SAS® OPTGRAPH Procedure 12.3

Graph Algorithms and Network Analysis

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2013. *SAS® OPTGRAPH Procedure 12.3: Graph Algorithms and Network Analysis.* Cary, NC: SAS Institute Inc.

SAS® OPTGRAPH Procedure 12.3: Graph Algorithms and Network Analysis

SAS Institute Inc., SAS Campus Drive, Cary, North Carolina 27513.

July 2013

SAS provides a complete selection of books and electronic products to help customers use SAS® software to its fullest potential. For more information about our e-books, e-learning products, CDs, and hard-copy books, visit **support.sas.com/bookstore** or call 1-800-727-3228.

Contents

Credits

Documentation

Writing	Matthew Galati, Yi Liao
Editing	Anne Baxter, Ed Huddleston
Documentation Support	Tim Arnold, Melanie Gratton, Daniel Underwood
Technical Review	Mustafa Kabul, Charles B. Kelly, Michelle Opp, Rob Pratt

Software

PROC OPTGRAPH	Matthew Galati, Yi Liao

Support Groups

Software Testing	Mustafa Kabul, Charles B. Kelly, Rob Pratt
Technical Support	Tonya Chapman

Chapter 1
The OPTGRAPH Procedure

Contents

Overview: OPTGRAPH Procedure

The OPTGRAPH procedure includes a number of graph theory, combinatorial optimization, and network analysis algorithms. The algorithm classes are listed in Table 1.1.

Table 1.1 Algorithm Classes in PROC OPTGRAPH

Algorithm Class	PROC OPTGRAPH Statement
Biconnected components	BICONCOMP
Centrality metrics	CENTRALITY
Maximal cliques	CLIQUE
Community detection	COMMUNITY
Connected components	CONCOMP
Core decomposition	CORE
Cycle detection	CYCLE
Eigenvector problem	EIGENVECTOR
Weighted matching	LINEAR_ASSIGNMENT
Minimum-cost network flow	MINCOSTFLOW
Minimum cut (experimental)	MINCUT
Minimum spanning tree	MINSPANTREE
Reach networks	REACH
Shortest path	SHORTPATH
Graph summary	SUMMARY
Transitive closure	TRANSITIVE_CLOSURE
Traveling salesman	TSP

The OPTGRAPH procedure can be used 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 when describing an entity. The term *arc* is often interchanged with the term *edge* or *link* when describing a connection.

This document relates to PROC OPTGRAPH 12.3, which is the most recent release available for SAS 9.4. You can check the SAS log for the version number being used in any invocation of PROC OPTGRAPH.

The following statements check the version:

```
proc optgraph;
run;
```

Then the log displays the version number as shown in Figure 1.1.

Figure 1.1 Version Number Displayed in Log

```
NOTE: ---------------------------------------------------------------------
NOTE: Running OPTGRAPH version 12.3.
NOTE: ---------------------------------------------------------------------
NOTE: The OPTGRAPH procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------
```

Getting Started: OPTGRAPH 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 two introductory examples for getting started with the OPTGRAPH procedure. For more detail about the input formats expected and the various algorithms available, see the sections "Details: OPTGRAPH Procedure" on page 48 and "Examples: OPTGRAPH Procedure" on page 180.

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. With 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 OPTGRAPH, you want to find the route that yields the shortest path between home (614CapitalBlvd) and the SAS headquarters (SASCampusDrive). This can be done with the SHORTPATH statement as follows:

```
proc optgraph
   data_links    = LinkSetInRoadNC10am;
   data_links_var
      from       = start_inter
```

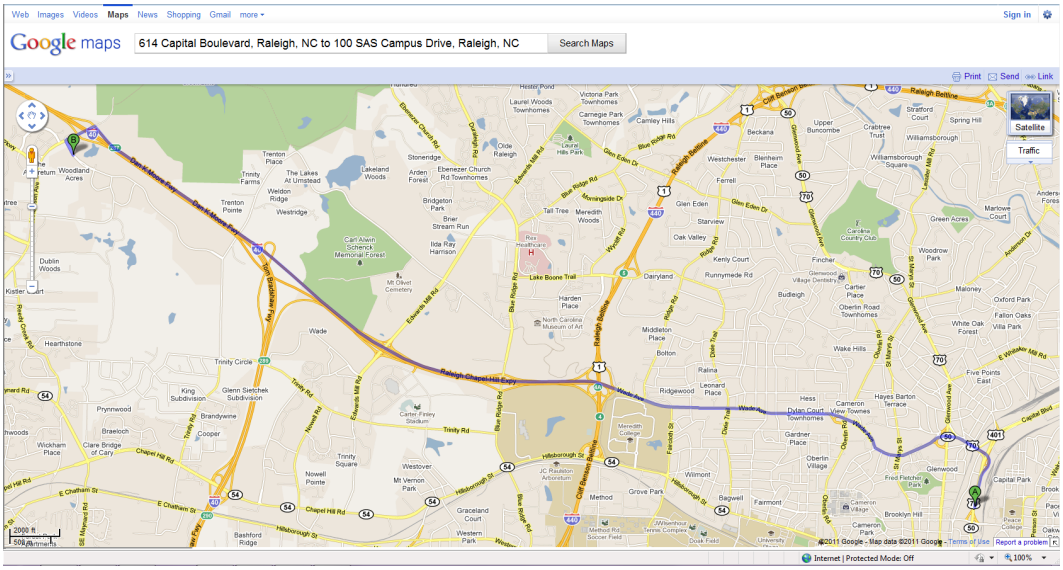
```
      to          = end_inter
      weight      = time_to_travel;
   shortpath
      out_paths   = ShortPath
      source      = "614CapitalBlvd"
      sink        = "SASCampusDrive";
run;
```

For more details about shortest path algorithms in PROC OPTGRAPH, see the section "Shortest Path" on page 143. Figure 1.2 displays the output data set ShortPath, which gives the best route to take to minimize travel time at 10:00 a.m. This route is also shown in Google Maps in Figure 1.3.

Figure 1.2 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 1.3 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 due to traffic patterns. You want to find the route that gives 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 OPTGRAPH run, except that they use the LinkSetInRoadNC5pm data set and the SOURCE and SINK option values are reversed:

```
proc optgraph
   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;
```

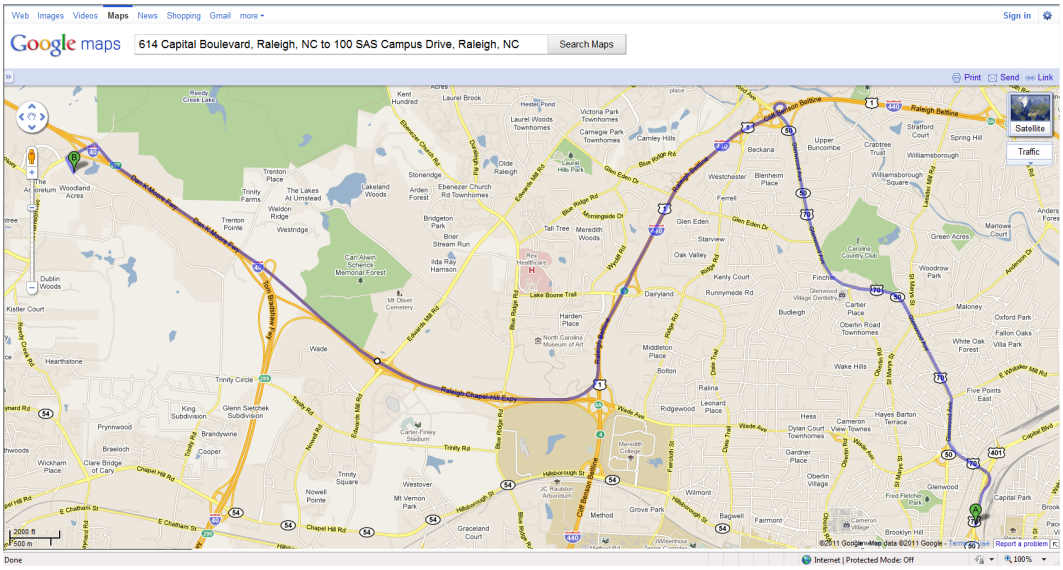
Now, the output data set ShortPath, shown in Figure 1.4, shows the best route for going home. Since the traffic on Wade Avenue is typically heavy at this time of day, the route home is different from the route to work.

Figure 1.4 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 1.5.

Figure 1.5 Shortest Path for Road Network at 5:00 P.M. in Google Maps

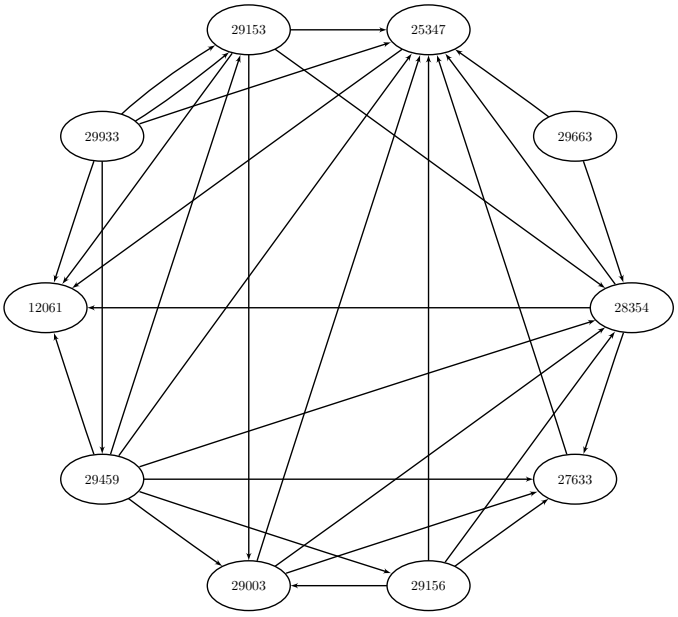


Authority in U.S. Supreme Court Precedence

This example looks at the use of precedents in court cases. Consider the judge's problem of identifying precedent court cases that are most relevant and important to the current case. This application of network analysis was published in Fowler and Joen 2008. Because of norms inherited from 19th century English law, judges are encouraged to follow precedent in order to take advantage of "accumulated experience of many judges responding to the arguments and evidence of many lawyers" (Landes and Posner 1976). In network analysis, one way to define the importance of a previous case is to look at the network of citations used in related cases. That is, if a particular case A cited case B to help support its argument, then a link exists from A to B in the citation network.

Given such a citation network, you can then use a metric known as *authority score* to rank the importance of these cases. This metric is explained in more detail in the section "Hub and Authority Scoring" on page 74. Figure 1.6 shows a small representative subset of the citation network for landmark abortion decisions from the example in Fowler and Joen 2008.

Figure 1.6 Citation Network for Some U.S. Supreme Court Cases



The data set Cases stores a mapping between case name and the case identifier:

```
data Cases;
   length case_id 8 case_name $80;
   input case_id 1-5 case_name $ 7-80;
   datalines;
12061 Jacobson v. Massachusetts, 197 U.S. 11 (1905)
25347 Roe vs. Wade, 410 U.S. 113 (1973)
27633 Akron vs. Akron Cntr for Repro-Health, 462 U.S. 416 (1983)
28354 Thornburgh vs. American College, 476 U.S. 747 (1986)
29003 Webster vs. Repro-Health Services, 492 U.S. 490 (1989)
29153 Cruzan v. Director, MO Dept of Health, 497 U.S. 261 (1990)
29155 Georgia v. South Carolina, 497 U.S. 376 (1990)
29156 Hodgson v. Minnesota, 497 U.S. 417 (1990)
29459 Planned Parenthood of SE PA vs. Casey, 505 U.S. 833 (1992)
29663 Madsen v. Women's Health Ctr., 512 U.S. 753 (1994)
29933 Wash. v. Glucksberg, 521 U.S. 702 (1997)
;
```

The data set LinkSetInCourt provides the citation network between case identifiers:

```
data LinkSetInCourt;
   input from_case to_case @@;
   datalines;
27633 25347  28354 25347  28354 27633  29003 25347  29003 27633
29003 28354  29459 25347  29459 27633  29459 28354  29459 29003
25347 12061  28354 12061  29459 12061  29933 25347  29933 29459
29933 12061  29933 29153  29663 25347  29663 28354  29153 12061
29153 28354  29153 29003  29153 25347  29459 29153  29933 29153
29156 27633  29156 28354  29156 29003  29156 25347  29459 29156
;
```

You can calculate the authority scores of each case by using the CENTRALITY statement with the AUTH= option, as follows:

```
proc optgraph
   direction      = directed
   data_links     = LinkSetInCourt
   out_nodes      = NodeSetOut;
   data_links_var
      from        = from_case
      to          = to_case;
   centrality
      auth        = unweight;
run;
```

The output data set NodeSetOut contains the authority score for each case (node). Then, the following DATA step combines the case names with the case identifiers and sorts on the score:

```
data NodeSetOut(drop=rc case_id);
   if _n_=1 then do;
       declare hash h(dataset:'cases');
       h.definekey('case_id');
       h.definedata('case_name');
       h.definedone();
   end;
   set NodeSetOut;
   length case_id 8 case_name $80;
   rc=h.find(key:node);
run;

proc sort data=NodeSetOut;
   by descending centr_auth_unwt;
run;
```

As expected, *Roe vs. Wade (1973)* has the highest authority ranking since it is most often cited by other cases.

Figure 1.7 Authority Ranking of Landmark U.S. Supreme Court Cases

node	centr_ auth_ unwt	case_name
25347	1.00000	Roe vs. Wade, 410 U.S. 113 (1973)
28354	0.72262	Thornburgh vs. American College, 476 U.S. 747 (1986)
12061	0.61717	Jacobson v. Massachusetts, 197 U.S. 11 (1905)
27633	0.59831	Akron vs. Akron Cntr for Repro-Health, 462 U.S. 416 (1983)
29003	0.50930	Webster vs. Repro-Health Services, 492 U.S. 490 (1989)
29153	0.31742	Cruzan v. Director, MO Dept of Health, 497 U.S. 261 (1990)
29156	0.20968	Hodgson v. Minnesota, 497 U.S. 417 (1990)
29459	0.10775	Planned Parenthood of SE PA vs. Casey, 505 U.S. 833 (1992)
29933	0.00000	Wash. v. Glucksberg, 521 U.S. 702 (1997)
29663	0.00000	Madsen v. Women's Health Ctr., 512 U.S. 753 (1994)

In such a small example, it is somewhat easy to see which cases have the most influence by looking at the directed graph of citations. As discussed in Fowler and Joen 2008, the real advantage of such an analysis can be seen when examining all the citations for all 30,288 cases available in their data.

Syntax: OPTGRAPH Procedure

PROC OPTGRAPH *options* **;**

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

Algorithm Statements:
BICONCOMP *< option >* **;**
CENTRALITY *< options >* **;**
CLIQUE *< options >* **;**
COMMUNITY *< options >* **;**
CONCOMP *< option >* **;**
CORE *< options >* **;**
CYCLE *< options >* **;**
EIGENVECTOR *< options >* **;**
LINEAR_ASSIGNMENT *< options >* **;**
MINCOSTFLOW *< option >* **;**
MINCUT *< options >* **;**
MINSPANTREE *< options >* **;**
REACH *< options >* **;**
SHORTPATH *< options >* **;**
SUMMARY *< options >* **;**
TRANSITIVE_CLOSURE *< option >* **;**
TSP *< option >* **;**

Performance Statement:
PERFORMANCE *< options >* **;**

PROC OPTGRAPH 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 OPTGRAPH 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 OPTGRAPH statement is described first, and sections that describe all other statements are presented in alphabetical order.

Functional Summary

Table 1.2 summarizes the statements and options available with PROC OPTGRAPH.

Table 1.2 Functional Summary

Description	Option
PROC OPTGRAPH Options	
Input	
Specifies the link data set (as an adjacency matrix)	DATA_ADJ_MATRIX=
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 subgraph filter level	FILTER_SUBGRAPH=
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_ADJ_MATRIX_VAR	
Specifies the data set variable names for adjacency matrix	
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 cluster identifiers	CLUSTER=
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=
Algorithm Statements	
BICONCOMP Option	
Specifies the log level for biconnected components	LOGLEVEL=
CENTRALITY Options	
Calculates authority centrality and specifies the type to process	AUTH=

Table 1.2 (continued)

Description	Option
Calculates betweenness centrality and specifies the type to process	BETWEEN=
Specifies whether to normalize the betweenness calculation	BETWEEN_NORM=
Decomposes the calculations for centrality by cluster (or subgraph)	BY_CLUSTER
Calculates closeness centrality and specifies the type to process	CLOSE=
Specifies the accounting method for no paths in closeness	CLOSE_NOPATH=
Calculates the node clustering coefficients	CLUSTERING_COEF
Calculates degree centrality and specifies the type to process	DEGREE=
Calculates eigenvector centrality and specifies the type to process	EIGEN=
Specifies the algorithm to use for eigenvector calculation	EIGEN_ALGORITHM=
Specifies the maximum number of iterations for eigenvector calculation	EIGEN_MAXITER=
Calculates hub centrality and specifies the type to process	HUB=
Calculates influence centrality and specifies the type to process	INFLUENCE=
Specifies the iteration log frequency (nodes)	LOGFREQNODE=
Specifies the iteration log frequency (seconds)	LOGFREQTIME=
Specifies the log level for centrality	LOGLEVEL=
Specifies the subgraph node size to run separately	SUBSIZESWITCH=
Specifies the data set variable to use for weight2 in centrality	WEIGHT2=
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=
COMMUNITY Options	
Specifies the community detection algorithm	ALGORITHM=
Specifies the percentage of small-weight links to be removed	LINK_REMOVAL_RATIO=
Specifies the log level for community detection	LOGLEVEL=
Specifies the maximum number of iterations for community detection	MAXITER=
Specifies the output data set for between-community links	OUT_COMM_LINKS=
Specifies the output data set for community summary table	OUT_COMMUNITY=
Specifies the output data set for community level summary table	OUT_LEVEL=
Specifies the output data set for community overlap table	OUT_OVERLAP=
Specifies the random factor in the parallel label propagation algorithm	RANDOM_FACTOR=
Specifies the random seed for the parallel label propagation algorithm	RANDOM_SEED=
Applies the recursive option to break large communities	RECURSIVE
Specifies the resolution list for community detection	RESOLUTION_LIST=
Specifies the modularity tolerance value for community detection	TOLERANCE=
CONCOMP Options	
Specifies the algorithm to use for connected components	ALGORITHM=
Specifies the log level for connected components	LOGLEVEL=
CORE Options	
Specifies the type of core to process	LINKS=
Specifies the log level for the core algorithm	LOGLEVEL=

Table 1.2 (continued)

Description	Option
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=
EIGENVECTOR Options	
Specifies the algebraic type of eigenvalues to calculate	EIGENVALUES=
Specifies the log level for eigenvector calculations	LOGLEVEL=
Specifies the maximum number of iterations for eigenvector calculation	MAXITER=
Specifies the number of eigenvectors to calculate	NEIGEN=
Specifies the output data set for one or more eigenvectors	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 (Experimental)	
Specifies the log level for the minimum cut algorithm	LOGLEVEL=
Specifies the maximum number of cuts to return from the algorithm	MAXNUMCUTS=
Specifies the maximum weight of the cuts to return from the algorithm	MAXWEIGHT=
Specifies the output data set for minimum cut	OUT=
MINSPANTREE Options	
Specifies the log level for the minimum spanning tree algorithm	LOGLEVEL=
Specifies the output data set for minimum spanning tree	OUT=
REACH Options	
Decomposes the calculations for reach by cluster (or subgraph)	BY_CLUSTER
Calculates the directed reach counts	DIGRAPH
Treats each node as a source in reach calculations	EACH_SOURCE

Table 1.2 (continued)

Description	Option
Ignores the source node in reach counts	IGNORE_SELF
Specifies the maximum number of links to allow in the reach calculations	MAXREACH=
Specifies the iteration log frequency (seconds)	LOGFREQTIME=
Specifies the log level for reach calculations	LOGLEVEL=
Specifies the output data set for reach counts	OUT_COUNTS=
Specifies the output data set for reach counts (limit=1)	OUT_COUNTS1=
Specifies the output data set for reach counts (limit=2)	OUT_COUNTS2=
Specifies the output data set for reach links	OUT_LINKS=
Specifies the output data set for reach nodes	OUT_NODES=
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=
SUMMARY Options	
Calculates information about biconnected components	BICONCOMP
Decomposes the calculations for summary by cluster (or subgraph)	BY_CLUSTER
Calculates information about connected components	CONCOMP
Calculates the approximate diameter and chooses the weight type	DIAMETER_APPROX=
Specifies the iteration log frequency (nodes)	LOGFREQNODE=
Specifies the iteration log frequency (seconds)	LOGFREQTIME=
Specifies the log level for summary calculations	LOGLEVEL=
Specifies the output data set for summary results	OUT=
Calculates information about shortest paths and chooses the weight type	SHORTPATH=
Specifies the subgraph node size to run separately	SUBSIZESWITCH=
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 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 maximum allowed difference between an integer variable's value and an integer	INTTOL=
Specifies the frequency of printing the branch-and-bound node log	LOGFREQ=

Table 1.2 (continued)

Description	Option
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	OUT=
Specifies the stopping criterion that is based on 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 37.

Table 1.3 lists the valid input formats, GRAPH_DIRECTION= values, and GRAPH_INTERNAL_FORMAT= values for each statement in the OPTGRAPH procedure.

