 01JUN10:00:00:00
;
```

The following statements calculate cycles in addition to the transitive closure of the graph G that is defined by the duplicated defects in DefectLinks. The output data set Cycles contains any circular dependencies, and the data set TransClosure contains the transitive closure G^T. To identify the chains, you can use PROC SQL to identify the links in G^T that are not in G.

```
proc optnet
   loglevel        = moderate
   graph_direction = directed
   data_links      = DefectLinks;
   data_links_var
      from         = defectId
      to           = linkedDefect;
   cycle
      out          = Cycles
      mode         = all_cycles;
   transitive_closure
      out          = TransClosure;
run;
%put &_OROPTNET_;
%put &_OROPTNET_CYCLE_;
%put &_OROPTNET_TRANSCL_;

proc sql;
   create table Chains as
```

```
    select defectId, linkedDefect from TransClosure
       except
    select defectId, linkedDefect from DefectLinks;
quit;
```

The progress of the procedure is shown in Output 2.6.1.

Output 2.6.1 PROC OPTNET Log: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System

```
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Running OPTNET version 15.1.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The OPTNET procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: There were 12 observations read from the data set WORK.DEFECTLINKS.
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 input graph storage is using 0.0 MBs (peak: 0.0 MBs) of memory.
NOTE: The number of nodes in the input graph is 16.
NOTE: The number of links in the input graph is 12.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Processing cycle detection.
NOTE: The graph has 1 cycle.
NOTE: Processing cycle detection used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Processing the transitive closure.
NOTE: Processing the transitive closure used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Creating cycle data set output.
NOTE: Creating transitive closure data set output.
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The data set WORK.CYCLES has 4 observations and 3 variables.
NOTE: The data set WORK.TRANSCLOSURE has 20 observations and 2 variables.
STATUS=OK  CYCLE=OK  TRANSCL=OK
STATUS=OK  NUM_CYCLES=1  CPU_TIME=0.00  REAL_TIME=0.00
STATUS=OK  CPU_TIME=0.00  REAL_TIME=0.00
NOTE: Table WORK.CHAINS created, with 8 rows and 2 columns.
```

The data set Cycles contains one case of a circular dependence in which the DUPs start and end at S0152674.

Output 2.6.2 Cycle in Bug Tracking System

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

The data set Chains contains the chains in the bug tracking system that come from the links in G^T that are not in G.

Output 2.6.3 Chains in Bug Tracking System

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

Example 2.7: Traveling Salesman Tour through US Capital Cities

Consider a cross-country trip where you want to travel the fewest miles to visit all of the capital cities in all US states except Alaska and Hawaii. Finding the optimal route is an instance of the traveling salesman problem, which is described in section "Traveling Salesman Problem" on page 89.

The following PROC SQL statements use the built-in data set maps.uscity to generate a list of the capital cities and their latitude and longitude:

```
/* Get a list of the state capital cities (with lat and long) */
proc sql;
   create table Cities as
   select unique statecode as state, city, lat, long
      from maps.uscity
      where capital='Y' and statecode not in ('AK' 'PR' 'HI');
quit;
```

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

```
/* Create a list of all the possible pairs of cities */
proc sql;
   create table CitiesDist as
   select
      a.city as city1, a.lat as lat1, a.long as long1,
      b.city as city2, b.lat as lat2, b.long as long2,
      geodist(lat1, long1, lat2, long2, 'DM') as distance
      from Cities as a, Cities as b
      where a.city < b.city;
quit;
```

The following PROC OPTNET statements find the optimal tour through each of the capital cities:

```
/* Find optimal tour using OPTNET */
proc optnet
   loglevel   = moderate
   data_links = CitiesDist
   out_nodes  = TSPTourNodes;
   data_links_var
      from    = city1
      to      = city2
      weight  = distance;
   tsp
      out     = TSPTourLinks;
run;
%put &_OROPTNET_;
%put &_OROPTNET_TSP_;
```

The progress of the procedure is shown in Output 2.7.1. The total mileage needed to optimally traverse the capital cities is 10,627.75 miles.

Output 2.7.1 PROC OPTNET Log: Traveling Salesman Tour through US Capital Cities