Table 1.3 Supported Input Formats and Graph Types by Statement

	Input Format		DIRECTION		INTERNAL_FORMAT	
Statement	Graph	Matrix	UNDIRECTED	DIRECTED	THIN	FULL
BICONCOMP	X		X			X
CENTRALITY						
AUTH=, HUB=	X			X		X
EIGEN=	X		X			X
BETWEEN=, CLOSE=, CLUSTERING_COEF, DEGREE=, INFLUENCE=,	X		X	X		X
CENTRALITY / BY_CLUSTER						
AUTH=, HUB=	X			X	X	X
EIGEN=	X		X		X	X
BETWEEN=, CLOSE= CLUSTERING_COEF, DEGREE=, INFLUENCE=,	X		X	X	X	X
CLIQUE	X		X			X
COMMUNITY						
ALGORITHM=						
LOUVAIN, LABEL_PROP	X		X		X	X
PARALLEL_LABEL_PROP	X		X	X	X	X

Table 1.3 (continued)

	Input Format		DIRECTION		INTERNAL_FORMAT	
Statement	Graph	Matrix	UNDIRECTED	DIRECTED	THIN	FULL
CONCOMP						
ALGORITHM=						
DFS	X		X	X		X
UNION_FIND	X		X		X	X
CORE	X		X	X		X
CYCLE	X		X	X		X
EIGENVECTOR	X	X	X			X
LINEAR_ASSIGNMENT	X	X		X		X
MINCOSTFLOW	X			X	X	X
MINCUT	X		X			X
MINSPANTREE	X		X		X	X
REACH	X		X	X		X
REACH / BY_CLUSTER	X		X	X	X	X
SHORTPATH	X		X	X		X
SUMMARY	X		X	X		X
SUMMARY / BY_CLUSTER	X		X	X	X	X
TRANSITIVE_CLOSURE	X		X	X		X
TSP	X		X			X

Table 1.4 indicates for each algorithm statement in the OPTGRAPH 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 OPTGRAPH statement.

Table 1.4 Output Options by Statement

Statement	**OUT_NODES**	**OUT_LINKS**	**Algorithm Statement Options**
BICONCOMP	X	X	
CENTRALITY			
AUTH=, CLOSE=,	X		
CLUSTERING_COEF,			
DEGREE=, EIGEN=, HUB=,			
INFLUENCE=			
BETWEEN=	X	X	
CLIQUE	X		OUT=
COMMUNITY			
ALGORITHM=			
LOUVAIN, LABEL_PROP,	X		OUT_COMM_LINKS=,
PARALLEL_LABEL_PROP			OUT_COMMUNITY=,
			OUT_LEVEL=,
			OUT_OVERLAP=
CONCOMP	X		
CORE	X		
CYCLE			OUT=
EIGENVECTOR			OUT=

Table 1.4 (continued)

Statement	OUT_NODES	OUT_LINKS	Algorithm Statement Options
LINEAR_ASSIGNMENT			OUT=
MINCOSTFLOW		X	
MINCUT	X		OUT=
MINSPANTREE			OUT=
REACH BY_CLUSTER BY_CLUSTER and EACH_SOURCE			OUT_COUNTS=, OUT_LINKS=, OUT_NODES= OUT_COUNTS=, OUT_NODES= OUT_COUNTS1=, OUT_COUNTS2=
SHORTPATH			OUT_PATHS=, OUT_WEIGHTS=
SUMMARY	X		OUT=
TRANSITIVE_CLOSURE			OUT=
TSP	X		OUT=

PROC OPTGRAPH Statement

PROC OPTGRAPH < *options* > **;**

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

DATA_ADJ_MATRIX=*SAS-data-set*

ADJ_MATRIX=*SAS-data-set*

specifies the input data set that contains the graph link information, where the links are defined as an adjacency matrix.

See the section "Adjacency Matrix Input Data" on page 53 for more information.

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 49 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 57 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 54 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 55 for more information.

FILTER_SUBGRAPH=*number*

specifies the minimum number of nodes allowed in a subgraph when processing is decomposed by cluster. When the BY_CLUSTER option is also specified in another statement, any subgraph whose number of nodes is less than or equal to *number* is skipped. The default setting is 0, so nothing is filtered by default.

See the section "Graph Input Data" on page 48 for more information.

GRAPH_DIRECTION=DIRECTED | UNDIRECTED

DIRECTION=DIRECTED | UNDIRECTED

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

Table 1.5 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* (or *tail*) node, and j is called the *sink* (or *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.

See the section "Graph Input Data" on page 48 for more information.

GRAPH_INTERNAL_FORMAT=THIN | FULL

INTERNAL_FORMAT=THIN | FULL

requests the internal graph format for the algorithms to use.

Table 1.6 Values for the GRAPH_INTERNAL_FORMAT= Option

Option Value	Description
FULL	Stores the graph in standard (full) format. This is the default.
THIN	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 OPTGRAPH 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. The THIN option can often be helpful when you do calculations that are decomposed by subgraph.

See the section "Graph Input Data" on page 48 for more information.

INCLUDE_SELFLINK
includes self links—for example, (i, i)—when an input graph is read. By default, when PROC OPTGRAPH 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 1.7 describes the valid values for this option.

Table 1.7 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

The default is 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.

TIMETYPE=*number* | *string*
specifies whether CPU time or real time is used for the MAXTIME= option for each applicable algorithm. Table 1.8 describes the valid values of the TIMETYPE= option.

Table 1.8 Values for TIMETYPE= Option

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

The default is CPU.

BICONCOMP Statement

BICONCOMP < *option* > **;**

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

See the section "Biconnected Components and Articulation Points" on page 59 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 1.9 describes the valid values for this option.

Table 1.9 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 OPTGRAPH statement (or BASIC if that option is not specified).

CENTRALITY Statement

CENTRALITY < *options* > **;**

The CENTRALITY statement enables you to select which centrality metrics to calculate for the given input graph. It also enables you to specify options for particular metrics. The resulting metrics are included in the node output data set (specified in the OUT_NODES= option) or the link output data set (specified in the OUT_LINKS= option).

The centrality metrics are described in the section "Centrality" on page 63.

You can specify the following *options* in the CENTRALITY statement.

AUTH=WEIGHT | UNWEIGHT | BOTH
specifies which type of authority centrality to calculate.

Table 1.10 Values for the AUTH= Option

Option Value	**Description**
WEIGHT	Calculates authority centrality based on the weighted graph.
UNWEIGHT	Calculates authority centrality based on the unweighted graph.
BOTH	Calculates authority centrality based on both weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (using 1.0 for each link weight). This centrality metric can be used only for directed graphs. The authority centrality metric is described in the section "Hub and Authority Scoring" on page 74.

BETWEEN=WEIGHT | UNWEIGHT | BOTH
specifies which type of betweenness centrality to calculate.

Table 1.11 Values for the BETWEEN= Option

Option Value	Description
WEIGHT	Calculates betweenness centrality based on the weighted graph.
UNWEIGHT	Calculates betweenness centrality based on the unweighted graph.
BOTH	Calculates betweenness centrality based on both weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (using 1.0 for each link weight). If the OUT_NODES= option is specified in the PROC OPTGRAPH statement, the node betweenness metric is produced. If the OUT_LINKS= option is specified, the link betweenness metric is produced. The betweenness centrality metric is described in the section "Betweenness Centrality" on page 70.

BETWEEN_NORM=YES | NO

specifies whether to normalize the betweenness centrality metrics.

Table 1.12 Values for the BETWEEN_NORM= Option

Option Value	Description
YES	Normalizes the betweenness metrics. This is the default.
NO	Does not normalize the betweenness metrics.

The normalization factor for betweenness centrality is described in the section "Betweenness Centrality" on page 70.

BY_CLUSTER

decomposes the calculations by cluster (or subgraph). If this option is specified, PROC OPTGRAPH looks for a definition of the clusters in the input data set specified by the DATA_NODES= option in the PROC OPTGRAPH statement. The use of the BY_CLUSTER option is described in the section "Processing by Cluster" on page 80.

CLOSE=WEIGHT | UNWEIGHT | BOTH

specifies which type of closeness centrality to calculate.

Table 1.13 Values for the CLOSE= Option

Option Value	Description
WEIGHT	Calculates closeness centrality based on the weighted graph.
UNWEIGHT	Calculates closeness centrality based on the unweighted graph.
BOTH	Calculates closeness centrality based on both weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (using 1.0 for each link weight). The closeness centrality metric is described in the section "Closeness Centrality" on page 68.

CLOSE_NOPATH=NNODES | DIAMETER | ZERO
specifies a method for accounting for a shortest path between two nodes when a path does not exist (disconnected nodes).

Table 1.14 Values for the CLOSE_NOPATH= Option

Option Value	Description
NNODES	Uses the number of nodes as a shortest path between disconnected nodes. This is the default.
DIAMETER	Uses the graph diameter as a shortest path between disconnected nodes.
ZERO	Uses zero as a shortest path between disconnected nodes.

For each option, there is a slight variation in the formula for the closeness centrality metric. These differences are described in the section "Closeness Centrality" on page 68.

CLUSTERING_COEF
calculates the node clustering coefficient. The cluster coefficient is described in the section "Clustering Coefficient" on page 65.

DEGREE=IN | OUT | BOTH
specifies which type of degree centrality to calculate for the input graph.

Table 1.15 Values for the DEGREE= Option

Option Value	Description
IN	Calculates degree based on in-links.
OUT	Calculates degree based on out-links.
BOTH	Calculates degree based on in-links and out-links.

For an undirected graph, option values IN and BOTH are ignored, since there is only one notion of degree, which corresponds to the degree of out-links. The degree centrality metric is described in the section "Degree Centrality" on page 63.

EIGEN=WEIGHT | UNWEIGHT | BOTH
specifies which type of eigenvector centrality to calculate.

Table 1.16 Values for the EIGEN= Option

Option Value	Description
WEIGHT	Calculates eigenvector centrality based on the weighted graph.
UNWEIGHT	Calculates eigenvector centrality based on the unweighted graph.
BOTH	Calculates eigenvector centrality based on both weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (using 1.0 for each link weight). This centrality metric can be used only for undirected graphs. The eigenvector centrality metric is described in the section "Eigenvector Centrality" on page 72.

EIGEN_ALGORITHM=AUTOMATIC | JACOBI_DAVIDSON | POWER

specifies the algorithm to use in calculating centrality metrics that require solving eigensystems (EIGEN, HUB, and AUTH).

Table 1.17 Values for the EIGEN_ALGORITHM= Option

Option Value	Description
AUTOMATIC	PROC OPTGRAPH automatically determines the eigensolver to use. This is the default.
JACOBI_DAVIDSON (JD)	Uses a variant of the Jacobi-Davidson algorithm for solving eigensystems (Sleijpen and van der Vorst 2000). This is used as the default for eigenvector, hub, and authority metrics.
POWER	Uses the power method to calculate eigenvectors. This method is not supported for eigenvector centrality.

EIGEN_MAXITER=*number*

specifies the maximum number of iterations to use for eigenvector calculations to limit the amount of computation time spent when convergence is slow. The default is 10,000.

HUB=WEIGHT | UNWEIGHT | BOTH

specifies which type of hub centrality to calculate.

Table 1.18 Values for the HUB= Option

Option Value	Description
WEIGHT	Calculates hub centrality based on the weighted graph.
UNWEIGHT	Calculates hub centrality based on the unweighted graph.
BOTH	Calculates hub centrality based on both weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (using 1.0 for each link weight). This centrality metric can be used only for directed graphs. The hub centrality metric is described in the section "Hub and Authority Scoring" on page 74.

INFLUENCE=WEIGHT | UNWEIGHT | BOTH

specifies which type of influence centrality to calculate.

Table 1.19 Values for the INFLUENCE= Option

Option Value	Description
WEIGHT	Calculates influence centrality based on the weighted graph.
UNWEIGHT	Calculates influence centrality based on the unweighted graph.
BOTH	Calculates influence centrality based on both weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (using 1.0 for each link weight). The influence centrality metric is described in the section "Influence Centrality" on page 64.

LOGFREQNODE=*number*

controls the frequency for displaying iteration logs for some of the centrality metrics. For computationally intensive algorithms such as betweenness and closeness centrality, this option displays progress every *number* nodes. If you also specify the BY_CLUSTER option in this statement or a value greater than 1 for the NTHREADS= option in the PERFORMANCE statement, this option is ignored and the display frequency is determined by using the LOGFREQTIME= option instead. 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.

LOGFREQTIME=*number*

controls the frequency for displaying iteration logs for some of the centrality metrics. For computationally intensive algorithms such as betweenness and closeness centrality, this option displays progress every *number* seconds. If you specify a value greater than 1 for the NTHREADS= option in the PERFORMANCE statement, PROC OPTGRAPH displays the number of nodes that have completed. If you specify the BY_CLUSTER option, PROC OPTGRAPH displays the number of subgraphs that have completed. The value of *number* can be any integer greater than or equal to 1; the default is 5. 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 1.20 describes the valid values for this option.

Table 1.20 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 LOGFREQNODE= or LOGFREQTIME= option
3	AGGRESSIVE	Displays a detailed summary of the algorithmic processing including a progress log using the interval that is specified in the LOGFREQNODE= or LOGFREQTIME= option

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

SUBSIZESWITCH=*number*

specifies the size of the subgraphs (number of nodes) to run separately when you also specify the BY_CLUSTER option in this statement and a value greater than 1 for the NTHREADS= option in the PERFORMANCE statement. When PROC OPTGRAPH processes summary by subgraphs, it uses thread logic to simultaneously process n subgraphs, where n is the number of threads specified in the NTHREADS= option in the PERFORMANCE statement. Subgraphs that have more nodes than *number* are processed sequentially, enabling the threading to be done at the centrality metric level. The default is 10,000.

WEIGHT2=*column*
specifies the data set variable name for a second link weight. The value of *column* must be numeric. The use of this option is described in more detail in the section "Weight Interpretation" on page 76.

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 88.

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 1.21 describes the valid values for this option.

Table 1.21 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 OPTGRAPH 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.

COMMUNITY Statement

COMMUNITY < *options* > **;**

The COMMUNITY statement invokes an algorithm that detects communities of the input graph. Community detection is described in the section "Community" on page 92.

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

ALGORITHM=LOUVAIN | LABEL_PROP | PARALLEL_LABEL_PROP
specifies whether to use the Louvain algorithm (LOUVAIN), the label propagation algorithm (LABEL_PROP), or the parallel label propagation algorithm (PARALLEL_LABEL_PROP). The Louvain algorithm is the default.

For more information about this option, see the sections "Community" on page 92 and "Parallel Community Detection" on page 93.

LINK_REMOVAL_RATIO=*number*
defines the percentage of small-weight links to be removed around each node neighborhood. A link is usually removed if its weight is relatively smaller than the weights of neighboring links. Suppose that node A links to node B and to node C, link $A \rightarrow B$ has weight of 100, and link $A \rightarrow C$ has weight of 1. When nodes are grouped into communities, link $A \rightarrow B$ is much more important than link $A \rightarrow C$ because it contributes much more to the overall modularity value. Therefore, link $A \rightarrow C$ can be dropped from the network if dropping it does not disconnect node C from the network. If the LINK_REMOVAL_RATIO= option is specified, then the links that are incident to each node are examined. If the weight of any link is less than (*number*/100)**max_link_weight*, where *max_link_weight* is the maximum link weight among all links incident to this node, it is removed provided that its removal does not disconnect any node from the network. This option can often dramatically improve the running time of large graphs. The valid range is between 0 and 100. The default value is 10.

LOGLEVEL=*number* | *string*
controls the amount of information that is displayed in the SAS log. Table 1.22 describes the valid values for this option.

Table 1.22 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 you specify in the LOGLEVEL= option in the PROC OPTGRAPH statement (or BASIC if that option is not specified).

MAXITER=*number*
specifies the maximum number of iterations allowed in the algorithm. The default is 20 when ALGORITHM=LOUVAIN and 100 when ALGORITHM=LABEL_PROP or ALGORITHM=PARALLEL_LABEL_PROP.

OUT_COMM_LINKS=*SAS-data-set*
specifies the output data set that describes the links between communities.

OUT_COMMUNITY=*SAS-data-set*
specifies the output data set that contains the number of nodes in each community.

OUT_LEVEL=*SAS-data-set*
specifies the output data set that contains community information at different resolution levels.

OUT_OVERLAP=*SAS-data-set*
specifies the output data set that describes the intensity of each node that belongs to multiple communities.

RANDOM_FACTOR=*number*
specifies the random factor for the parallel label propagation algorithm. Specify a *number* between 0 and 1. At each iteration, *number* $\times$ 100% of the nodes are randomly selected to skip the label propagation step. The default is 0.15, which means that 15% of nodes skip the label propagation step at each iteration.

RANDOM_SEED=*number*
specifies the random seed for the parallel label propagation algorithm. At each iteration, some nodes are randomly selected to skip the label propagation step, based on the value that you specify in the RANDOM_FACTOR= option. To choose a different set of random samples, specify a *number* in the RANDOM_SEED= option. The default is 1234.

RECURSIVE (*options***)**
requests that the algorithm recursively break down large communities into smaller ones until the specified conditions are satisfied. This option starts with the keyword RECURSIVE followed by any combination of three suboptions enclosed in parentheses—for example, RECURSIVE (MAX_COMM_SIZE=500) or RECURSIVE (MAX_COMM_SIZE=1000 MAX_DIAMETER=3 RELATION=AND).

Table 1.23 RECURSIVE *options*

option	**Description**
MAX_COMM_SIZE=	Specifies the maximum number of nodes to be contained in any community.
MAX_DIAMETER=	Specifies the maximum number of links on the shortest paths between any pair of nodes in any community.
RELATION=	Specifies the relationship between the values of MAX_COMM_SIZE and MAX_DIAMETER options. If RELATION=AND, then recursive splitting continues until both MAX_COMM_SIZE and MAX_DIAMETER conditions are satisfied. If RELATION=OR, then recursive splitting continues until either the MAX_COMM_SIZE or the MAX_DIAMETER condition is satisfied. The valid values are AND and OR. The default is OR.

The MAX_DIAMETER= option is ignored when you specify ALGORITHM=PARALLEL_LABEL_PROP.

RESOLUTION_LIST=*num_list*

specifies a list of resolution values that are separated by spaces (for example, 4.3 2.1 1.0 0.6 0.2). The OPTGRAPH procedure interprets the RESOLUTION_LIST= option differently depending on the value of the ALGORITHM= option:

- When ALGORITHM=LOUVAIN, specifying multiple resolution values enables you to see how communities are merged at various resolution levels. A larger parameter value indicates a higher resolution. For example, resolution 4.3 produces more communities than resolution 0.2. The default is 1.0. When you also specify the RECURSIVE option, the first value in the resolution list is used and the other values are ignored.
- When ALGORITHM=LABEL_PROP, PROC OPTGRAPH ignores the RESOLUTION_LIST= option. It uses the default value of 1.0.
- When ALGORITHM=PARALLEL_LABEL_PROP, specifying multiple resolution values requests that the OPTGRAPH procedure perform community detection multiple times, each time with a different resolution value. The default is 0.001. In this case, the RESOLUTION_LIST= option is fully compatible with the RECURSIVE option.

For more information about the use of the RESOLUTION_LIST= option, see the section "Large Community" on page 95.

TOLERANCE=*number*

MODULARITY=*number*

specifies the tolerance value for when to stop iterations. When you specify ALGORITHM=LOUVAIN, the algorithm stops iterations when the percentage modularity gain between two consecutive iterations falls within the specified tolerance value. When you specify ALGORITHM=LABEL_PROP or ALGORITHM=PARALLEL_LABEL_PROP, the algorithm stops iterations when the percentage of label changes for all nodes in the graph falls within the tolerance specified by *number*. The valid range is strictly between 0 and 1. The default is 0.01.

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 100.

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

ALGORITHM=DFS | UNION_FIND

specifies the algorithm to use for calculating connected components.

Table 1.24 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 OPTGRAPH statement.

Table 1.24 (continued)

Option Value	Description
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 OPTGRAPH statement. This value can be faster than DFS when used with GRAPH_INTERNAL_FORMAT=THIN. However, you can use it only with undirected graphs.

The default is DFS.

LOGLEVEL=*number* | *string*

controls the amount of information that is displayed in the SAS log. Table 1.25 describes the valid values for this option.

Table 1.25 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 OPTGRAPH statement (or BASIC if that option is not specified).

CORE Statement

CORE < *options* > **;**

The CORE statement invokes an algorithm that finds the core decomposition of the input graph. Core decompositions are described in the section "Core Decomposition" on page 105.

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

LINKS=IN | OUT | BOTH

specifies which type of cores to calculate for a directed graph. You can choose to calculate the cores based on in-links (IN), out-links (OUT), or both (BOTH). For an undirected graph, core applies only to out-links.

Table 1.26 Values for the LINKS= Option

Option Value	Description
IN	Calculates core based on in-links.
OUT	Calculates core based on out-links. This is the default.
BOTH	Calculates core based on in-links and out-links.

LOGLEVEL=*number* | *string*
controls the amount of information that is displayed in the SAS log. Table 1.27 describes the valid values for this option.

Table 1.27 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 OPTGRAPH 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 109.

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 1.28 describes the valid values for this option.

Table 1.28 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 OPTGRAPH 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. If a cycle is found whose length is greater than *number*, that cycle is removed from the results. The default is the positive number that has the

largest absolute value representable 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. If a cycle is found whose sum of link weights is greater than *number*, that cycle is removed from the results. The default is the positive number that has the largest absolute value representable 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. If a cycle is found whose sum of node weights is greater than *number*, that cycle is removed from the results. The default is the positive number that has the largest absolute value representable 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. If a cycle is found that has fewer links than *number*, that cycle is removed from the results. The default is 1. By default, 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. If a cycle is found whose sum of link weights is less than *number*, that cycle is removed from the results. The default is the negative number that has the largest absolute value representable 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. If a cycle is found whose sum of node weights is less than *number*, that cycle is removed from the results. The default is the negative number that has the largest absolute value representable in your operating environment. By default, nothing is filtered. This option works only when you also specify MODE=ALL_CYCLES.

MODE=*option*
specifies the mode for processing cycles.

Table 1.29 Values for the MODE= Option

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

The default is FIRST_CYCLE.

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

DATA_ADJ_MATRIX_VAR Statement

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

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

The DATA_ADJ_MATRIX_VAR statement enables you to explicitly define the data set variable names for PROC OPTGRAPH to use when it reads the data set that is specified in the DATA_ADJ_MATRIX= option in the PROC OPTGRAPH statement. The format of the adjacency matrix input data set is defined in the section "Adjacency Matrix Input Data" on page 53. The value of each *column* variable must be numeric.

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 OPTGRAPH to use when it reads the data set that is specified in the DATA_LINKS= option in the PROC OPTGRAPH statement. The format of the links input data set is defined in the section "Link Input Data" on page 49.

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

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

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

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

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

WEIGHT=*column*
specifies the data set variable name for link weights. The value of *column* 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 OPTGRAPH to use when it reads the data set that is specified in the DATA_MATRIX= option in the PROC OPTGRAPH statement. The format of the matrix input data set is defined in the section "Matrix Input Data" on page 57. 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 OPTGRAPH to use when it reads the data set that is specified in the DATA_NODES= option in the PROC OPTGRAPH statement. The format of the node input data set is defined in the section "Node Input Data" on page 54.

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

CLUSTER=*column*
specifies the data set variable name for clusters identifiers. The value of *column* must be numeric.

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

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

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

EIGENVECTOR Statement

EIGENVECTOR < *options* > **;**

The EIGENVECTOR statement invokes a variant of the Jacobi-Davidson algorithm (Sleijpen and van der Vorst 2000) that finds eigenvectors (and eigenvalues) for symmetric matrices. The matrix is typically defined in the input data set that is specified in the DATA_MATRIX= option in the PROC OPTGRAPH statement. The matrix can also be input as a graph by using the DATA_LINKS= option in the PROC OPTGRAPH statement. Internally, the graph is converted into a (sparse) adjacency matrix.

Eigenvectors and eigenvalues are described in the section "Eigenvector Problem" on page 115.

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

EIGENVALUES=LA | SA
specifies the type of eigenvector to calculate. Table 1.30 describes the valid values for this option.

Table 1.30 Values for the EIGENVALUES= Option

Option Value	Description
LA	Calculates the *n* largest algebraic eigenvalues (and their corresponding eigenvectors), where *n* is the value of the NEIGEN= option. This is the default.
SA	Calculates the *n* smallest algebraic eigenvalues (and their corresponding eigenvectors), where *n* is the value of the NEIGEN= option.

LOGLEVEL=*number* | *string*
controls the amount of information that is displayed in the SAS log. Table 1.31 describes the valid values for this option.

Table 1.31 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 OPTGRAPH statement (or BASIC if that option is not specified).

MAXITER=*number*
specifies the maximum number of maxtrix-vector multiplications used in the Jacobi-Davidson algorithm to calculate eigenvectors. The default is 10,000.

NEIGEN=*number*
specifies the number of eigenvalues (and their corresponding eigenvectors) to generate. This value must be less than or equal to the dimension of the matrix. The default is 1.

OUT=*SAS-data-set*
specifies the output data set to contain the eigenvectors (and eigenvalues) found.

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 OPTGRAPH statement. The data can also be defined as a directed graph by specifying the DATA_LINKS= option in the PROC OPTGRAPH 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 118.

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 1.32 describes the valid values for this option.

Table 1.32 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 OPTGRAPH 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 125.

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 1.33 describes the valid values for this option.

Table 1.33 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 OPTGRAPH statement (or BASIC if that option is not specified).

MAXTIME=*option*
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 (Experimental)

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 119.

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 1.34 describes the valid values for this option.

Table 1.34 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 OPTGRAPH 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. The default is 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 representable 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* > **;**

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 123.

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 1.35 describes the valid values for this option.

Table 1.35 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 OPTGRAPH 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 OPTGRAPH 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 the procedure produce the Timing ODS table. This table shows a breakdown of the time used in each step of the procedure.

NTHREADS=*number* **| CPUCOUNT**
specifies the number of threads that the procedure 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 actual 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 OPTGRAPH procedure. In some circumstances, the OPTGRAPH 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 OPTGRAPH uses eight threads for the parallel label propagation algorithm:

```
proc optgraph
   data_links       = links
   graph_direction  = directed
   out_nodes        = outNodes;
   performance
      nthreads      = 8;
   community
      algorithm     = parallel_label_prop
      out_community = outComm;
run;
```

REACH Statement

REACH < *options* > **;**

The REACH statement invokes an algorithm that calculates the reach (ego) network on an input graph.

The reach network is described in the section "Reach (Ego) Network" on page 130.

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

BY_CLUSTER
decomposes the calculations by cluster (subgraph). If this option is specified, PROC OPTGRAPH looks for a definition of the clusters in the input data set specified in the DATA_NODES= option in the PROC OPTGRAPH statement. If BY_CLUSTER is specified, the reach network links output (specified in the OUT_LINKS= option) cannot be generated.

DIGRAPH
calculates the directed reach counts when computing the reach networks and includes the directed counts in the resulting output data set that is specified in the OUT_COUNTS= option. This option is ignored unless you specify MAXREACH=1 in the REACH statement.

EACH_SOURCE
treats each node as a source and calculates a reach network from each one.

IGNORE_SELF
ignores the source nodes in the reach network node counts.

MAXREACH=*number*
specifies the maximum number of links to allow from each source node in a reach network. The default is 1.

LOGFREQTIME=*number*
displays iteration logs for the reach algorithm every *number* seconds. When PROC OPTGRAPH runs the reach algorithm, it displays the number of source networks that have completed. When you also specify the BY_CLUSTER option in the REACH statement, PROC OPTGRAPH displays the number of subgraphs that have completed. The value of *number* can be any integer greater than or equal to 1; the default is 5. 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 1.36 describes the valid values for this option.

Table 1.36 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 OPTGRAPH statement (or BASIC if that option is not specified).

OUT_COUNTS=*SAS-data-set*
specifies the output data set to contain the node counts in each reach network.

OUT_COUNTS1=*SAS-data-set*
specifies the output data set to contain the node counts in each reach network for the special case of calculating only counts that have limit 1 and 2. This data set holds the counts with MAXREACH=1. This option works only when the EACH_SOURCE and BY_CLUSTER options are specified.

OUT_COUNTS2=*SAS-data-set*
specifies the output data set to contain the node counts in each reach network for the special case of calculating only counts that have limit 1 and 2. This data set holds the counts with MAXREACH=2. This option works only when the EACH_SOURCE and BY_CLUSTER options are specified.

OUT_LINKS=*SAS-data-set*
specifies the output data set to contain the links in each reach network.

OUT_NODES=*SAS-data-set*
specifies the output data set to contain the nodes in each reach network.

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 143.

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 1.37 describes the valid values for this option.

Table 1.37 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 OPTGRAPH 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 | SHORTEST | LONGEST
specifies the type of output to produce in the output data set that is specified in the OUT_PATHS= option.

Table 1.38 Values for the PATHS= Option

Option Value	Description
ALL	Outputs shortest paths for all pairs of source-sinks. This is the default.
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.

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 OPTGRAPH 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 OPTGRAPH statement.

USEWEIGHT=YES | NO

specifies whether to use link weights (if they exist) in calculating shortest paths.

Table 1.39 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 *column* must be numeric.

SUMMARY Statement

SUMMARY < *options* > **;**

The SUMMARY statement invokes an algorithm that calculates various summary metrics on an input graph.

The summary metrics are described in the section "Summary" on page 155.

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

BICONCOMP

specifies whether to calculate information about biconnected components. The graph must be undirected.

BY_CLUSTER
specifies whether to decompose the calculations by cluster (or subgraph). If this option is specified, PROC OPTGRAPH looks for a definition of the clusters in the input data set specified in the DATA_NODES= option.

CONCOMP
specifies whether to calculate information about connected components.

DIAMETER_APPROX=WEIGHT | UNWEIGHT | BOTH
specifies whether to calculate information about the approximate diameter and what type of calculations to perform. Use this option when calculating the exact diameter (by calculating all shortest paths) is too expensive.

Table 1.40 Values for the DIAMETER_APPROX= Option

Option Value	Description
WEIGHT	Calculates approximate diameter based on the weighted graph.
UNWEIGHT	Calculates approximate diameter based on the unweighted graph.
BOTH	Calculates approximate diameter based on both weighted and unweighted graphs.

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

LOGFREQNODE=*number*
controls the frequency for displaying iteration logs for some of the summary metrics. For computationally intensive summary metrics such as shortest path, this option displays progress every *number* nodes. If you also specify the BY_CLUSTER option in this statement or a value greater than 1 for the NTHREADS= option in the PERFORMANCE statement, this option is ignored and the display frequency is determined by using the LOGFREQTIME= option instead. 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.

LOGFREQTIME=*number*
controls the frequency for displaying iteration logs for some of the summary metrics. For computationally intensive summary metrics such as shortest path, this option displays progress every *number* seconds. When you specify a value greater than 1 for the NTHREADS= option in the PERFORMANCE statement, PROC OPTGRAPH displays the number of nodes that have completed. When you specify the BY_CLUSTER option, PROC OPTGRAPH displays the number of subgraphs that have completed. The value of *number* can be any integer greater than or equal to 1; the default is 5. 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 1.41 describes the valid values for this option.

Table 1.41 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 1.41 (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 OPTGRAPH statement (or BASIC if that option is not specified).

OUT=*SAS-data-set*
specifies the output data set to contain the summary results.

SHORTPATH=WEIGHT | UNWEIGHT | BOTH
specifies whether to calculate information about shortest paths and what type of calculations to perform.

Table 1.42 Values for the SHORTPATH= Option

Option Value	**Description**
WEIGHT	Calculates shortest paths based on the weighted graph.
UNWEIGHT	Calculates shortest paths based on the unweighted graph.
BOTH	Calculates shortest paths based on both weighted and unweighted graphs.

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

SUBSIZESWITCH=*number*
specifies the size of the subgraphs (number of nodes) to run separately when you also specify the BY_CLUSTER option in this statement and a value greater than 1 for the NTHREADS= option in the PERFORMANCE statement. When PROC OPTGRAPH processes summary by subgraphs, it uses thread logic to simultaneously process *n* subgraphs, where *n* is the number of threads specified in the NTHREADS= option in the PERFORMANCE statement. Subgraphs that have more nodes than *number* are processed sequentially, enabling the threading to be done at the summary metric level. The default is 10,000.

TRANSITIVE_CLOSURE Statement

TRANSITIVE_CLOSURE < *options* > **;**

TRANSC < *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 161.

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 1.43 describes the valid values for this option.

Table 1.43 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 OPTGRAPH 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 164. 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 options below are the same as those described for PROC OPTMILP 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 procedure stops. The value of *number* can be any nonnegative number; the default value is 1E–6.

CUTOFF=*number*
cuts off any branch-and-bound nodes in a minimization problem with 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=*option*
specifies the level of cuts to be generated by PROC OPTGRAPH. Table 1.44 lists the valid values for this option.

Table 1.44 Values for CUTSTRATEGY= Option

number	*string*	**Description**
–1	AUTOMATIC	Disables most of the generic mixed-integer programming cuts and focuses on the generation of TSP-specific cuts

Table 1.44 (continued)

number	*string*	**Description**
0	NONE	Disables generation of cutting planes
1	MODERATE	Uses a moderate cut strategy
2	AGGRESSIVE	Uses an aggressive cut strategy

The default is AUTOMATIC.

EMPHASIS=*number* | *string*

specifies a search emphasis *option* or its corresponding value *number* as listed in Table 1.45.

Table 1.45 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

The default is BALANCE.

HEURISTICS=*number* | *string*

controls the level of initial and primal heuristics that are applied by PROC OPTGRAPH. This level determines how frequently primal heuristics are applied 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 1.46 lists the valid values for this option.

Table 1.46 Values for HEURISTICS= Option

number	*string*	**Description**
–1	AUTOMATIC	Applies the default level of heuristics
0	NONE	Disables all initial and primal heuristics
1	BASIC	Applies basic intial and primal heuristics at low frequency
2	MODERATE	Applies most intial and primal heuristics at moderate frequency
3	AGGRESSIVE	Applies all intitial primal heuristics at high frequency

The default is AUTOMATIC.

INTTOL=*number*

specifies the amount by which an integer variable value can differ from an integer and still be considered integer feasible. The value of *number* can be any number between 0.0 and 1.0; the default value is 1E–5. PROC OPTGRAPH attempts to find an optimal solution with integer infeasibility less than *number*. If you assign a value that is less than 1E–10 to *number* and the best solution found by PROC OPTGRAPH has integer infeasibility between *number* and 1E–10, then PROC OPTGRAPH ends with a solution status of OPTIMAL_COND (see the section "TSP" on page 172).

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 1.47 describes the valid values for this option.

Table 1.47 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 OPTGRAPH uses its initial heuristics to find a feasible, but not necessarily optimal, tour as quickly as possible. Table 1.48 describes the valid values for this option.

Table 1.48 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

NODESEL=*number* | *string*
specifies the branch-and-bound node selection strategy *option* or its corresponding value *number*, as listed in Table 1.49.

Table 1.49 Values for NODESEL= Option

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

The default is AUTOMATIC. For more information about node selection, see Chapter 11, "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* or its corresponding value *number*, as listed in Table 1.50:

Table 1.50 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

The default value is 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

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

When this value becomes less than the specified gap size *number*, the procedure stops. The value of *number* can be any number between 0 and 1; the default value is 1E–4.

STRONGITER=*number*

specifies the number of simplex iterations to be performed for each variable in the candidate list when using the strong branching variable selection strategy. The value of *number* can be any positive number; the default value is automatically calculated by PROC OPTGRAPH.

STRONGLEN=*number*

specifies the number of candidates to be used when performing the strong branching variable selection strategy. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$. The default value is 10.

TARGET=*number*

specifies a stopping criterion for minimization (maximization) problems. If the best integer objective is better than or equal to *number*, the procedure stops. The value of *number* can be any number; the default is the negative (positive) 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 1.51 lists the valid values for this option.

Table 1.51 Values for VARSEL= Option

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

The default is STRONG. For more information about variable selection, see Chapter 11, "The OPTMILP Procedure" (*SAS/OR User's Guide: Mathematical Programming*).

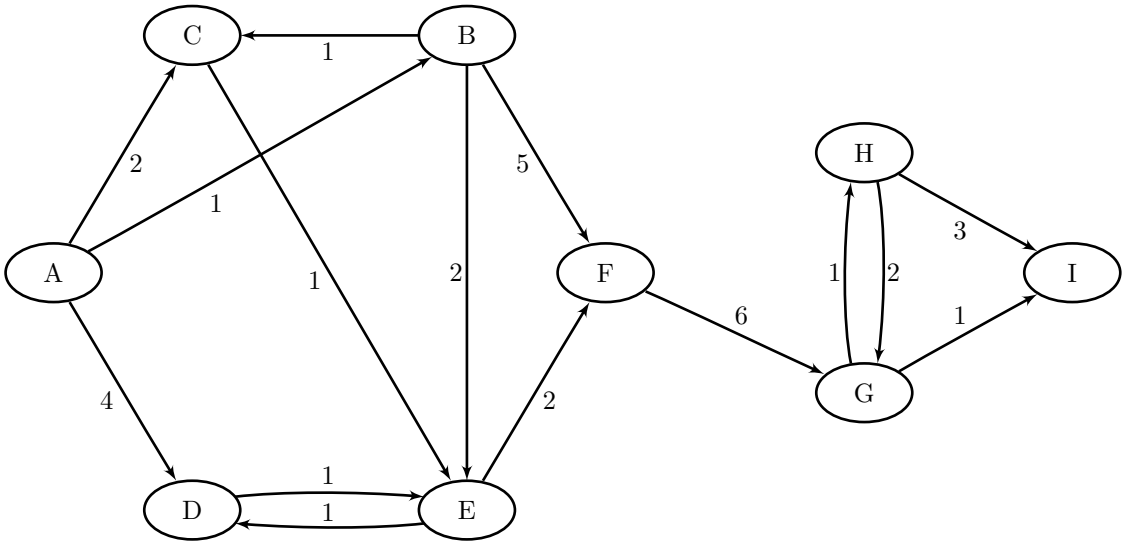
Details: OPTGRAPH Procedure

Graph Input Data

This section describes how to input a graph for analysis by PROC OPTGRAPH. Let $G = (N, A)$ define a graph with a set N of nodes and a set A of links. There are two main methods for defining the set of links A as a SAS data set. The first is to use a list of links as described in the section "Link Input Data" on page 49. The second is to use an adjacency matrix as described in the section "Adjacency Matrix Input Data" on page 53.

To illustrate the different methods for input of a graph, consider the directed graph shown in Figure 1.8.

Figure 1.8 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 OPTGRAPH 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)
- 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 32.

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 OPTGRAPH by using the following equivalent statements:

```
proc optgraph
   data_links = LinkSetInA;
run;

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

The directed graph G shown in Figure 1.8 can be represented by the links data set LinkSetIn as follows:

```
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 simply echoes back the input graph.

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   out_nodes       = NodeSetOut
   out_links       = LinkSetOut;
run;
```

The data set NodeSetOut, shown in Figure 1.9, now contains the nodes that were read from the input link data set. The variable node shows the label associated with each node.

Figure 1.9 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 1.10, contains the links that were read from the input link data set. The variables from and to show the associated node labels.

Figure 1.10 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, $D \leftrightarrow E$) are treated as the same link and duplicates are removed. PROC OPTGRAPH takes the first occurrence of the link and ignores the others. The default for the GRAPH_DIRECTION= option is UNDIRECTED, so you can just remove this option to declare the graph as undirected.

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

The progress of the procedure is shown in Figure 1.11. The log now shows the links (and their observation identifiers) that were declared as duplicates and removed.

Figure 1.11 PROC OPTGRAPH Log: Link Data Set of a Simple Undirected Graph

```
NOTE: ------------------------------------------------------------------------
NOTE: Running OPTGRAPH version 12.3.
NOTE: ------------------------------------------------------------------------
NOTE: The OPTGRAPH procedure is executing in single-machine mode.
NOTE: ------------------------------------------------------------------------
NOTE: Data input used 0.01 (cpu: 0.02) 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 1.9. However, the new links data set LinkSetOut shown in Figure 1.12 contains two fewer links than before, because duplicates are removed.

Figure 1.12 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 OPTGRAPH statement. However, when you specify this option, duplicate links are not removed by the procedure. Instead, you should use appropriate DATA steps to clean your data before calling PROC OPTGRAPH.

Adjacency Matrix Input Data

An alternate way to define the links of an input graph is to use an adjacency matrix and the DATA_ADJ_MATRIX= option in the PROC OPTGRAPH statement. An *adjacency matrix* is a square matrix with one row and column for each node in the graph and a nonzero value to represent the existence (or weight) of a link in the graph. The row index defines the *from* node, and the column index defines the *to* node. A matrix value that is 0 or missing (.) represents a link that does not exist in the graph.

You can specify any values that you want for the data set variable names (the columns) by using the DATA_ADJ_MATRIX_VAR statement, as described in the section "DATA_ADJ_MATRIX_VAR Statement" on page 32. If no names are given, then PROC OPTGRAPH assumes that all numeric variables in the data set are to be used in defining nodes and links.

The directed graph G shown in Figure 1.8 can be represented structurally by using the adjacency matrix data set AdjMatSetIn as follows:

```
data AdjMatSetIn;
   input var1-var9;
   datalines;
0 1 1 1 0 0 0 0 0
0 0 1 0 1 1 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0 0
0 0 0 1 0 1 0 0 0
0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 1 1
0 0 0 0 0 0 1 0 1
0 0 0 0 0 0 0 0 0
;
```

Equivalently, the following data set provides the same information by using missing values (.) instead of 0s:

```
data AdjMatSetIn;
   input var1-var9;
   datalines;
. 1 1 1 . . . . .
. . 1 . 1 1 . . .
. . . . 1 . . . .
. . . . 1 . . . .
. . . 1 . 1 . . .
. . . . . . 1 . .
. . . . . . . 1 1
. . . . . . 1 . 1
. . . . . . . . .
;
```

To represent the weights, you can simply use the weights from Figure 1.8 in the input matrix as follows:

```
data AdjMatWtSetIn;
   input var1-var9;
   datalines;
. 1 2 4 . . . . .
. . 1 . 2 5 . . .
. . . . 1 . . . .
. . . . 1 . . . .
```

```
. . . 1 . 2 . . .
. . . . . . 6 . .
. . . . . . . 1 1
. . . . . . 2 . 3
. . . . . . . . .
;
```

This same graph can be represented by the links data set LinkSetInNum as follows:

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

So the following two procedure calls are equivalent:

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetInNum;
run;

proc optgraph
   graph_direction = directed
   data_adj_matrix = AdjMatWtSetIn;
run;
```

The first set of statements uses the DATA_LINKS= option, which represents the graph in sparse format, as described in the section "Link Input Data" on page 49. The second set of statements uses the DATA_ADJ_MATRIX= option, which represents the graph as an adjacency matrix (a dense format). The dense format is not appropriate for large graphs because the memory requirements grow quadratically with the number of nodes.

Node Input Data

The DATA_NODES= option in the PROC OPTGRAPH statement defines the data set that contains the list of nodes in the graph. This data set is used to define clusters (subgraphs) or 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)
- cluster: the node cluster identifier (this variable must be numeric)
- weight: the node weight (this variable must be numeric)
- weight2: the auxiliary node weight (this variable must be numeric)

The variable cluster is used to define clusters (subgraphs) for decomposing the input graph into subgraphs for processing. This is useful for the algorithms specified in the CENTRALITY, REACH, and SUMMARY statements. The use of the variable cluster is explained in more detail in the section "Processing by Cluster" on page 80.

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 33.

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 OPTGRAPH takes the first occurrence of the node and ignores the others. A warning is printed to the log.

Node Subset Input Data

For various algorithms, you might want to process only a subset of the nodes in the input graph. You can accomplish this by using the DATA_NODES_SUB= option in the PROC OPTGRAPH statement. You can use the node subset data set in conjunction with the SHORTPATH, REACH, or CENTRALITY statement. (See the sections "Shortest Path" on page 143, "Reach (Ego) Network" on page 130, and "Centrality" on page 63, respectively.) 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)
- reach: for the reach algorithm, the index of the source subgraph for processing (this variable must be numeric)
- centr: whether to process this node in centrality algorithms (this variable must be numeric)

Table 1.52 shows how PROC OPTGRAPH processes nodes for each algorithm type. The missing indicator (.) can also be used in place of 0 to designate that a node is not to be processed.

Table 1.52 Determining How to Process a Node

Algorithm Type	Variable Designations	Example Shown In:
Shortest path	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 section "Shortest Path" on page 143
Centrality	A value of 0 for the centr variable designates that the node is not to be processed. A value of 1 designates that the node is to be processed.	The section "Processing a Subset of Nodes" on page 77
Reach	A value of 0 for the reach variable designates that the node is not to be processed. A value greater than 0 defines a marker for the source subgraph to which this node belongs. All nodes with the same marker are processed as source nodes together.	The section "Reach (Ego) Network" on page 130

A representative example of a node subset data set that might be used with the graph in Figure 1.8 is as follows:

```
data NodeSubSetIn;
   input node $ reach centr source sink;
   datalines;
A 1 1 1 .
F 2 1 . 1
E 2 . 1 .
;
```

The data set NodeSubSetIn indicates that you want to process the following:

- the reach network from the subgraph defined by node A
- the reach network from the subgraph defined by nodes F and E
- the centrality metrics on nodes A and F

The data set NodeSubSetIn indicates that you want to process the shortest paths from nodes A and E and the shortest paths to node F.

Matrix Input Data

This section describes the matrix input format that you can use with some of the algorithms in PROC OPTGRAPH. The DATA_MATRIX= option in the PROC OPTGRAPH 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 32. If you do not specify any names, then PROC OPTGRAPH assumes that all numeric variables in the data set are to be used in defining the matrix.

The following statements find the principal eigenvector of the square symmetric matrix that is defined in the data set Matrix:

```
data Matrix;
   input col1-col5;
   datalines;
1 0 2 6 1
0 2 3 0 1
2 3 1 0 2
6 0 0 0 0
1 1 2 0 0
;

proc optgraph
   data_matrix    = Matrix;
   eigenvector
      eigenvalues = LA
      nEigen      = 1
      out         = EigenVector;
run;
```

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

Parallel Processing

A number of the algorithms in PROC OPTGRAPH can take advantage of multicore chip technology by performing some of the computations in parallel. To enable PROC OPTGRAPH to process in parallel, you can specify the number of threads to use with the NTHREADS= option in the PERFORMANCE statement. There are two ways in which PROC OPTGRAPH can decompose the computational work in order to take advantage of parallel processing: by node and by subgraph.

To process the nodes of the graph individually, set the NTHREADS= option to some value greater than 1. You can do this for the centrality metrics closeness (see the section "Closeness Centrality" on page 68) and betweenness (see the section "Betweenness Centrality" on page 70). An example of this is shown in "Example 1.4: Betweenness and Closeness Centrality for Project Groups in a Research Department" on page 191.

To process the subgraphs of the original graph individually, set the NTHREADS= option to some value greater than 1, and designate the clusters in each node by using the cluster variable in the nodes data set, as described in the section "Node Input Data" on page 54. You can do this for centrality metrics, reach networks, and summary statistics. (See the sections "Centrality" on page 63, "Reach (Ego) Network" on page 130, and "Summary" on page 155, respectively.) A common use for this feature is to first decompose the original graph into communities or components. (See the sections "Community" on page 92 and "Connected Components" on page 100, respectively.) Then, from these results, define the clusters in the node data set and run the analysis of each subgraph individually and in parallel. PROC OPTGRAPH takes care of all of the accounting with the associated decomposition and returns results in terms of the original graph. An example of this process for centrality is shown in the section "Processing by Cluster" on page 80.

You can improve the performance of the OPTGRAPH procedure by running it in distributed computing mode. For more information about the high-performance features of the OPTGRAPH procedure, see *SAS OPTGRAPH Procedure: High-Performance Features*.

NOTE: Distributed computing mode requires SAS High-Performance Analytics software.

Size Limitations

PROC OPTGRAPH can handle any graph whose number of nodes is less than or equal to 2,147,483,647 (the maximum representable 32-bit integer). This maximum also applies to 64-bit systems. For graphs of two billion nodes, memory limitations also become a limiting factor. For example, see the discussion of memory requirements for the community detection algorithm in the section "Memory Requirement" on page 94.

If the data from your problem require a graph with more than two billion nodes, 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 OPTGRAPH. 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 OPTGRAPH, 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 for the biconnected components algorithm are written to the output links data set that is specified in the OUT_LINKS= option in the PROC OPTGRAPH 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 for the articulation points are written to the output nodes data set that is specified in the OUT_NODES= option in the PROC OPTGRAPH 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).

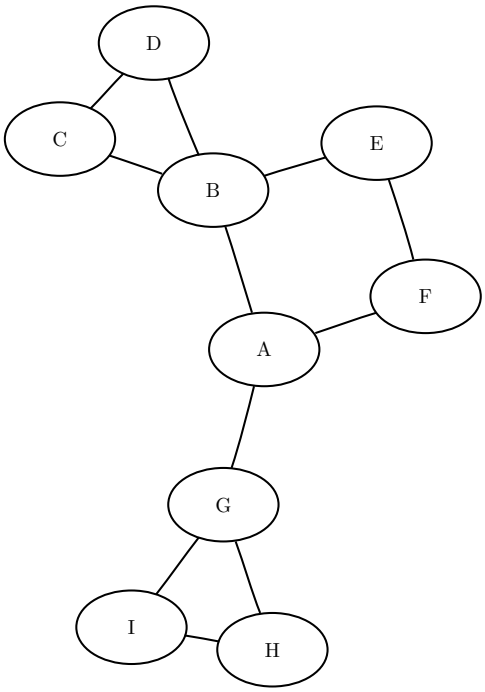
The biconnected components algorithm reports status information in a macro variable called _OPTGRAPH_BICONCOMP_. See the section "Macro Variable _OPTGRAPH_BICONCOMP_" on page 173 for more information about this macro variable.

The algorithm used by PROC OPTGRAPH 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 shown in Figure 1.13.

Figure 1.13 A Simple Undirected Graph G



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

```
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 optgraph
   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 1.14.

Figure 1.14 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 1.15.

Figure 1.15 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 1.16 and Figure 1.17.

Figure 1.16 Biconnected Components C^1 and C^2

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

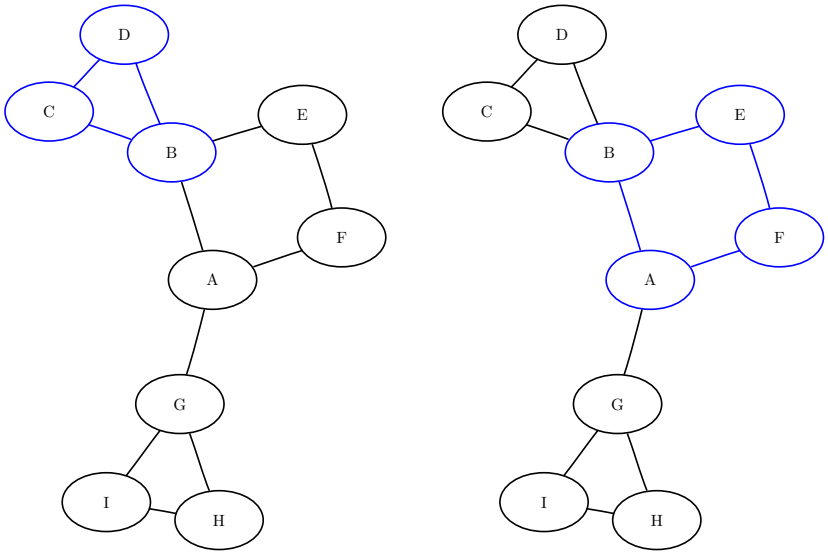
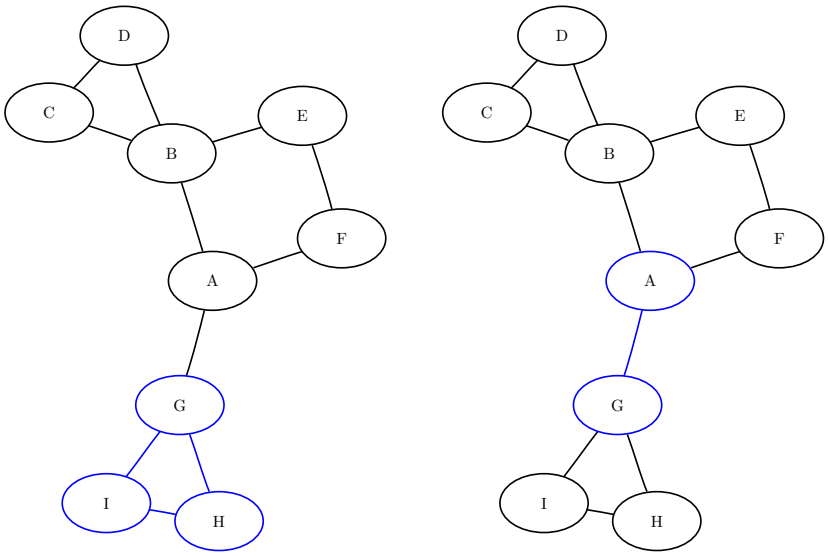


Figure 1.17 Biconnected Components C^3 and C^4

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



For a more detailed example, see "Example 1.1: Articulation Points in a Terrorist Network" on page 180.

Centrality

In general terms, the *centrality* of a node or link in a graph gives some indication of its relative importance within a graph. In the field of network analysis, many different types of centrality metrics are used to better understand levels of prominence. For a good review of centrality metrics, see Newman 2010.

You can use the CENTRALITY statement in PROC OPTGRAPH to calculate several of these metrics. The options for this statement are described in the section "CENTRALITY Statement" on page 20.

The CENTRALITY statement reports status information in a macro variable called _OPTGRAPH_CENTR_. See the section "Macro Variable _OPTGRAPH_CENTR_" on page 173 for more information about this macro variable.

The following sections describe each of the possible centrality metrics that can be calculated in PROC OPTGRAPH.

Degree Centrality

The *degree* of a node v in an undirected graph is the number of links that are incident to node v. The *out-degree* of a node in a directed graph is the number of out-links incident to that node; the *in-degree* is the number of in-links incident. The term *degree* and *out-degree* are interchangeable for an undirected graph. *Degree centrality* is simply the (in- or out-) degree of a node and can be interpreted as some form of relative importance to a network. For example, in a network where nodes are people and you are tracking the flow of a virus, the degree centrality gives some idea of the magnitude of the risk of spreading the virus. People with a higher out-degree can lead to a quicker and more widespread transmission. In a friendship network, in-degree often indicates popularity.

Degree centrality is calculated according to the value specified for the DEGREE= option in the CENTRALITY statement. The results are provided in the node output data set that is specified in the OUT_NODES= option in the PROC OPTGRAPH statement.

The algorithm used by PROC OPTGRAPH to compute degree centrality is a simple lookup, runs in time $O(|N|)$, and therefore should scale to very large graphs.

As a simple example, consider again the directed graph in Figure 1.8 with data set LinkSetIn defined in the section "Link Input Data" on page 49. The following statements calculate the degree centrality for both in- and out-degree:

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   out_nodes       = NodeSetOut;
   centrality
      degree       = both;
run;
```

The node data set NodeSetOut now contains the degree centrality of the input graph. For a directed graph, the data set provides the in-degree (variable centr_degree_in), the out-degree (variable centr_degree_out), and the degree that is the sum of in- and out-degrees (variable centr_degree). This data set is shown in Figure 1.18.

Figure 1.18 Degree Centrality of a Simple Directed Graph

node	centr_ degree_ in	centr_ degree_ out	centr_ degree
A	0	3	3
B	1	3	4
C	2	1	3
D	2	1	3
E	3	2	5
F	2	1	3
G	2	2	4
H	1	2	3
I	2	0	2

Influence Centrality

Influence centrality is a generalization of degree centrality that considers the link and node weights of adjacent nodes (C_1) in addition to the link weights of nodes that are adjacent to adjacent nodes (C_2). The metric C_1 is referred to as *first-order influence centrality*, and the metric C_2 is referred to as *second-order influence centrality*.

Let w_{uv} define the link weight for link (u, v), and let w_u define the node weight for node u. Let δ_u represent the list of nodes connected to node u (that is, its neighbors); this list is called the adjacency list. For directed graphs, the neighbors are the out-links. The general formula for influence centrality is

$$C_1(u) = \frac{\sum_{v \in \delta_u} w_{uv}}{\sum_{v \in N} w_v}$$

$$C_2(u) = \sum_{v \in \delta_u} C_1(v)$$

As the name suggests, this metric gives some indication of potential influence, performance, or ability to transfer knowledge.

Influence centrality is calculated according to the value of the INFLUENCE= option in the CENTRALITY statement. The results are provided in the node output data set that is specified in the OUT_NODES= option in the PROC OPTGRAPH statement.

The algorithm used by PROC OPTGRAPH to compute influence centrality is a simple traversal, runs in time $O(|A|)$, and therefore should scale to very large graphs.

Consider again the directed graph in Figure 1.8. Ignore the weights and just calculate the C_1 and C_2 metrics based on connections (that is, consider all link and node weights as 1). The following statements calculate the unweighted influence centrality:

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   out_nodes       = NodeSetOut;
   centrality
      influence    = unweight;
run;
```

The node data set NodeSetOut now contains the unweighted influence centrality of the input graph, including the C_1 variable centr_influence1_unwt and the C_2 variable centr_influence2_unwt. This data set is shown in Figure 1.19.

Figure 1.19 Influence Centrality of a Simple Directed Graph

node	centr_ influence1_ unwt	centr_ influence2_ unwt
A	0.33333	0.55556
B	0.33333	0.44444
C	0.11111	0.22222
D	0.11111	0.22222
E	0.22222	0.22222
F	0.11111	0.22222
G	0.22222	0.22222
H	0.22222	0.22222
I	0.00000	0.00000

For a more detailed example, see "Example 1.2: Influence Centrality for Project Groups in a Research Department" on page 183.

Clustering Coefficient

The *clustering coefficient* for a node is the number of links between the nodes within its neighborhood divided by the number of links that could possibly exist between them.

Let δ_u represent the list of nodes that are connected to node u. The formula for the clustering coefficient is:

$$C(i) = \frac{|\{(u,v) \in A : u,v \in \delta_i\}|}{|\delta_i|(|\delta_i| - 1)}$$

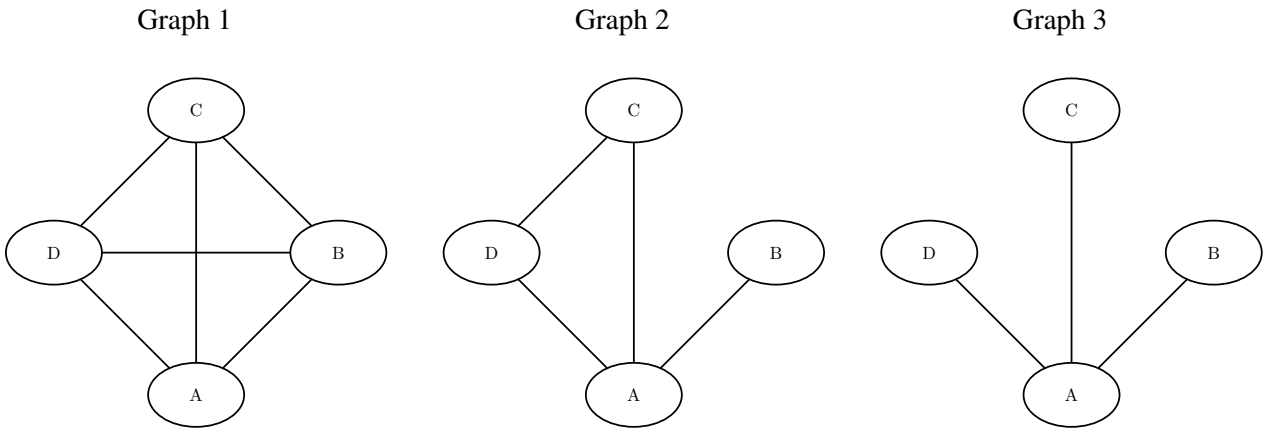
For a particular node i, the clustering coefficient determines how close to being a clique (complete subgraph) the subgraph induced by itself and its neighbor set δ_i are. In social networks, a high clustering coefficient can help predict relationships that might not be known, confirmed, or realized yet. The fact that person A knows person B and person B knows person C does not guarantee that person A knows person C, but it is much more likely that person A knows person C than that person A knows some random person.

The clustering coefficient is calculated when the CLUSTERING_COEF option is specified in the CENTRALITY statement. The results are provided in the node output data set that is specified in the OUT_NODES= option in the PROC OPTGRAPH statement.

The algorithm used by PROC OPTGRAPH to compute the clustering coefficient is a relatively simple traversal, runs in time $O(|A|)$, and therefore should scale to very large graphs.

Consider the three undirected graphs on four nodes shown in Figure 1.20.

Figure 1.20 Three Undirected Graphs



Define the three link data sets as follows:

```
data LinkSetInCC1;
   input from $ to $ @@;
   datalines;
A B  A C  A D
B C  B D  C D
;

data LinkSetInCC2;
   input from $ to $ @@;
   datalines;
A B  A C  A D
C D
;

data LinkSetInCC3;
   input from $ to $ @@;
   datalines;
A B  A C  A D
;
```

The following statements use three calls to PROC OPTGRAPH to calculate the clustering coefficients for each graph:

```
proc optgraph
   data_links = LinkSetInCC1
   out_nodes  = NodeSetOut1;
   centrality
      clustering_coef;
run;
```

```
proc optgraph
   data_links = LinkSetInCC2
   out_nodes  = NodeSetOut2;
   centrality
      clustering_coef;
run;

proc optgraph
   data_links = LinkSetInCC3
   out_nodes  = NodeSetOut3;
   centrality
      clustering_coef;
run;
```

The node data sets provide the clustering coefficients for each graph (variable centr_cluster) as shown in Figure 1.21 through Figure 1.23.

Figure 1.21 Clustering Coefficient of a Simple Undirected Graph 1

node	centr_ cluster
A	1
B	1
C	1
D	1

Figure 1.22 Clustering Coefficient of a Simple Undirected Graph 2

node	centr_ cluster
A	0.33333
B	0.00000
C	1.00000
D	1.00000

Figure 1.23 Clustering Coefficient of a Simple Undirected Graph 3

node	centr_ cluster
A	0
B	0
C	0
D	0

Closeness Centrality

Closeness centrality is the reciprocal of the average of the shortest paths (geodesic distances) to all other nodes. Closeness can be thought of as a measure of how long it would take information to spread from a given node to other nodes in the network.

The general formula for closeness centrality is

$$C_c(u) = \frac{|C| - 1}{\sum_{v \in N \setminus u} d_{uv}}$$

where C is the component that contains u and d_{uv} is the shortest path from node u to node v.

Directed graphs have three versions of the metric:

$$\begin{aligned} C_c^{\text{out}}(u) &= \frac{|C| - 1}{\sum_{v \in N \setminus u} d_{uv}} \\ C_c^{\text{in}}(u) &= \frac{|C| - 1}{\sum_{v \in N \setminus u} d_{vu}} \\ C_c(u) &= \frac{|C| - 1}{\left(\sum_{v \in N \setminus u} d_{uv} + d_{vu}\right)/2} \end{aligned}$$

Closeness centrality is calculated according to the value of the CLOSE= option in the CENTRALITY statement. The results are provided in the node output data set that is specified in the OUT_NODES= option in the PROC OPTGRAPH statement. If CLOSE=WEIGHT (or BOTH), then the calculation of shortest paths is done using the weighted graph. Because the metric uses shortest paths to determine closeness, the weight and the closeness metric are inversely related. In general the lower the weight, the higher the contribution to the closeness metric.

The CLOSE_NOPATH= option specifies the handling of the case where a path between two nodes does not exist. The textbook formula for centrality says to contribute nothing to the total distance and normalize based on the size of the component that contains the source node. This approach corresponds to CLOSE_NOPATH=ZERO. For this option, the metric is also scaled by $(|C| - 1)/(|N| - 1)$. That is, it is scaled by the number of nodes in the component (minus 1) divided by the number of nodes in the entire original graph (minus 1). Closeness centrality is then calculated as

$$C_c(u) = \left(\frac{|C| - 1}{|N| - 1}\right)\left(\frac{|C| - 1}{\sum_{v \in C \setminus u} d_{uv}}\right)$$

A more common approach is to add, for each node that is not reachable, the longest possible geodesic distance (the *diameter*). Then, consider the entire graph as the normalizing factor. This approach corresponds to CLOSE_NOPATH=DIAMETER. Let

$$d_{uv}^{\text{diam}} = \begin{cases} d_{uv} & \text{if } d_{uv} < \infty \\ \max_{(u,v):d_{uv}<\infty}(d_{uv}) & \text{otherwise} \end{cases}$$

Then

$$C_c(u) = \frac{|N| - 1}{\sum_{v \in N \setminus u} d_{uv}^{\text{diam}}}$$

Another alternative is to use the total number of nodes when accounting for unreachable nodes. This approach corresponds to CLOSE_NOPATH=NNODES and is the default setting in PROC OPTGRAPH. Let

$$d_{uv}^{\mathrm{N}} = \begin{cases} d_{uv} & \text{if } d_{uv} < \infty \\ |N| & \text{otherwise} \end{cases}$$

Then

$$C_c(u) = \frac{|N| - 1}{\sum_{v \in N \setminus u} d_{uv}^{\mathrm{N}}}$$

The algorithm used by PROC OPTGRAPH to compute closeness centrality relies on calculating shortest paths for all source-sink pairs and runs in time $O(|N| \times (|N| \log |N| + |A|))$. Therefore, it is not expected to scale to very large graphs. Because the shortest path computations can be calculated independently (for each source node), the algorithm can be sped up by using the NTHREADS= option in the PERFORMANCE statement.