```
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Running OPTNET version 15.1.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: The OPTNET procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: There were 1176 observations read from the data set WORK.CITIESDIST.
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 input graph storage is using 0.2 MBs (peak: 0.2 MBs) of memory.
NOTE: The number of nodes in the input graph is 49.
NOTE: The number of links in the input graph is 1176.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 10643.64317 using 0.08 (cpu: 0.07)
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  10643.6431703  10038.3677946    6.03%       0
             0       1       1  10643.6431703  10239.5078223    3.95%       0
             0       1       1  10643.6431703  10260.7137067    3.73%       0
             0       1       1  10643.6431703  10265.4304687    3.68%       0
             0       1       1  10643.6431703  10272.9542665    3.61%       0
             0       1       1  10643.6431703  10281.0153870    3.53%       0
             0       1       1  10643.6431703  10343.1038027    2.91%       0
             0       1       1  10643.6431703  10347.8667383    2.86%       0
             0       1       1  10643.6431703  10353.1821797    2.81%       0
             0       1       1  10643.6431703  10379.2348305    2.55%       0
             0       1       1  10643.6431703  10479.0050603    1.57%       0
             0       1       1  10643.6431703  10557.9265173    0.81%       0
             0       1       1  10643.6431703  10573.8216736    0.66%       0
             0       1       1  10643.6431703  10609.8944131    0.32%       0
             0       1       1  10643.6431703  10617.4242796    0.25%       0
             0       1       2  10625.4826179  10625.4826179    0.00%       0
             0       0       2  10625.4826179  10625.4826179    0.00%       0
NOTE: The MILP solver added 16 cuts with 4818 cut coefficients at the root.
NOTE: Objective = 10625.482618.
NOTE: Processing the traveling salesman problem used 0.11 (cpu: 0.09) seconds.
NOTE: ---------------------------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------------------------
NOTE: Creating nodes data set output.
NOTE: Creating traveling salesman data set output.
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------------------------
```

Output 2.7.1 *continued*

```
NOTE: ---------------------------------------------------------------------------------------
NOTE: The data set WORK.TSPTOURNODES has 49 observations and 2 variables.
NOTE: The data set WORK.TSPTOURLINKS has 49 observations and 3 variables.
STATUS=OK  TSP=OPTIMAL
STATUS=OPTIMAL  OBJECTIVE=10625.482618  RELATIVE_GAP=0  ABSOLUTE_GAP=0  PRIMAL_INFEASIBILITY=0
BOUND_INFEASIBILITY=0  INTEGER_INFEASIBILITY=0  BEST_BOUND=10625.482618  NODES=1
ITERATIONS=179  CPU_TIME=0.09  REAL_TIME=0.11
```

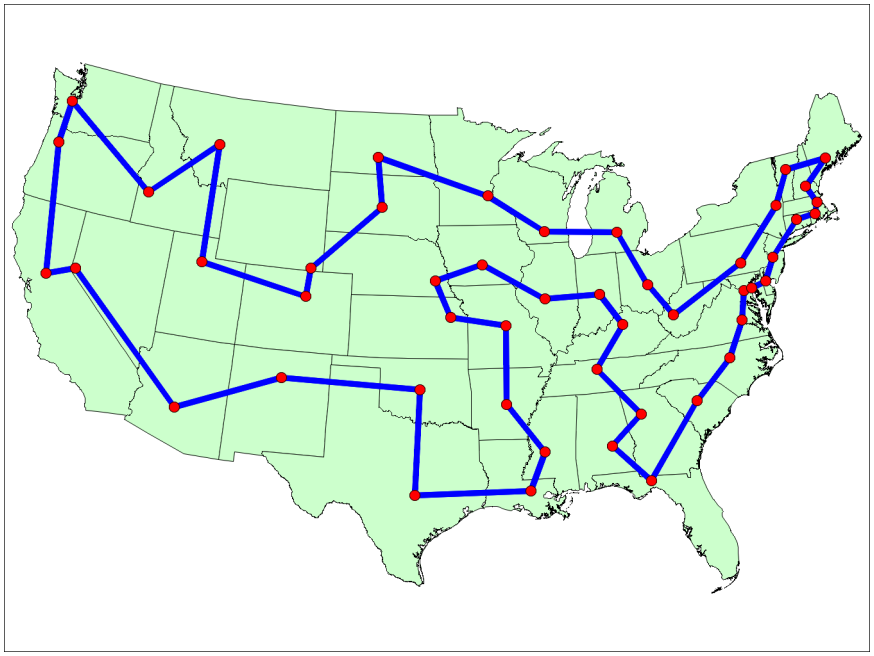
The following PROC GPROJECT and PROC GMAP statements produce a graphical display of the solution:

```
/* Merge latitude and longitude */
proc sql;
   /* merge in the lat & long for city1 */
   create table TSPTourLinksAnno1 as
   select unique TSPTourLinks.*, cities.lat as lat1, cities.long as long1
      from TSPTourLinks left join cities
      on TSPTourLinks.city1=cities.city;
   /* merge in the lat & long for city2 */
   create table TSPTourLinksAnno2 as
   select unique TSPTourLinksAnno1.*, cities.lat as lat2, cities.long as long2
      from TSPTourLinksAnno1 left join cities
      on TSPTourLinksAnno1.city2=cities.city;
quit;

/* Create the annotated data set to draw the path on the map
    (convert lat & long degrees to radians, since the map is in radians) */
data anno_path;
   set TSPTourLinksAnno2;
   length function color $8;
   xsys='2'; ysys='2'; hsys='3'; when='a'; anno_flag=1;
   function='move';
   x=atan(1)/45 * long1;
   y=atan(1)/45 * lat1;
   output;
   function='draw';
   color="blue"; size=0.8;
   x=atan(1)/45 * long2;
   y=atan(1)/45 * lat2;
   output;
run;

/* Get a map with only the contiguous 48 states */
data states;
   set maps.states (where=(fipstate(state) not in ('HI' 'AK' 'PR')));
run;

data combined;
   set states anno_path;
run;
```

```
/* Project the map and annotate the data */
proc gproject data=combined out=combined dupok;
   id state;
run;

data states anno_path;
   set combined;
   if anno_flag=1 then output anno_path;
   else                output states;
run;

/* Get a list of the endpoints locations */
proc sql;
   create table anno_dots as
   select unique x, y from anno_path;
quit;

/* Create the final annotate data set */
data anno_dots;
   set anno_dots;
   length function color $8;
   xsys='2'; ysys='2'; when='a'; hsys='3';
   function='pie';
   rotate=360; size=0.8; style='psolid'; color="red";
   output;
   style='pempty'; color="black";
   output;
run;

/* Generate the map with GMAP */
pattern1 v=s c=cxccffcc repeat=100;
proc gmap data=states map=states anno=anno_path all;
   id state;
   choro state / levels=1 nolegend coutline=black
                 anno=anno_dots des='' name="tsp";
run;
```

The minimal cost tour through the capital cities is shown on the US map in Figure 2.7.2.

Output 2.7.2 Optimal Traveling Salesman Tour through US Capital Cities



The data set TSPTourLinks contains the links in the optimal tour. To display the links in the order 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 data set TSPTourLinksDirected is shown in Figure 2.88.

Figure 2.88 Links in the Optimal Traveling Salesman Tour

City Name	City Name	distance
Montgomery	Tallahassee	177.10
Tallahassee	Columbia	311.16
Columbia	Raleigh	182.95
Raleigh	Richmond	135.56
Richmond	Washington	97.94
Washington	Annapolis	27.89
Annapolis	Dover	54.00
Dover	Trenton	83.86
Trenton	Hartford	151.62
Hartford	Providence	65.54
Providence	Boston	38.40
Boston	Concord	66.29
Concord	Augusta	117.33
Augusta	Montpelier	139.29
Montpelier	Albany	126.16
Albany	Harrisburg	230.20
Harrisburg	Charleston	287.27
Charleston	Columbus	134.62
Columbus	Lansing	205.04
Lansing	Madison	246.83
Madison	Saint Paul	226.21
Saint Paul	Bismarck	391.17
Bismarck	Pierre	170.24
Pierre	Cheyenne	317.83
Cheyenne	Denver	98.31
Denver	Salt Lake City	372.97
Salt Lake City	Helena	403.31
Helena	Boise City	291.13
Boise City	Olympia	401.23
Olympia	Salem	145.97
Salem	Sacramento	447.31
Sacramento	Carson City	101.49
Carson City	Phoenix	577.71
Phoenix	Santa Fe	378.19
Santa Fe	Oklahoma City	474.82
Oklahoma City	Austin	357.30
Austin	Baton Rouge	394.69
Baton Rouge	Jackson	139.72
Jackson	Little Rock	206.83
Little Rock	Jefferson City	264.69
Jefferson City	Topeka	191.63
Topeka	Lincoln	132.91
Lincoln	Des Moines	168.07
Des Moines	Springfield	242.97
Springfield	Indianapolis	186.42
Indianapolis	Frankfort	129.87
Frankfort	Nashville-Davidson	175.54
Nashville-Davidson	Atlanta	212.56
Atlanta	Montgomery	145.36
		10,625.48

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.

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

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

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.

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

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

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

Index