Consider again the directed graph in Figure 1.8 with data set LinkSetIn defined in the section "Link Input Data" on page 49. The following statements calculate the closeness centrality for both the weighted and unweighted graphs:

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   out_nodes       = NodeSetOut;
   centrality
      close        = both;
run;
```

The node data set NodeSetOut now contains the weighted and unweighted directed closeness centrality of the input graph. The data set provides the unweighted closeness (the centr_close_unwt variable), in-closeness (the centr_close_in_unwt variable), and out-closeness (the centr_close_out_unwt variable). It also provides the weighted variants centr_close_wt, centr_close_in_wt, and centr_close_out_wt. This data set is shown in Figure 1.24.

Figure 1.24 Closeness Centrality of a Simple Directed Graph

node	centr_ close_wt	centr_ close_ in_wt	centr_ close_ out_wt	centr_ close_ unwt	centr_ close_in_ unwt	centr_ close_out_ unwt
A	0.13115	0.11111	0.16000	0.17778	0.11111	0.44444
B	0.13913	0.12500	0.15686	0.18605	0.12500	0.36364
C	0.14545	0.14035	0.15094	0.17778	0.14286	0.23529
D	0.15094	0.17391	0.13333	0.19277	0.19048	0.19512
E	0.16162	0.18605	0.14286	0.20513	0.19512	0.21622
F	0.14679	0.18182	0.12308	0.18824	0.22857	0.16000
G	0.13333	0.12500	0.14286	0.20000	0.33333	0.14286
H	0.12500	0.11594	0.13559	0.18605	0.26667	0.14286
I	0.11852	0.12698	0.11111	0.17021	0.36364	0.11111

Betweenness Centrality

Betweenness centrality counts the number of times a particular node (or link) occurs on shortest paths between other nodes. Betweenness can be thought of as a measure of the control a node (or link) has over the communication flow among the rest of the network. In this sense, the nodes (or links) with high betweenness are the *gatekeepers* of information, because of their relative location in the network.

The formula for node betweenness centrality is

$$C_b(u) = \sum_{\substack{s \neq u \neq t \in N \\ s \neq t}} \frac{\sigma_{st}(u)}{\sigma_{st}}$$

where σ_{st} is the number of shortest paths from s to t and $\sigma_{st}(u)$ is the number of shortest paths from s to t that pass through node u.

The formula for link betweenness centrality is

$$C_b(u,v) = \sum_{\substack{s,t \in N \\ s \neq t}} \frac{\sigma_{st}(u,v)}{\sigma_{st}}$$

where $\sigma_{st}(u,v)$ is the number of shortest paths from s to t that pass through link (u,v).

By default, this metric is normalized by dividing through by two times the number of pairs of nodes, not including u, which is $(|N|-1)(|N|-2)$. This normalization can be disabled by using the BETWEEN_NORM= option.

For directed graphs, because the paths are directed, only the *out-betweenness* is computed. To get the *in-betweenness*, you must reverse all the directions of the graph and run the procedure again. This can be accomplished by simply using the DATA_LINKS_VAR statement to reverse the interpretation of *from* and *to*.

Betweenness centrality is calculated according to the value of the BETWEEN= option in the CENTRALITY statement. The node betweenness results are provided in the node output data set that is specified in the OUT_NODES= option in the PROC OPTGRAPH statement. The link betweenness results are provided in the link output data set that is specified in the OUT_LINKS= option in the PROC OPTGRAPH statement. Like closeness, if BETWEEN=WEIGHT (or BOTH), then the calculation of shortest paths is done using the weighted graph. Because the metric uses shortest paths to determine betweenness, the weight and the betweenness metric are inversely related. In general the lower the weight, the higher the contribution to the betweenness metric.

The algorithm used by PROC OPTGRAPH to compute betweenness centrality relies on calculating shortest paths for all source-sink pairs and runs in time $O(|N| \times (|N| \log |N| + |A|))$. Therefore, it is not expected to scale to very large graphs. Similar to closeness centrality, because shortest path computations can be calculated independently (for each source node), the algorithm can be sped up by using the NTHREADS= option in the PERFORMANCE statement. When closeness and betweenness centrality are run together, PROC OPTGRAPH calculates both metrics in one run.

Consider again the directed graph in Figure 1.8 with data set LinkSetIn defined in the section "Link Input Data" on page 49. The following statements calculate the betweenness centrality for both the weighted and unweighted graphs:

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   out_links       = LinkSetOut
   out_nodes       = NodeSetOut;
   centrality
      between      = both;
run;
```

The node data set NodeSetOut now contains the weighted (variable centr_between_wt) and unweighted (variable centr_between_unwt) node betweenness centrality of the input graph. This data set is shown in Figure 1.25.

Figure 1.25 Node Betweenness Centrality of a Simple Directed Graph

node	centr_ between_ wt	centr_ between_ unwt
A	0.00000	0.00000
B	0.07738	0.07738
C	0.12202	0.00595
D	0.00000	0.00595
E	0.33482	0.17857
F	0.26786	0.26786
G	0.22321	0.21429
H	0.00000	0.00000
I	0.00000	0.00000

In addition, the link data set LinkSetOut contains the weighted (variable centr_between_wt) and unweighted (variable centr_between_unwt) link betweenness centrality of the input graph. This data set is shown in Figure 1.26.

Figure 1.26 Link Betweenness Centrality of a Simple Directed Graph

from	to	weight	centr_ between_ wt	centr_ between_ unwt
A	B	1	0.09524	0.09524
A	C	2	0.04315	0.02381
A	D	4	0.00446	0.02381
B	C	1	0.11458	0.01786
B	E	2	0.08780	0.04167
B	F	5	0.00000	0.14286
C	E	1	0.22917	0.11310
D	E	1	0.08929	0.09524
E	D	1	0.06696	0.05357
E	F	2	0.35714	0.21429
F	G	6	0.32143	0.32143
G	H	1	0.12500	0.12500
G	I	1	0.13393	0.12500
H	G	2	0.02679	0.01786
H	I	3	0.00893	0.01786

For more detailed examples, see "Example 1.3: Betweenness and Closeness Centrality for Computer Network Topology" on page 187 and "Example 1.4: Betweenness and Closeness Centrality for Project Groups in a Research Department" on page 191.

Eigenvector Centrality

Eigenvector centrality is an extension to degree centrality, in which *centrality points* are awarded for each neighbor. However, not all neighbors are equally important. Intuitively, a connection to an important node should contribute more to the centrality score than a connection to a less important node. This is the basic idea behind eigenvector centrality. Eigenvector centrality of a node is defined to be proportional to the sum of the scores of all nodes that are connected to it. Mathematically, it is

$$x_i = \frac{1}{\lambda} \sum_{j \in \delta_i} x_j = \frac{1}{\lambda} \sum_{j \in N} A_{ij} x_j$$

where x_i is the eigenvector centrality of node i, λ is a constant, δ_i is the set of nodes that connects to node i, and A_{ij} is the weight of the link from node i to node j.

Eigenvector centrality can be written as an eigenvector equation in matrix form as

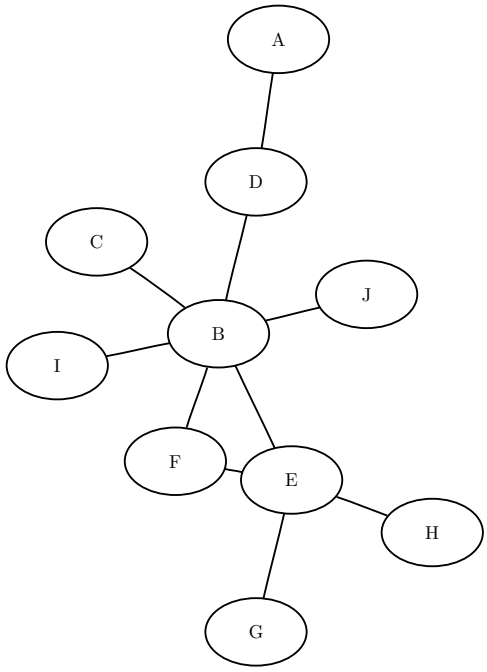
$$Ax = \lambda x$$

As can be seen from the preceding equation, x is the eigenvector and λ is the eigenvalue. Because x should be positive, only the principal eigenvector that corresponds to the largest eigenvalue is of interest.

Eigenvector centrality is calculated according to the value specified in the EIGEN= option in the CENTRALITY statement. The results are provided in the node output data set that is specified in the OUT_NODES= option in the PROC OPTGRAPH statement.

The following example illustrates the use of eigenvector centrality on the undirected graph G shown in Figure 1.27.

Figure 1.27 Eigenvector Centrality Example of a Simple Undirected Graph



The graph can be represented using the links data set LinkSetIn as follows:

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

The following statements compute the eigenvector centrality:

```
proc optgraph
   data_links        = LinkSetIn
   out_nodes         = NodeSetOut;
   centrality
      eigen          = unweight;
run;
```

The data set NodeSetOut now contains the eigenvector centrality of each node. It is shown in Figure 1.28.

Figure 1.28 Eigenvector Centrality Output

node	centr_ eigen_ unwt
B	1.00000
E	0.75919
F	0.61981
D	0.40226
C	0.35233
I	0.35233
J	0.35233
G	0.26749
H	0.26749
A	0.14173

Even though nodes F and D both have the same degree of 2, node F has a higher eigenvector centrality than node D. This is because node F links to two important nodes (B and E), whereas node D links to one important node (B) and one unimportant node (A).

For a more detailed example, see "Example 1.5: Eigenvector Centrality for Word Sense Disambiguation" on page 195.

One drawback of eigenvector centrality is that it does not work well for directed graphs. The reason is that many nodes in directed graphs have zero eigenvector centrality, except for the nodes in a strongly connected component of two or more nodes or the out-component of such a component. Therefore, nodes with zero scores cannot be distinguished. For this reason, PROC OPTGRAPH supports eigenvector centrality only on undirected graphs.

For a directed graph, you can use hub and authority scores, as described in the following sections.

Hub and Authority Scoring

Hub and authority centrality was originally developed by Kleinberg (1998) to rank the importance of web pages. Certain web pages are important in the sense that they point to many important pages (these are called *hubs*). On the other hand, some web pages are important because they are linked by many important pages (called *authorities*). In other words, a good hub node is one that points to many good authorities, and a good authority node is one that is pointed to by many good hub nodes. This idea can be applied to many other types of graphs besides web pages. For example, it can be applied to a citation network for journal articles. A review article that cites many good authority papers has a high hub score, whereas a paper that is referenced by many other papers has a high authority score. The section "Authority in U.S. Supreme Court Precedence" on page 7 shows a similar example.

The authority centrality of a node is proportional to the sum of the hub centrality of nodes that point to it. Similarly, the hub centrality of a node is proportional to the sum of the authorities of nodes that it points to.

That is,

$$\begin{aligned} x_i &= \alpha \sum_{j \in N} A_{ij} y_j \\ y_i &= \beta \sum_{j \in N} A_{ji} x_j \end{aligned}$$

where x_i is the authority centrality of node i, y_i is the hub centrality of node i, A_{ij} is the weight of the link from node i to node j, and α and β are constants.

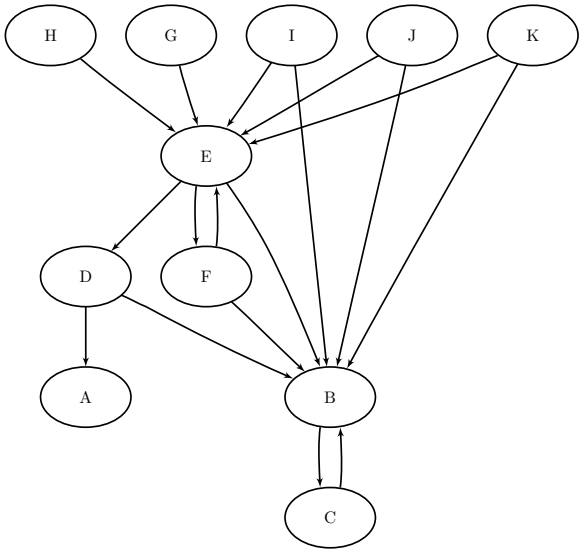
The definition can be written in matrix form as follows:

$$\begin{aligned} AA^T x &= \lambda x \\ A^T A y &= \lambda y \end{aligned}$$

Thus, the authority and hub centralities are the principal eigenvectors of $A^T A$ and AA^T, respectively. To solve this eigenvector problem, PROC OPTGRAPH provides two algorithms: the Jacobi-Davidson algorithm and the power method. You use the EIGEN_ALGORITHM= option in the CENTRALITY statement to specify which algorithm to use. JACOBI_DAVIDSON, which is the default, is a state-of-the-art package for solving large-scale eigenvalue problems (Sleijpen and van der Vorst 2000). The power method is one of the standard algorithms for solving eigenvalue problems, but it converges slowly for certain problems.

The following example illustrates the use of hub and authority scoring on the directed graph G shown in Figure 1.29. Each node represents a web page. If web page i has a hyperlink that points to web page j, then there is a directed link from i to j.

Figure 1.29 Hub and Authority Centrality Example of a Simple Directed Graph



The graph can be represented using the links data set LinkSetIn as follows:

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

The following statements compute hub and authority centrality:

```
proc optgraph
   graph_direction   = directed
   data_links        = LinkSetIn
   out_nodes         = NodeSetOut;
   centrality
      hub            = unweight
      auth           = unweight;
run;
```

The data set NodeSetOut now contains the hub and authority scores of each node. It is shown in Figure 1.30.

Figure 1.30 Hub and Authority Centrality Output

node	centr_ hub_unwt	centr_ auth_ unwt
B	0.00000	1.00000
C	0.54135	0.00000
D	0.59703	0.11466
A	0.00000	0.10287
E	0.66549	0.84725
F	1.00000	0.11466
G	0.45865	0.00000
H	0.45865	0.00000
I	1.00000	0.00000
J	1.00000	0.00000
K	1.00000	0.00000

The output shows that nodes B and E have high authority scores because they have many incoming links. Nodes F, I, J, K have high hub scores because they all point to good authority nodes B and E.

Weight Interpretation

In certain situations, you might want to calculate various centrality metrics on the same weighted graph. As described above, closeness and betweenness centrality have inverse relationships with the link weights, because these metrics are calculated using shortest paths. So the lower the weight, the higher the contribution to the centrality metric. All of the other metrics are direct relationships. That is, the higher the weight, the higher the contribution to the centrality metric.

To calculate these metrics in one invocation of PROC OPTGRAPH, you can use the WEIGHT2= option. The variable defined by this option is used as link weights for closeness and betweenness calculations whereas all other metrics use the standard weight variable.

For a more detailed example, see "Example 1.6: Centrality Metrics for Project Groups in a Research Department" on page 197, which uses the WEIGHT2= option.

Processing a Subset of Nodes

You might want to calculate centrality metrics for some subset of nodes instead of for the entire node set. This can help reduce the amount of computation when you are interested only in a portion of the nodes. To accomplish this, you can use the DATA_NODES_SUB= option in the PROC OPTGRAPH statement, as described in the section "Node Subset Input Data" on page 55.

Unfortunately, because of the nature of centrality metrics, this is not entirely straightforward in all cases. For centrality metrics that depend on solving eigensystems (eigenvector, hub, and authority), this option is not allowed. If it is used, PROC OPTGRAPH issues a warning and calculates the centrality metrics based on the entire node set. For metrics that depend on shortest paths (closeness and betweenness), this option should be used with caution. For closeness centrality on a directed graph, calculating over just a subset does not work correctly for in-closeness. However, it works fine for undirected graphs. For betweenness centrality, the only shortest paths that contribute to the metric are those with a source in the node subset. Because of this, the betweenness value alone might not be directly useful, unless it is combined with results from other subsets. An example of this is shown below.

Degree and Influence Centrality Using a Node Subset

For clustering coefficients, degree, and influence centrality, using the DATA_NODES_SUB= option is simple. To process a particular node, you indicate a value of 1 in the node's subset data set for the variable centr. The following example processes only nodes 1 and 4:

```
data LinkSetIn;
   input from to @@;
   datalines;
0 1  0 2  0 3
0 4  1 4  3 1
3 2  3 4  4 3
;

data NodeSubSet;
   input node centr;
   datalines;
1 1
4 1
;

proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   data_nodes_sub  = NodeSubSet
   out_nodes       = NodeSetOut;
   centrality
      clustering_coef
      degree       = out
      influence    = unweight;
run;
```

The resulting node data set NodeSetOut contains the degree and influence centrality of just the two nodes indicated in the node subset data set.

Figure 1.31 Degree and Influence Centrality Using a Node Subset

node	centr_ degree_ out	centr_ influence1_ unwt	centr_ influence2_ unwt	centr_ cluster
1	1	0.5	0.5	0
4	1	0.5	0.0	0

Betweenness Centrality Using a Node Subset

Betweenness centrality (unnormalized) gives the number of times a particular node is found in a shortest path. Because of this, you would need to calculate shortest paths for all source-sink pairs. If you want to consider only a subset of source nodes, then the betweenness result gives only a partial count, which is somewhat useless by itself. Despite this, you might still want to calculate betweenness over a subset of nodes. An example of this would be distributing the calculations on a distributed memory computing environment using SAS® Grid Manager. (See *Grid Computing in SAS* for more information.)

The following DATA steps define two node subsets that cover the entire node set:

```
data NodeSubSet1;
   input node centr;
   datalines;
1 1
3 1
;

data NodeSubSet2;
   input node centr;
   datalines;
0 1
2 1
4 1
;
```

The following statements find the betweenness counts for each set separately. This could be done in parallel on different machines (or threads). In this case, BETWEEN_NORM=NO, so you get counts rather than the normalized value.

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   data_nodes_sub  = NodeSubSet1
   out_nodes       = NodeSetOut1;
   centrality
      between      = unweight
      between_norm = no;
run;
```

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   data_nodes_sub  = NodeSubSet2
   out_nodes       = NodeSetOut2;
   centrality
      between      = unweight
      between_norm = no;
run;
```

The resulting node data sets NodeSetOut1 and NodeSetOut2 are shown below. Each of them contains a partial betweenness count based on the nodes that are defined in their respective node subset data sets.

Figure 1.32 Betweenness Centrality Using Node Subset 1

node	centr_ between_ unwt
0	0
1	0
2	0
3	1
4	2

Figure 1.33 Betweenness Centrality Using Node Subset 2

node	centr_ between_ unwt
0	0
1	0
2	0
3	2
4	0

The following statements recover the betweenness counts for the entire node set:

```
data b;
   set NodeSetOut1 NodeSetOut2;
run;

proc sql noprint;
   create table NodeSetOut as
   select distinct node, sum(centr_between_unwt) as centr_between_unwt
      from b
      group by node;
quit;
```

The resulting node data set NodeSetOut now contains the full betweenness count. The final values are equivalent to running against the entire graph in one call to PROC OPTGRAPH (with no subsets).

Figure 1.34 Betweenness Centrality Using a Node Subset

node	centr_ between_ unwt
0	0
1	0
2	0
3	3
4	2

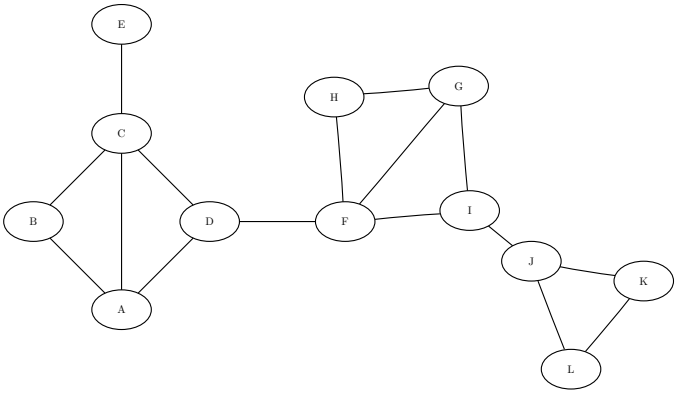
Processing by Cluster

You can process a number of induced subgraphs of a graph with only one call to PROC OPTGRAPH by using the BY_CLUSTER option in the CENTRALITY statement. This section shows an example of how to use this option.

Centrality by Cluster for a Simple Undirected Graph

Consider the graph depicted in Figure 1.35.

Figure 1.35 Undirected Graph



The following statements create the data set LinkSetIn:

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

The graph seems to have three distinct parts, which are connected by just a few links. Assume that you have already partitioned the set into three sets of nodes: $N^0 = \{A, B, C, D, E\}$, $N^1 = \{F, G, H, I\}$, and $N^2 = \{J, K, L\}$. The induced subgraphs on these three sets of nodes are shown in blue in Figure 1.36 through Figure 1.38. Notice that links that connect different partitions have been removed.

Figure 1.36 Subgraph $N^0 = \{A, B, C, D, E\}$

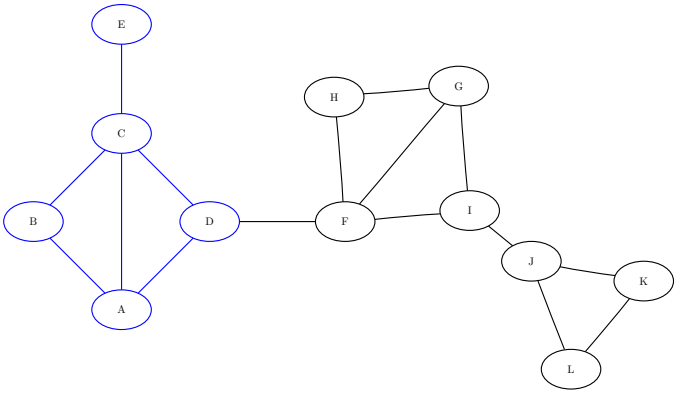


Figure 1.37 Subgraph $N^1 = \{F, G, H, I\}$

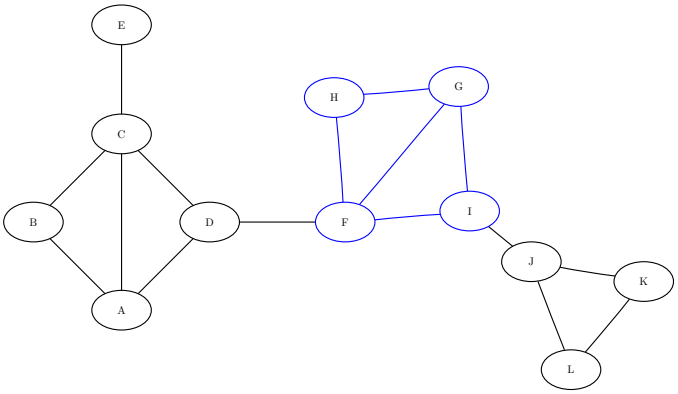
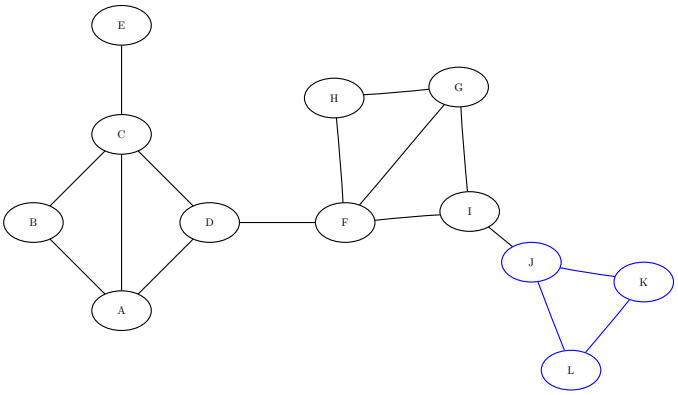


Figure 1.38 Subgraph $N^2 = \{J, K, L\}$



The following data sets define the three induced subgraphs:

```
data LinkSetIn0;
   input from $ to $ @@;
   datalines;
A B  A C  A D  B C  C D  C E
;

data LinkSetIn1;
   input from $ to $ @@;
   datalines;
F G  F H  F I  G H  G I
;

data LinkSetIn2;
   input from $ to $ @@;
   datalines;
J K  J L  K L
;
```

To calculate centrality metrics on the three subgraphs, you could run PROC OPTGRAPH three times, as follows:

```
proc optgraph
   data_links = LinkSetIn0
   out_nodes  = NodeSetOut0;
   centrality
      degree    = out
      influence = unweight
      close     = unweight
      between   = unweight
      eigen     = unweight;
run;

proc optgraph
   data_links = LinkSetIn1
   out_nodes  = NodeSetOut1;
   centrality
      degree    = out
      influence = unweight
      close     = unweight
      between   = unweight
      eigen     = unweight;
run;
```

```
proc optgraph
   data_links = LinkSetIn2
   out_nodes  = NodeSetOut2;
   centrality
      degree    = out
      influence = unweight
      close     = unweight
      between   = unweight
      eigen     = unweight;
run;
```

This produces the results shown in Figure 1.39 through Figure 1.41.

Figure 1.39 Centrality for Induced Subgraph 0

node	centr_ degree_ out	centr_ eigen_ unwt	centr_ close_ unwt	centr_ between_ unwt	centr_ influence1_ unwt	centr_ influence2_ unwt
A	3	0.89897	0.80000	0.08333	0.6	1.6
B	2	0.70711	0.66667	0.00000	0.4	1.4
C	4	1.00000	1.00000	0.58333	0.8	1.6
D	2	0.70711	0.66667	0.00000	0.4	1.4
E	1	0.37236	0.57143	0.00000	0.2	0.8

Figure 1.40 Centrality for Induced Subgraph 1

node	centr_ degree_ out	centr_ eigen_ unwt	centr_ close_ unwt	centr_ between_ unwt	centr_ influence1_ unwt	centr_ influence2_ unwt
F	3	1.00000	1.00	0.16667	0.75	1.75
G	3	1.00000	1.00	0.16667	0.75	1.75
H	2	0.78078	0.75	0.00000	0.50	1.50
I	2	0.78078	0.75	0.00000	0.50	1.50

Figure 1.41 Centrality for Induced Subgraph 2

node	centr_ degree_ out	centr_ eigen_ unwt	centr_ close_ unwt	centr_ between_ unwt	centr_ influence1_ unwt	centr_ influence2_ unwt
J	2	1	1	0	0.66667	1.33333
K	2	1	1	0	0.66667	1.33333
L	2	1	1	0	0.66667	1.33333

A much more efficient way to process these graphs is to define the partition by using the cluster variable in the nodes data set and using the BY_CLUSTER option. Define the partitions of the original graph as follows:

```
data NodeSetIn;
   input node $ cluster @@;
   datalines;
A 0  B 0  C 0  D 0  E 0
F 1  G 1  H 1  I 1
J 2  K 2  L 2
;
```

Now, using one call to PROC OPTGRAPH, you can process all three induced subgraphs. In addition, because the processing of these subgraphs is completely independent, you can do the processing in parallel by using the NTHREADS= option in the PERFORMANCE statement.

```
proc optgraph
   loglevel     = moderate
   data_nodes   = NodeSetIn
   data_links   = LinkSetIn
   out_nodes    = NodeSetOut;
   performance
      nthreads  = 3;
   centrality
      by_cluster
      degree    = out
      influence = unweight
      close     = unweight
      between   = unweight
      eigen     = unweight;
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_CENTR_;
```

Assuming that your machine has at least three cores, all three subgraphs are processed simultaneously with one call to PROC OPTGRAPH. The progress of the procedure is shown in Figure 1.42.

Figure 1.42 PROC OPTGRAPH Log: Centrality by Cluster for a Simple Undirected Graph

```
NOTE: ------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------
NOTE: Running OPTGRAPH version 12.3.
NOTE: ------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------
NOTE: The OPTGRAPH procedure is executing in single-machine mode.
NOTE: ------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: Reading the nodes data set.
NOTE: There were 12 observations read from the data set WORK.NODESETIN.
NOTE: There were 16 observations read from the data set WORK.LINKSETIN.
NOTE: Data input used 0.01 (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 of memory.
NOTE: The number of nodes in the input graph is 12.
NOTE: The number of links in the input graph is 16.
NOTE: ------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------
NOTE: Processing CENTRALITY statement with BY_CLUSTER option.
NOTE: ------------------------------------------------------------------------
NOTE: Using CLUSTER variable from DATA_NODES to partition the graph.
NOTE: Links that cross subgraphs will be ignored.
NOTE: ------------------------------------------------------------------------
NOTE: Distribution of 3 subgraphs by number of nodes:
             3 subgraphs of size [     3,    10] (100.0%)
NOTE: ------------------------------------------------------------------------
NOTE: Processing centrality by subgraph using 3 threads.
                                                      Cpu      Real   Active
      Algorithm              SubGraphs   Complete     Time      Time  Threads
      centrality                     3       100%     0.00      0.00        0
NOTE: The centrality algorithms are using 0.5 MBs of memory.
NOTE: Processing centrality by subgraph used 0.00 (cpu: 0.00) seconds.
NOTE: ------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------
NOTE: Creating nodes data set output.
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------
NOTE: The data set WORK.NODESETOUT has 12 observations and 8 variables.
STATUS=OK  CENTR=OK
STATUS=OK  CPU_TIME=0.00  REAL_TIME=0.00
```

The results are shown in Figure 1.43.

Figure 1.43 Centrality for All Induced Subgraphs

node	cluster	centr_ degree_ out	centr_ eigen_ unwt	centr_ close_ unwt	centr_ between_ unwt	centr_ influence1_ unwt	centr_ influence2_ unwt
A	0	3	0.89897	0.80000	0.08333	0.60000	1.60000
B	0	2	0.70711	0.66667	0.00000	0.40000	1.40000
C	0	4	1.00000	1.00000	0.58333	0.80000	1.60000
D	0	2	0.70711	0.66667	0.00000	0.40000	1.40000
E	0	1	0.37236	0.57143	0.00000	0.20000	0.80000
F	1	3	1.00000	1.00000	0.16667	0.75000	1.75000
G	1	3	1.00000	1.00000	0.16667	0.75000	1.75000
H	1	2	0.78078	0.75000	0.00000	0.50000	1.50000
I	1	2	0.78078	0.75000	0.00000	0.50000	1.50000
J	2	2	1.00000	1.00000	0.00000	0.66667	1.33333
K	2	2	1.00000	1.00000	0.00000	0.66667	1.33333
L	2	2	1.00000	1.00000	0.00000	0.66667	1.33333

Centrality by Community for a Simple Undirected Graph

The partition defined in the data set NodeSetIn could have also been calculated by PROC OPTGRAPH using a method called *community detection*. This method is discussed in the section "Community" on page 92. First, call the community detection method as follows:

```
proc optgraph
   data_links = LinkSetIn
   out_nodes  = Communities;
   community;
run;
```

The resulting output is a partition of the nodes of the original graph into *communities*. The Communities data set is shown in Figure 1.44.

Figure 1.44 Communities for a Simple Undirected Graph

node	community_ 1
A	0
B	0
C	0
D	0
E	0
F	1
G	1
H	1
I	1
J	2
K	2
L	2

To calculate centrality by induced subgraph, you can simply use the communities output as the nodes data set input and use the DATA_NODES_VAR statement to define the cluster variable:

```
proc optgraph
   data_nodes    = Communities
   data_links    = LinkSetIn
   out_nodes     = NodeSetOut;
   data_nodes_var
      cluster    = community_1;
   performance
      nthreads   = 3;
   centrality
      by_cluster
      degree     = out
      influence  = unweight
      close      = unweight
      between    = unweight
      eigen      = unweight;
run;
```

This gives the same results as before, when you manually defined the partition. These results are shown in Figure 1.43.

Centrality by Filtered Community for a Simple Undirected Graph

In some situations, the community detection algorithm might find a large number of small communities. Those communities might not be relevant, and you might want to focus only on communities of a certain size. When you use the BY_CLUSTER option, you can also use the FILTER_SUBGRAPH= option to ignore any subgraph whose number of nodes is less than or equal to a certain size. This can save on computation time, and the resulting output contains only the subgraphs of interest.

Returning to the data in the section "Centrality by Community for a Simple Undirected Graph" on page 86, you can use the filtering option as follows:

```
proc optgraph
   filter_subgraph = 3
   data_nodes      = Communities
   data_links      = LinkSetIn
   out_nodes       = NodeSetOut;
   data_nodes_var
      cluster      = community_1;
   performance
      nthreads     = 3;
   centrality
      by_cluster
      degree       = out
      influence    = unweight
      close        = unweight
      between      = unweight
      eigen        = unweight;
run;
```

The results, shown in Figure 1.45, now contain only those subgraphs with node size greater than 3.

Figure 1.45 Centrality for Some Induced Subgraphs

node	community_ 1	centr_ degree_ out	centr_ eigen_ unwt	centr_ close_ unwt	centr_ between_ unwt	centr_ influence1_ unwt	centr_ influence2_ unwt
A	0	3	0.89897	0.80000	0.08333	0.60	1.60
B	0	2	0.70711	0.66667	0.00000	0.40	1.40
C	0	4	1.00000	1.00000	0.58333	0.80	1.60
D	0	2	0.70711	0.66667	0.00000	0.40	1.40
E	0	1	0.37236	0.57143	0.00000	0.20	0.80
F	1	3	1.00000	1.00000	0.16667	0.75	1.75
G	1	3	1.00000	1.00000	0.16667	0.75	1.75
H	1	2	0.78078	0.75000	0.00000	0.50	1.50
I	1	2	0.78078	0.75000	0.00000	0.50	1.50

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 given 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 25. This algorithm works only with undirected graphs.

The results for 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.

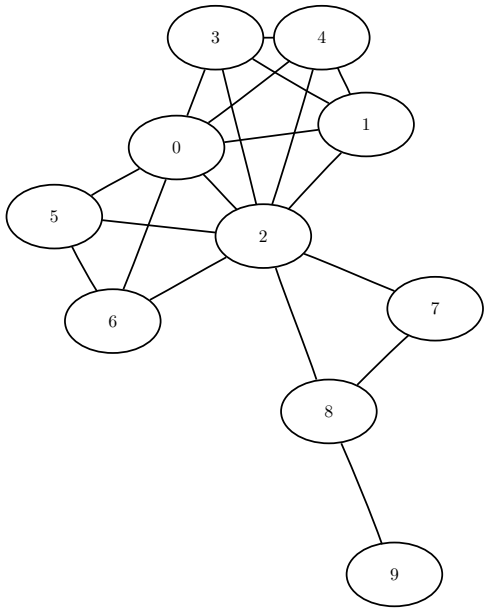
The clique algorithm reports status information in a macro variable called _OPTGRAPH_CLIQUE_. See the section "Macro Variable _OPTGRAPH_CLIQUE_" on page 173 for more information about this macro variable.

The algorithm used by PROC OPTGRAPH 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 shown in Figure 1.46.

Figure 1.46 A Simple Undirected Graph G



The undirected graph G can be represented by the links data set LinkSetIn as follows:

```
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 optgraph
   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 1.47.

Figure 1.47 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 1.48.

Figure 1.48 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 1.49 and Figure 1.50.

Figure 1.49 Maximal Cliques C^1 and C^2

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

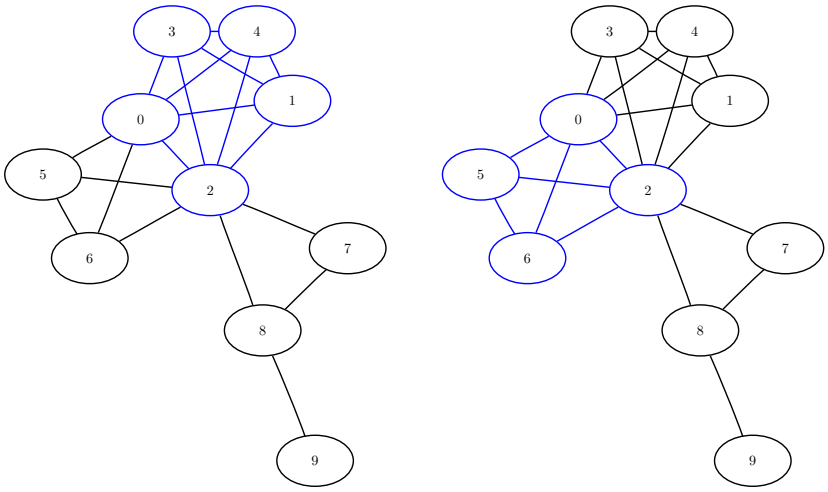
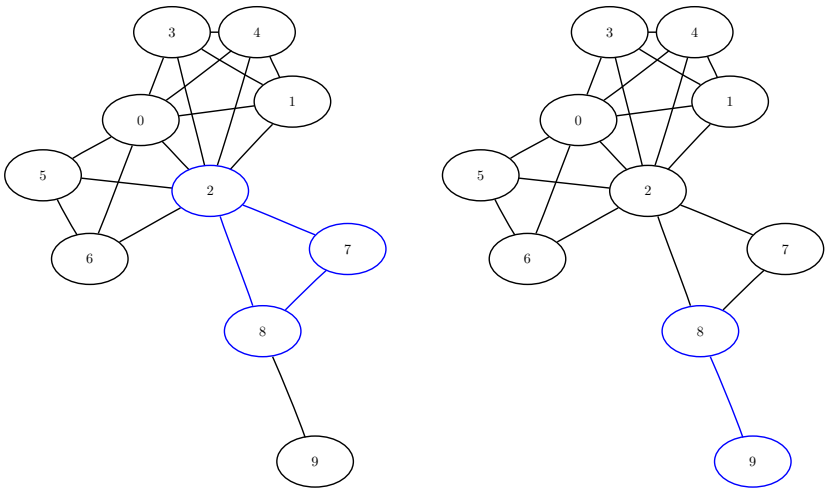


Figure 1.50 Maximal Cliques C^2 and C^3

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



Community

Community detection partitions a graph into communities such that the links within the community subgraphs are more densely connected than the links between communities.

In PROC OPTGRAPH, community detection can be determined by using the COMMUNITY statement. The options for this statement are described in the section "COMMUNITY Statement" on page 26.

The COMMUNITY statement reports status information in a macro variable called _OPTGRAPH_COMMUNITY_. See the section "Macro Variable _OPTGRAPH_COMMUNITY_" on page 174 for more information about this macro variable.

When you specify ALGORITHM=PARALLEL_LABEL_PROP in the COMMUNITY statement, community detection supports both undirected and directed graphs. When you specify ALGORITHM=LOUVAIN or ALGORITHM=LABEL_PROP in the COMMUNITY statement, community detection is supported only for undirected graphs. For directed graphs, you need to aggregate directed links into undirected links before you call the algorithm. For example, suppose there are two directed links: a link from i to j with a link weight of 4.3, and a link from j to i with a link weight of 3.2. One common aggregation strategy is to sum the link weights together. Using this strategy, the weight of the undirected link between i and j is 7.5.

PROC OPTGRAPH implements three heuristic algorithms for finding communities: the LOUVAIN algorithm proposed in Blondel et al. (2008), the label propagation algorithm proposed in Raghavan, Albert, and Kumara (2007), and the parallel label propagation algorithm developed by SAS (patent pending).

Given a graph $G = (N, A)$, all three algorithms run in time $O(k|A|)$, where k is the average number of links per node. The Louvain algorithm aims to optimize modularity, which is one of the most popular merit functions for community detection. Modularity is a measure of the quality of a division of a graph into communities. The *modularity* of a division is defined to be the fraction of the links that fall within the communities minus the expected fraction if the links were distributed at random, assuming that you do not change the degree of each node.

Mathematically, modularity is defined as

$$
\begin{aligned}
Q &= \frac{1}{2w} \sum_{(u,v)\in A} \left(w_{uv} - \frac{w_u w_v}{2w} \right) \Delta(c_u, c_v) \\
w &= \sum_{(u,v)\in A} w_{uv} \\
w_u &= \sum_{v\in \delta_u} w_{uv}
\end{aligned}
$$

where Q is the modularity, w_{uv} is the link weight between node u and v, δ_u is the set of nodes that connects to node u, w_u is the sum of link weights incident to node u, w is the sum of link weights of the graph, c_u is the community to which node u belongs, and $\Delta(c_u, c_v)$ is the Kronecker delta symbol, defined as

$$
\Delta(c_u, c_v) = \begin{cases} 1 & \text{if } c_u = c_v \\ 0 & \text{otherwise} \end{cases}
$$

The following is a brief description of the Louvain algorithm:

1. Initialize each node as its own community.
2. Move each node from its current community to the neighboring community that increases modularity the most. Repeat this step until modularity cannot be improved.
3. Group the nodes in each community into a supernode. Construct a new graph based on supernodes. Repeat these steps until modularity cannot be further improved or the maximum number of iterations has been reached.

The more recently proposed label propagation algorithm moves a node to a community that most of its neighbors belong to. Extensive testing by Lancichinetti and Fortunato (2009) has empirically demonstrated that the label propagation algorithm performs as well as the Louvain method in most cases.

The following is a brief description of the label propagation algorithm:

1. Initialize each node as its own community.
2. Move each node from its current community to the neighboring community that has the maximum number of nodes; break ties randomly if necessary. Repeat this step until there are no more movements.
3. Group the nodes in each community into a supernode. Construct a new graph based on supernodes. Repeat these steps until there are no more movements or the maximum number of iterations has been reached.

The parallel label propagation algorithm is an extension of the basic label propagation algorithm. During each iteration, rather than updating node labels sequentially, nodes update their labels simultaneously by using the node label information from the previous iteration. In this approach, node labels can be updated in parallel. However, simultaneous updating of this nature often leads to oscillating labels because of the bipartite subgraph structure often present in large graphs. To address this issue, at each iteration the parallel algorithm skips the labeling step at some randomly chosen nodes in order to break the bipartite structure. You can control the random samples that the algorithm takes by specifying the RANDOM_FACTOR= or RANDOM_SEED= options in the COMMUNITY statement.

As you can see from their descriptions, all three algorithms adopt a heuristic local optimization approach. The final result often depends on the sequence of nodes that are presented in the links input data set. Therefore, if the sequence of nodes in the links data set has been changed, the result is likely to be slightly different.

Parallel Community Detection

Parallel community detection can be invoked by specifying ALGORITHM=PARALLEL_LABEL_PROP in the COMMUNITY statement. The computation is executed with multiple threads on a single computer. The number of threads being used can be controlled by specifying the NTHREADS= option in the PERFORMANCE statement.

The following statements demonstrate how to invoke parallel community detection using eight threads:

```
proc optgraph
   data_links       = links
   graph_direction  = directed
   out_nodes        = outNodes;
   performance
      nthreads      = 8;
   community
      algorithm     = parallel_label_prop
      out_community = outComm;
run;
```

Memory Requirement

When you specify GRAPH_INTERNAL_FORMAT=THIN in the PROC OPTGRAPH statement and ALGORITHM=LOUVAIN or ALGORITHM=LABEL_PROP in the COMMUNITY statement, the memory (number of bytes) required for community detection can be estimated approximately as follows given a graph $G = (N, A)$:

$$(2 \times |A| + |N|) \times \text{sizeof(int)} + (3 \times |A| + |N|) \times \text{sizeof(double)}$$

When you specify GRAPH_INTERNAL_FORMAT=THIN and ALGORITHM=PARALLEL_LABEL_PROP, the memory required for community detection is approximately twice this amount.

Assume that your machine architecture is such that an integer is 4 bytes and a double is 8 bytes. Then, a graph with 100 million nodes and 650 million links would require approximately 21 gigabytes (GB) of memory when you specify ALGORITHM=LOUVAIN or ALGORITHM=LABEL_PROP:

$$(2 \times 650M + 100M) \times 4 + (3 \times 650M + 100M) \times 8 = 21GB$$

The same graph would require approximately 42 GB if you specify ALGORITHM=PARALLEL_LABEL_PROP.

This is only an estimate for the amount of memory that is required. PROC OPTGRAPH itself might require more memory to maintain the input and output data structures. In addition, other running processes might take away from the available memory.

PROC OPTGRAPH uses significantly more memory if GRAPH_INTERNAL_FORMAT=FULL. It is recommended that you use GRAPH_INTERNAL_FORMAT=THIN when you apply community detection on large graphs.

Graph Direction

If you specify ALGORITHM=PARALLEL_LABEL_PROP in the COMMUNITY statement, community detection supports both undirected and directed graphs. However, you should be careful in deciding whether to model your problem as an undirected or a directed graph. For an undirected graph, the algorithm finds communities based on the density of the subgraphs. For a directed graph, the algorithm finds communities based on the information flow along the directed links. That is, the algorithm propagates the community ID along the outgoing links of a node. Therefore, nodes are likely to be in the same community if they form circles along the outgoing links. If the directed graph lacks this circle structure, the nodes are likely to switch between communities during the computation. As a result, the algorithm does not converge well and cannot find a good community structure in the graph.

Large Community

It has often been observed in practice that the number of nodes contained in communities (produced by community detection algorithms) usually follows a power law distribution. That is, a few communities contain a very large number of nodes, whereas most communities contain a small number of nodes. This is especially true for large graphs. PROC OPTGRAPH provides two approaches to alleviate this problem: one uses the RECURSIVE option, and the other uses the RESOLUTION_LIST= option.

Recursive

You can apply the RECURSIVE option to recursively break large communities into smaller ones. At the first step, PROC OPTGRAPH processes data as if no RECURSIVE option were specified. At the end of this step, it checks whether the community result satisfies the RECURSIVE option criteria. If the community result satisfies these criteria, PROC OPTGRAPH stops iterations and outputs results. Otherwise, it treats each large community as an independent graph and recursively applies community detection on top of it.

In certain cases, a community is not further split even if it does not meet the recursive criteria that you specified. One example is a star-shaped community that contains 200 nodes while MAX_COMM_SIZE is specified as 100; another example is a symmetric community whose diameter is 2 while MAX_DIAMETER is specified as 1.

Resolution List

The second way to combat the problem, provided you have specified ALGORITHM=LOUVAIN in the COMMUNITY statement, is to assign a larger value than the default value of 1 to the RESOLUTION_LIST= option. When ALGORITHM=LOUVAIN, the value assigned to the RESOLUTION_LIST= option can be interpreted as follows: Suppose the resolution value is x. Two communities are merged if the sum of the weights of intercommunity links is at least x times the expected value of the same sum if the graph is reconfigured randomly. Therefore, a larger resolution value produces more communities, each of which contains a smaller number of nodes. However, there is no explicit formula to detail the number of nodes in communities with respect to the resolution value. You must use trial and error to get to the expected community size. More information about resolution value is available in Ronhovde and Nussinov 2010.

If you specify ALGORITHM=LOUVAIN, you can supply multiple resolution values at one time. If you supply multiple resolution values at one time, PROC OPTGRAPH detects communities at the highest resolution level first, then merges communities at a lower resolution, and repeats the process until it reaches the lowest level. This process enables you to see how the communities are merged at different levels. Due to the local nature of this optimization algorithm, two different runs do not produce the same result if the two runs share a common resolution level. For example, the algorithm can produce different results at resolution 0.5 in two runs: one with RESOLUTION_LIST = 1 0.7 0.5, and the other with RESOLUTION_LIST = 1 0.5.

If you specify ALGORITHM=PARALLEL_LABEL_PROP in the COMMUNITY statement, the resolution value can be interpreted as the minimal density of communities in an undirected and unweighted graph. The *density* of a community is defined as the number of links inside the community divided by the total number of possible links. A larger resolution value likely results in communities that contain fewer nodes. For more information about resolution values for label propagation, see Traag, Van Dooren, and Nesterov (2011).

If you supply multiple resolution values at one time and you specify ALGORITHM=PARALLEL_LABEL_PROP, the OPTGRAPH procedure performs community detection multiple times, each time with a different resolution value. This is equivalent to calling the OPTGRAPH procedure several times, each time with a different (single) resolution value specified for the RESOLUTION_LIST= option.

If you specify ALGORITHM=PARALLEL_LABEL_PROP in the COMMUNITY statement, the value that is specified in the RESOLUTION_LIST= option has a major impact on the running time of the algorithm. When a large resolution value is specified, the algorithm is likely to create many tiny communities, and nodes are likely to change communities between iterations. Therefore, the algorithm might not converge properly. On the other hand, when the resolution value is small, the algorithm might find some very large communities, such as a community that contains more than a million nodes. In this case, if you specify the RECURSIVE option, the algorithm spends a long time in the recursive step in order to break large communities into smaller ones.

The recommended approach is to first experiment with a set of resolution values without using the RECURSIVE option. At the end of the run, examine the resulting modularity values and the community size distributions. Remove the resolution values that lead to small modularity values or huge communities. Then add the RECURSIVE option to the COMMUNITY statement, if desired, and run PROC OPTGRAPH again.

Example 1.7 shows the use of the RESOLUTION_LIST= option in the calculation of communities.

Large Graphs

When you are dealing with large graphs, the following practices are recommended:

- Use GRAPH_INTERNAL_FORMAT=THIN instead of GRAPH_INTERNAL_FORMAT=FULL. This enables PROC OPTGRAPH to store the data in memory compactly.
- Use the LINK_REMOVAL_RATIO= option to remove unimportant links. This practice can often dramatically improve the running time of large graphs.

Output Data Sets

Community detection produces up to five output data sets. In these data sets, if you specify ALGORITHM=LOUVAIN or ALGORITHM=LABEL_PROP in the COMMUNITY statement, resolution level numbers are in decreasing order of the values that are specified in the RESOLUTION_LIST= option. That is, resolution level 1 corresponds to the largest value specified in the RESOLUTION_LIST= option, and resolution level K corresponds to the smallest value specified in the RESOLUTION_LIST= option. For example, if RESOLUTION_LIST=2.5 3.1 0.6, then resolution level 1 is at value 3.1, resolution level 2 is at value 2.5, and resolution level 3 is at value 0.6.

If you specify ALGORITHM=PARALLEL_LABEL_PROP in the COMMUNITY statement, resolution level numbers are in the same order as the values that are specified in the RESOLUTION_LIST= option. For example, if RESOLUTION_LIST=0.001 0.005 0.01, then resolution level 1 is at value 0.001, resolution level 2 is at value 0.005, and resolution level 3 is at value 0.01.

OUT_NODES= Data Set

This data set describes the community identifier of each node. If multiple resolution values have been specified, the data set reports the community identifier of each node at each resolution level. The data set contains the following columns:

- node: node label
- community_i: community identifier at resolution level i, where i is the resolution level number as previously described. There are K such columns if K different values are specified in the RESOLUTION_LIST= option.

OUT_LEVEL= Data Set

This data set describes the number of communities and their corresponding modularity values at various resolution levels. It contains the following columns:

- level: resolution level number
- resolution: resolution value
- communities: number of communities at the current resolution level
- modularity: modularity value at the current resolution level

OUT_COMMUNITY= Data Set

This data set describes the number of nodes in each community. It contains the following columns:

- level: resolution level number
- resolution: resolution value
- community: community identifier
- nodes: number of nodes contained in the community

OUT_OVERLAP= Data Set

This data set describes the intensity of a node that belongs to multiple communities. At the end of community detection, a node could have links that connect to multiple communities. The intensity of a node that belongs to community i is computed as the sum of the weights of links that connect community i divided by the total link weights of the node. This data set is computationally expensive to produce, and it requires a large amount of disk space. Therefore, this data set is not produced if you specify multiple resolution values in the RESOLUTION_LIST= option. The data set contains the following columns:

- node: node label
- community: community identifier
- intensity: intensity of the node that belongs to the community

OUT_COMM_LINKS= Data Set

This data set describes how communities are connected. If you specify ALGORITHM=LOUVAIN or ALGORITHM=LABEL_PROP in the COMMUNITY statement, this data set contains the following columns:

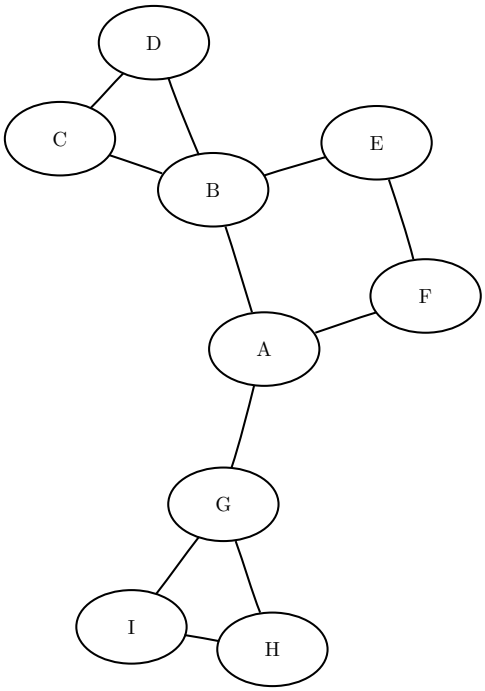
- level: resolution level number
- resolution: resolution value
- from_community: community identifier of the *from* community
- to_community: community identifier of the *to* community
- link_weight: sum of link weights of all links between from_community and to_community

If you specify ALGORITHM=PARALLEL_LABEL_PROP in the COMMUNITY statement, this data set contains the from, to, and link_weight columns. This data set is not produced if you specify ALGORITHM=PARALLEL_LABEL_PROP together with multiple resolution values in the RESOLUTION_LIST= option.

Community Detection on a Simple Graph

This section illustrates the use of the community detection algorithm on the simple undirected graph *G* shown in Figure 1.51.

Figure 1.51 A Simple Undirected Graph *G*



The undirected graph *G* can be represented using the links data set LinkSetIn as follows:

```
data LinkSetIn;
   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 perform community detection and output the results in the specified data sets. The Louvain algorithm is used by default because no value is specified for the ALGORITHM= option.

```
proc optgraph
   data_links    = LinkSetIn
   out_nodes     = NodeSetOut;
   community
      resolution_list = 1.0 0.5
      out_level       = CommLevelOut
      out_community   = CommOut
      out_overlap     = CommOverlapOut
      out_comm_links  = CommLinksOut;
run;
```

The data set NodeSetOut contains the community identifier of each node and is shown in Figure 1.52.

Figure 1.52 Community Detection on a Simple Graph: Nodes Output

node	community_ 1	community_ 2
A	0	0
B	1	0
F	0	0
G	2	1
C	1	0
D	1	0
E	0	0
I	2	1
H	2	1

The data set CommLevelOut contains summary information at each resolution level and is shown in Figure 1.53.

Figure 1.53 Community Detection on a Simple Graph: Level Output

level	resolution	communities	modularity
1	1.0	3	0.39256
2	0.5	2	0.34298

The data set CommOut contains the number of nodes in each community and is shown in Figure 1.54.

Figure 1.54 Community Detection on a Simple Graph: Community Summary

level	resolution	community	nodes
1	1.0	0	3
1	1.0	1	3
1	1.0	2	3
2	0.5	0	6
2	0.5	1	3

The data set CommOverlapOut contains community overlap information and is shown in Figure 1.55.

Figure 1.55 Community Detection on a Simple Graph: Community Overlap

node	community	intensity
A	0	0.66667
A	1	0.33333
B	0	1.00000
F	0	1.00000
G	0	0.33333
G	1	0.66667
C	0	1.00000
D	0	1.00000
E	0	1.00000
I	1	1.00000
H	1	1.00000

The data set CommLinksOut describes how the communities are connected and is shown in Figure 1.56.

Figure 1.56 Community Detection on a Simple Graph: Intercommunity Links

level	resolution	from_ community	to_community	link_ weight
1	1.0	0	1	2
1	1.0	0	2	1
2	0.5	0	1	1

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 exists between any two nodes.

In PROC OPTGRAPH, connected components can be invoked by using the CONCOMP statement. The options for this statement are described in the section "CONCOMP Statement" on page 28.

There are two main algorithms for finding connected components on an undirected graph: a depth-first search algorithm (ALGORITHM=DFS) and a union-find algorithm (ALGORITHM=UNION_FIND). Given a graph $G = (N, A)$, both algorithms run in time $O(|N| + |A|)$ and typically can scale to very large graphs. The default, depth-first search, works only with a full graph structure (GRAPH_INTERNAL_FORMAT=FULL) and for this reason can sometimes be slower than the union-find algorithm. For directed graphs, the only algorithm available is depth-first search.

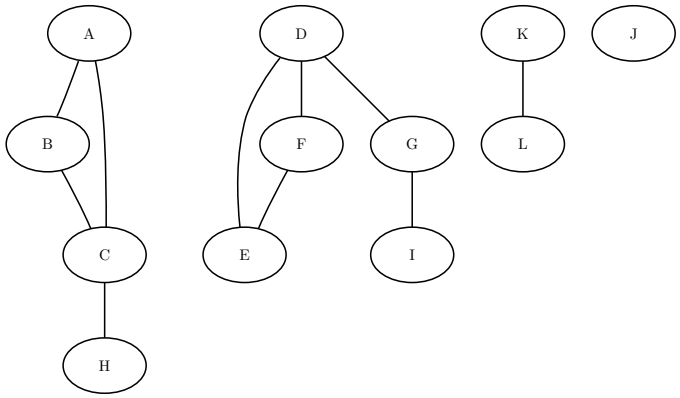
The results for the connected components algorithm are written to the output node data set that is specified in the OUT_NODES= option in the PROC OPTGRAPH statement. For each node in the node data set, the variable concomp identifies its component. The component identifiers are numbered sequentially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

The connected components algorithm reports status information in a macro variable called _OPTGRAPH_CONCOMP_. See the section "Macro Variable _OPTGRAPH_CONCOMP_" on page 174 for more information about this macro variable.

Connected Components of a Simple Undirected Graph

This section illustrates the use of the connected components algorithm on the simple undirected graph G shown in Figure 1.57.

Figure 1.57 A Simple Undirected Graph G



The undirected graph G can be represented by the links data set LinkSetIn as follows:

```
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 optgraph
   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 1.58.

Figure 1.58 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 was defined by using only the links data set. As seen in Figure 1.57, 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 with the links data set alone, it did not show up in the results data set. To define a graph with 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 and links input data sets:

```
proc optgraph
   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 1.59.

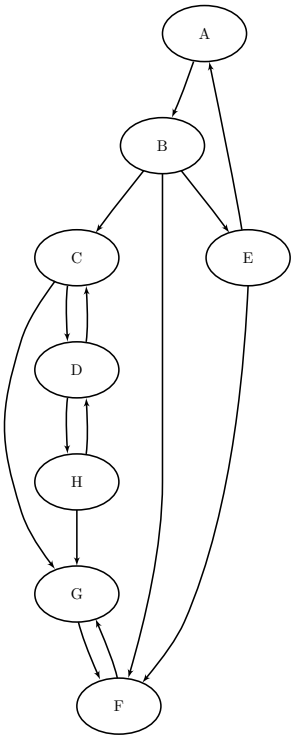
Figure 1.59 Connected Components of a Simple Undirected Graph

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 shown in Figure 1.60.

Figure 1.60 A Simple Directed Graph G



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

```
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 optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   out_nodes       = NodeSetOut;
   concomp;
run;
```

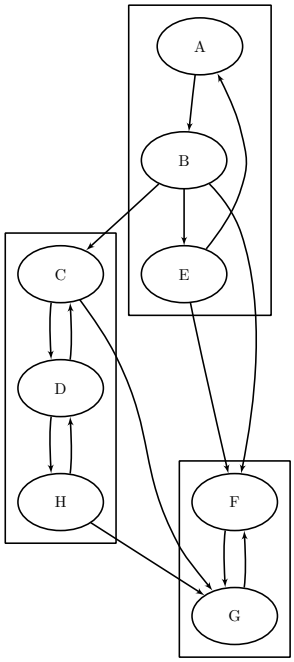
The data set NodeSetOut, shown in Figure 1.61, now contains the connected components of the input graph.

Figure 1.61 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 1.62.

Figure 1.62 Strongly Connected Components of G



Core Decomposition

An alternative to community detection for detecting cohesive subgroups is a method for extracting *k-cores*, known as *core decomposition*. Although this method is generally not as powerful as community detection for extracting a detailed community structure, it can give a coarse approximation of cohesive structure at a very low computational cost. Let $G = (N, A)$ define a graph with nodes N and links A, and let $G_S = (S, A_S)$ be an induced subgraph on nodes S. The subgraph G_S is a *k-core* if and only if for every node $v \in S$, the degree of v is greater than or equal to k and G_S is the maximum subgraph with this property. By definition, the cores are nested. That is, if G_{S_k} is a k-core of size k, then $G_{S_{k+1}}$ is contained in G_{S_k}.

In PROC OPTGRAPH, core decomposition can be invoked by using the CORE statement. The options for this statement are described in the section "CORE Statement" on page 29.

The results for the core decomposition algorithm are given in the output node data set that is specified in the OUT_NODES= option in the PROC OPTGRAPH statement. For each node in the node data set, the variable core_out identifies its *core number*, the highest-order core that contains this node. The core identifiers are numbered sequentially starting from 0.

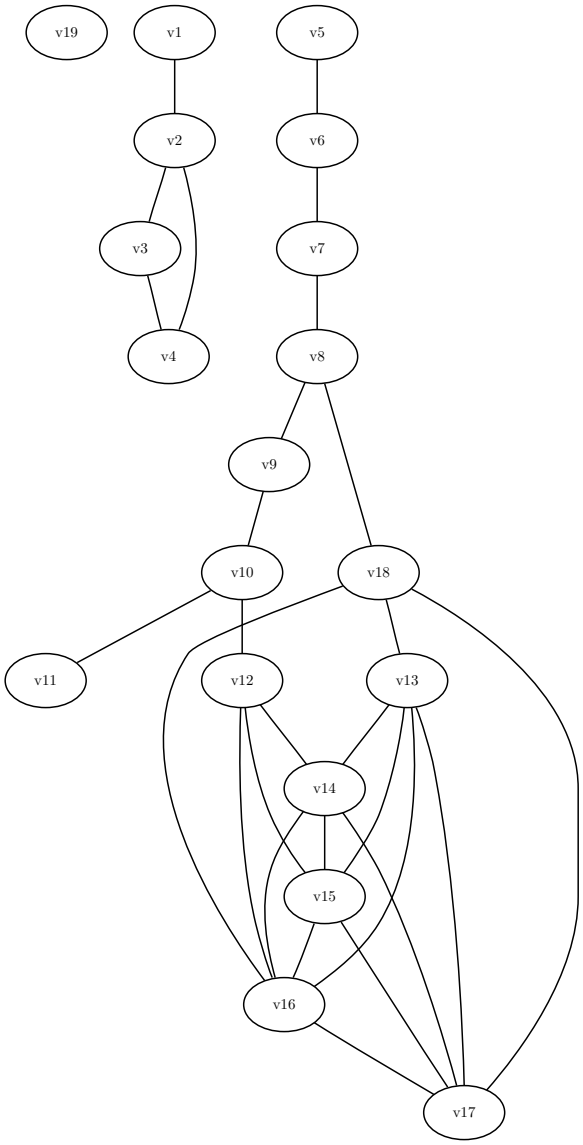
The core decomposition algorithm reports status information in a macro variable called _OPTGRAPH_CORE_. See the section "Macro Variable _OPTGRAPH_CORE_" on page 175 for more information about this macro variable.

The algorithm used for core decomposition is based on the work presented in Batagelj and Zaversnik 2003. This algorithm runs in time $O(|A|)$ and therefore should scale to very large graphs.

Core Decomposition of a Simple Undirected Graph

This section illustrates the use of the core decomposition algorithm on the simple undirected graph G shown in Figure 1.63.

Figure 1.63 Simple Undirected Graph



The undirected graph G can be represented using the nodes data set NodeSetIn and the links data set LinkSetIn as follows:

```
data NodeSetIn;
   input node $ @@;
   datalines;
v1    v2   v3   v4   v5
v6    v7   v8   v9  v10
```

```
v11  v12  v13  v14  v15
v16  v17  v18  v19
;

data LinkSetIn;
   input from $ to $ @@;
   datalines;
v1   v2   v5  v6   v6  v7   v7  v8  v10 v11
v2   v3   v3  v4   v2  v4   v8  v9   v9 v10
v8  v18  v10 v12  v13 v14  v13 v15  v13 v16
v13 v17  v14 v15  v14 v16  v14 v17  v15 v16
v15 v17  v16 v17  v18 v13  v18 v17  v18 v16
v12 v14  v12 v15  v12 v16
;
```

The following statements calculate the core decomposition and output the results in the data set NodeSetOut:

```
proc optgraph
   data_nodes = NodeSetIn
   data_links = LinkSetIn
   out_nodes  = NodeSetOut;
   core;
run;
```

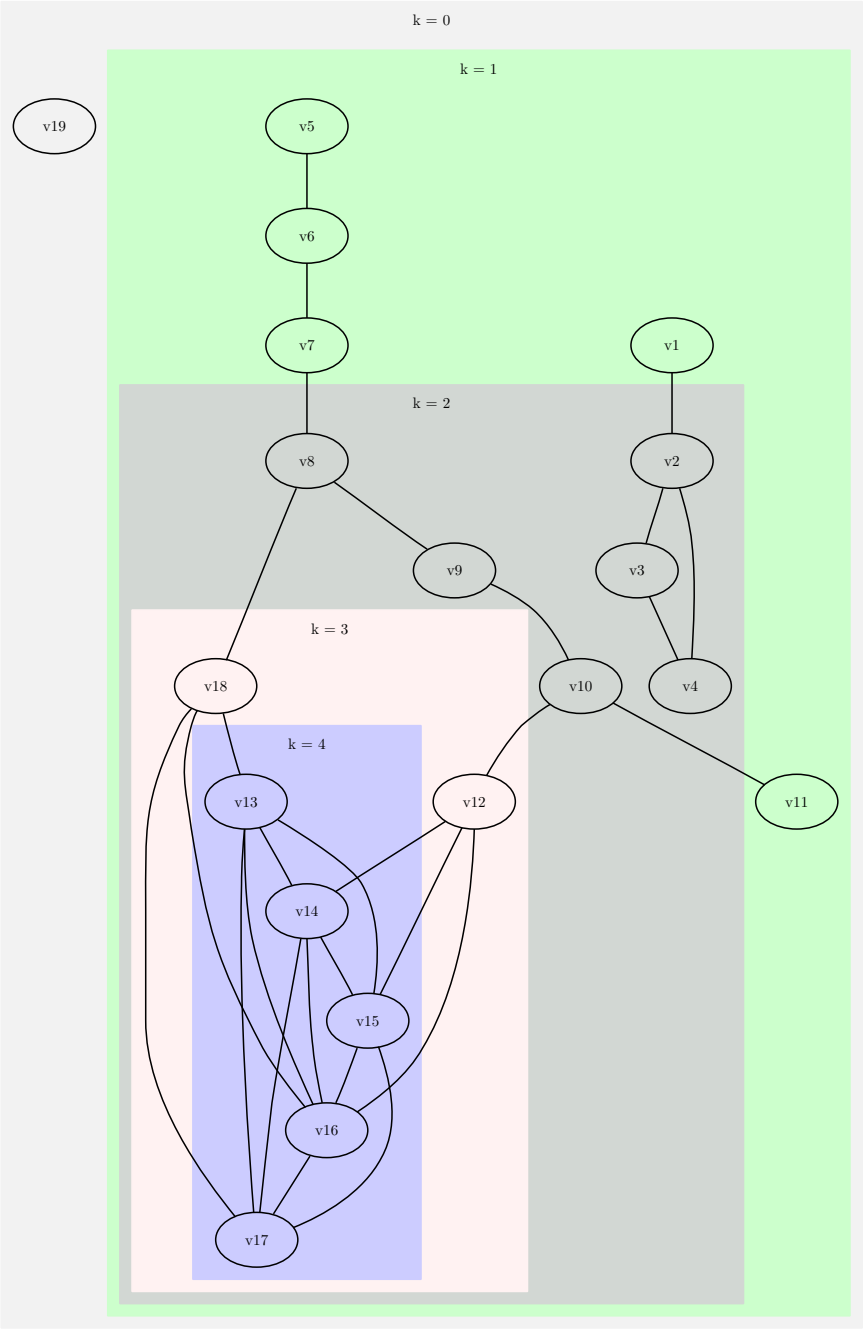
The node data set NodeSetOut contains the core number (variable core_out) for each node and is shown in Figure 1.64.

Figure 1.64 Core Decomposition of a Simple Undirected Graph

node	core_out
v19	0
v1	1
v5	1
v6	1
v7	1
v11	1
v2	2
v3	2
v4	2
v8	2
v9	2
v10	2
v12	3
v18	3
v13	4
v14	4
v15	4
v16	4
v17	4

Figure 1.65 shows the graph layered by its core number.

Figure 1.65 Core Decomposition



Cycle

A *path* in a graph is a sequence of nodes, each of which has a link to the next node in the sequence. A *cycle* is a path in which the start node and end node are the same.

In PROC OPTGRAPH, you can find cycles (or just count the cycles) of an input graph by invoking the CYCLE statement. The options for this statement are described in the section "CYCLE Statement" on page 30. 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 for the cycle detection algorithm are written to the output data set that is specified 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 _OPTGRAPH_CYCLE_. See the section "Macro Variable _OPTGRAPH_CYCLE_" on page 175 for more information about this macro variable.

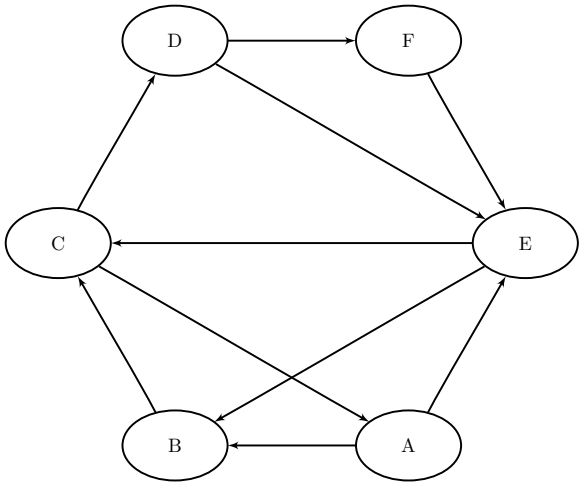
The algorithm used by PROC OPTGRAPH to compute all cycles is a variant of the algorithm found in Johnson 1975. This algorithm runs in time $O((|N| + |A|)(c + 1))$, where c is the number of elementary cycles in the graph. So, the algorithm should scale to large graphs that contain few cycles. However, 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 found and stops processing. This should run relatively quickly. On large-scale graphs, the MINLINKWEIGHT= and MAXLINKWEIGHT= options can be relatively expensive and might increase the computation time. See the section "CYCLE Statement" on page 30 for more information about these options.

Cycle Detection of a Simple Directed Graph

This section provides a simple example for using the cycle detection algorithm on the simple directed graph G shown in Figure 1.66. Two other examples are Example 1.9, which shows the use of cycle detection for optimizing a kidney donor exchange, and Example 1.13, which shows another application of cycle detection.

Figure 1.66 A Simple Directed Graph G



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

```
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 optgraph
   graph_direction = directed
   data_links      = LinkSetIn;
   cycle
      mode         = first_cycle;
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_CYCLE_;
```

The result is written to the log of the procedure, as shown Figure 1.67.

Figure 1.67 PROC OPTGRAPH Log: Check the Existence of a Cycle in a Simple Directed Graph

```
NOTE: ---------------------------------------------------------------------------
NOTE: Running OPTGRAPH version 12.3.
NOTE: ---------------------------------------------------------------------------
NOTE: The OPTGRAPH procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------
NOTE: Data input used 0.01 (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 statement.
NOTE: The graph does have a cycle.
NOTE: Processing cycles 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 optgraph
   graph_direction = directed
   data_links      = LinkSetIn;
   cycle
      mode         = all_cycles;
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_CYCLE_;
```

The result is written to the log of the procedure, as shown in Figure 1.68.

Figure 1.68 PROC OPTGRAPH Log: Count the Number of Cycles in a Simple Directed Graph

```
NOTE: ---------------------------------------------------------------------------
NOTE: Running OPTGRAPH version 12.3.
NOTE: ---------------------------------------------------------------------------
NOTE: The OPTGRAPH procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------
NOTE: Data input used 0.01 (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 statement.
NOTE: The graph has 7 cycles.
NOTE: Processing cycles 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 optgraph
   graph_direction = directed
   data_links      = LinkSetIn;
   cycle
      out          = Cycles
      mode         = first_cycle;
run;
```

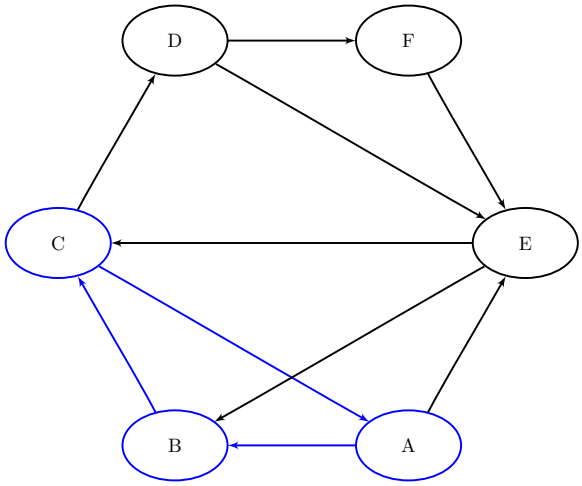
The data set Cycles now contains the first cycle found in the input graph; it is shown in Figure 1.69.

Figure 1.69 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 found in the input graph is shown graphically in Figure 1.70.

Figure 1.70 $A \rightarrow B \rightarrow C \rightarrow A$



The following statements return all of the cycles in the graph:

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

The data set Cycles now contains all of the cycles in the input graph; it is shown in Figure 1.71.

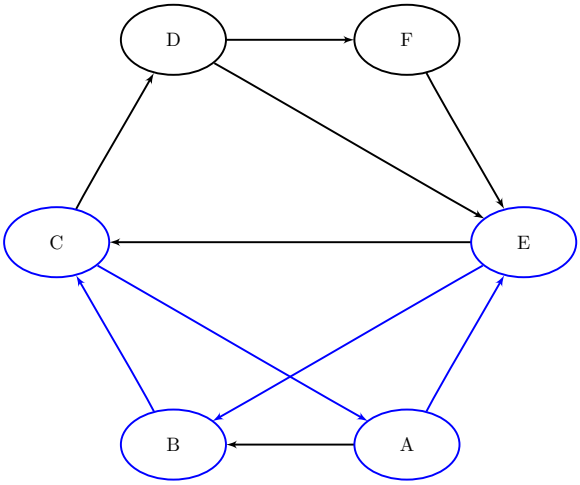
Figure 1.71 All Cycles in a Simple Directed Graph

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 cycles are shown graphically in Figure 1.72 through Figure 1.74.

Figure 1.72 Cycles

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



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

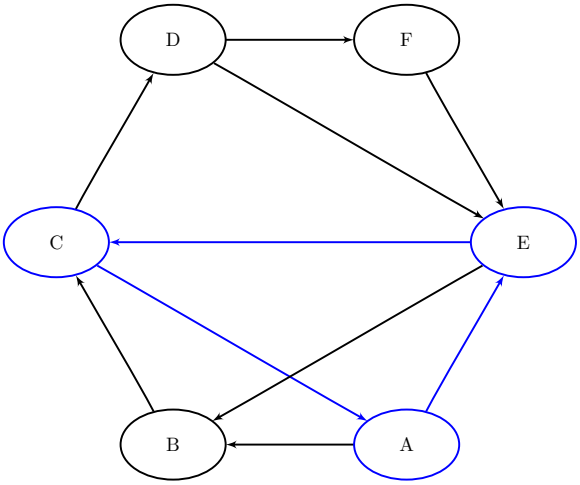
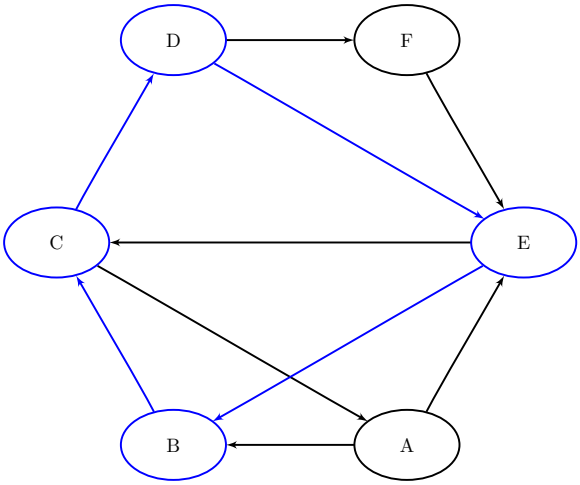


Figure 1.73 Cycles

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



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

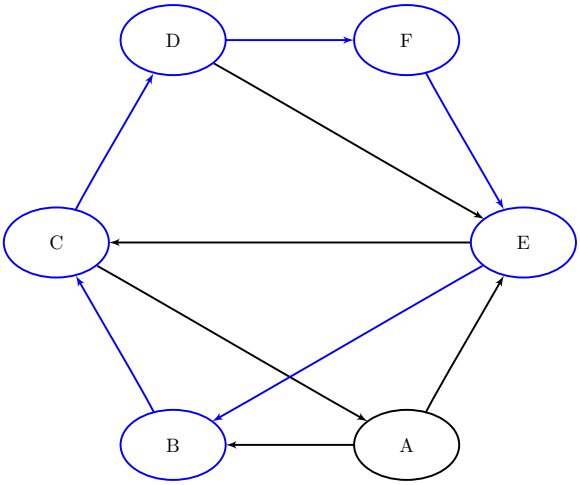
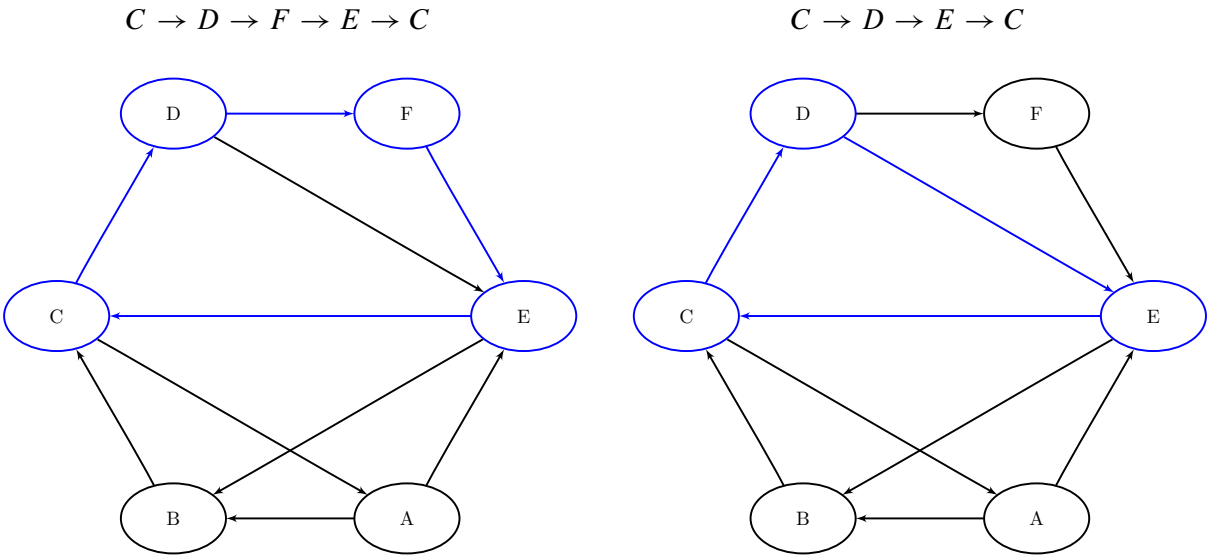


Figure 1.74 Cycles



Eigenvector Problem

For a given square matrix A, the *eigenvectors* of the matrix are those nonzero vectors that remain proportional to the original vector after being multiplied by A. That is, upon multiplication, an eigenvector changes magnitude, but not direction. The corresponding amount that the vector changes in magnitude is called the *eigenvalue*. Mathematically, a nonzero vector v and scalar λ is an eigenvector/value pair if and only if it satisfies the equation $Av = \lambda v$.

In PROC OPTGRAPH, you can calculate eigenvectors of a given matrix by invoking the EIGENVECTOR statement. The options for this statement are described in the section "EIGENVECTOR Statement" on page 33.

The EIGENVECTOR statement reports status information in a macro variable called _OPTGRAPH_EIGEN_. See the section "Macro Variable _OPTGRAPH_EIGEN_" on page 176 for more information about this macro variable.

Although the matrix is typically defined in the input data set specified in the DATA_MATRIX= option, it can also be presented in the form of a graph by using the DATA_LINKS= option. In this case, the graph is converted to the corresponding adjacency matrix. This conversion enables you to calculate the eigenvectors of very large matrices, since the data format for a graph is very sparse. Because of memory limitations, the matrix format is useful only for relatively small matrices. Because the matrix must be symmetric, the graph input format works only for undirected graphs.

The algorithm that PROC OPTGRAPH uses to solve the eigensystem is a variant of the Jacobi-Davidson algorithm (Sleijpen and van der Vorst 2000). This algorithm uses sparse computations for efficiency and is designed to find a small number of extremal eigenvectors. If you want to find all the eigenvectors and your matrix is relatively small, the best alternative is to use the dense solver in the IML procedure. (See the *SAS/IML User's Guide*.)

Eigenvector Problem for a Small Matrix with Dense Input

This section shows the calculation of the principal eigenvectors of a small matrix with the following dense input:

```
data MatrixSetIn;
   input col1-col5;
   datalines;
1 0 2 6 1
0 2 3 0 1
2 3 1 0 2
6 0 0 0 0
1 1 2 0 0
;
```

The following statements calculate the two algebraically largest eigenvalues for the matrix defined in the data set MatrixSetIn:

```
proc optgraph
   data_matrix    = MatrixSetIn;
   eigenvector
      eigenvalues = LA
      nEigen      = 2
      out         = EigenMatrixOut;
run;
```

For a matrix with n columns, and NEIGEN=m requested eigenpairs, the algebraically largest eigenvalue is given in the last observation $(n + 1)$ of the column eigen_1. The corresponding eigenvector is given in the same column in observations 1 through n. The second largest is given in column eigen_2, and so on, up to column eigen_m.

In this case, the resulting data set EigenMatrixOut (shown in Figure 1.75) gives the two largest eigenvector and eigenvalue pairs in columns eigen_1 and eigen_2. The first five observations give the values of the eigenvectors (one for each column of the matrix), and the sixth observation gives the corresponding eigenvalue.

Figure 1.75 Eigenvector Problem for a Small Matrix with Dense Input

obs	eigen_1	eigen_2
1	−0.65778	−0.32280
2	−0.26459	0.64125
3	−0.40078	0.49082
4	−0.53174	−0.40988
5	−0.23227	0.27513
.	7.42209	4.72527

Eigenvector Problem for a Small Matrix with Sparse Input

This section shows the use of a sparse input format for the eigenvector problem. The following statements define the same matrix that is used in the section "Eigenvector Problem for a Small Matrix with Dense Input" on page 116, but they represent it sparsely in the form of graph links:

```
data LinkSetIn;
   input from to weight;
   datalines;
0 0 1
0 2 2
0 3 6
0 4 1
1 1 2
1 2 3
1 4 1
2 2 1
2 4 2
;
```

Notice that there are self links $i \rightarrow i$. These correspond to the diagonal entries in the matrix that is defined in the data set MatrixSetIn. By default, PROC OPTGRAPH ignores self links. Therefore, in the sparse format, you must use the INCLUDE_SELFLINK option to match the dense matrix from the section "Eigenvector Problem for a Small Matrix with Dense Input" on page 116. Now you can calculate the same eigenvectors using sparse input as follows:

```
proc optgraph
   include_selflink
   data_links     = LinkSetIn;
   eigenvector
      eigenvalues = LA
      nEigen      = 2
      out         = EigenLinksOut;
run;
```

The output is shown in Figure 1.76.

Figure 1.76 Eigenvector Problem for a Small Matrix with Sparse Input

node	eigen_1	eigen_2
0	-0.65778	0.32280
2	-0.40078	-0.49082
3	-0.53174	0.40988
4	-0.23227	-0.27513
1	-0.26459	-0.64125
.	7.42209	4.72527

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 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 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 OPTGRAPH, the linear assignment problem solver can be invoked by using the LINEAR_ASSIGNMENT statement. The options for this statement are described in the section "LINEAR_ASSIGNMENT Statement" on page 34. The algorithm used in PROC OPTGRAPH for solving 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 57) or graph data input (see the section "Graph Input Data" on page 48) as long as the graph is bipartite.

The resulting assignment (or matching) is given 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 _OPTGRAPH_LAP_. See the section "Macro Variable _OPTGRAPH_LAP_" on page 176 for more information about this macro variable.

For a detailed example, see "Example 1.10: Linear Assignment Problem for Minimizing Swim Times" on page 214.

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 OPTGRAPH, the minimum cut algorithm can be invoked by using the experimental MINCUT statement. The options for this statement are described in the section "MINCUT Statement (Experimental)" on page 36. 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 OPTGRAPH 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.

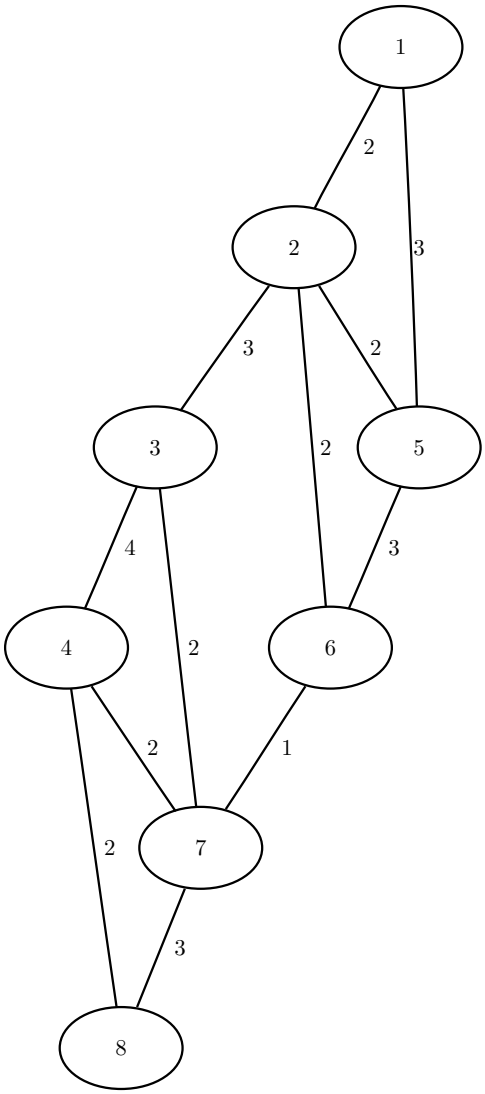
The minimum cut algorithm reports status information in a macro variable called _OPTGRAPH_MINCUT_. See the section "Macro Variable _OPTGRAPH_MINCUT_" on page 177 for more information about this macro variable.

PROC OPTGRAPH 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 1.77.

Figure 1.77 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 optgraph
   loglevel     = moderate
   out_nodes    = NodeSetOut
   data_links   = LinkSetIn;
   mincut
     out        = MinCut
     maxnumcuts = 3;
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_MINCUT_;
```

The progress of the procedure is shown in Figure 1.78.

Figure 1.78 PROC OPTGRAPH Log for Minimum Cut

```
NOTE: ----------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------
NOTE: Running OPTGRAPH version 12.3.
NOTE: ----------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------
NOTE: The OPTGRAPH 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 0.01 (cpu: 0.02) seconds.
NOTE: Building the input graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The input graph storage is using 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 MINCUT statement.
NOTE: The MINCUT algorithm is experimental in this release.
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 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 0.00 (cpu: 0.00) 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=OK
STATUS=OK  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 1.79.

Figure 1.79 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 1.80 along with each cut separately.

Figure 1.80 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

Figure 1.80 *continued*

```
---------------------------------- 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. Given weights on the links, a *minimum spanning tree* (MST) is a spanning tree whose weight is less than or equal to the weight 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 OPTGRAPH, the minimum spanning tree algorithm can be invoked by using the MINSPANTREE statement. The options for this statement are described in the section "MINSPANTREE Statement" on page 37. This algorithm can be used only on undirected graphs.

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

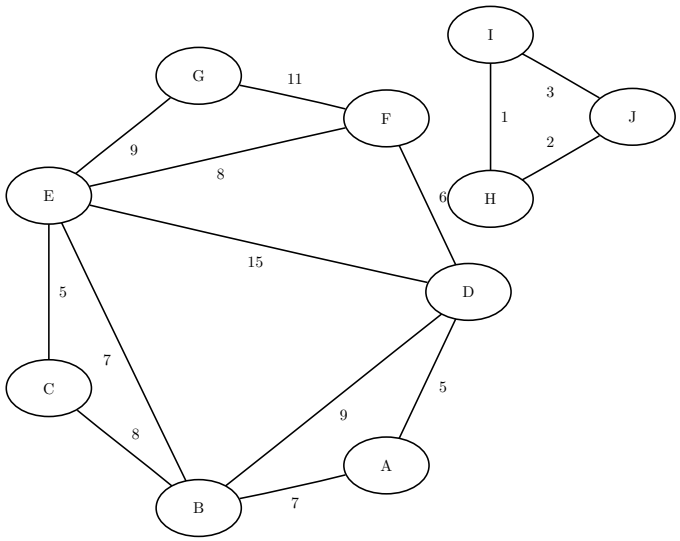
The minimum spanning tree algorithm reports status information in a macro variable called _OPTGRAPH_MST_. See the section "Macro Variable _OPTGRAPH_MST_" on page 177 for more information about this macro variable.

PROC OPTGRAPH 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 1.81.

Figure 1.81 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 optgraph
   data_links = LinkSetIn;
   minspantree
      out     = MinSpanForest;
run;
```

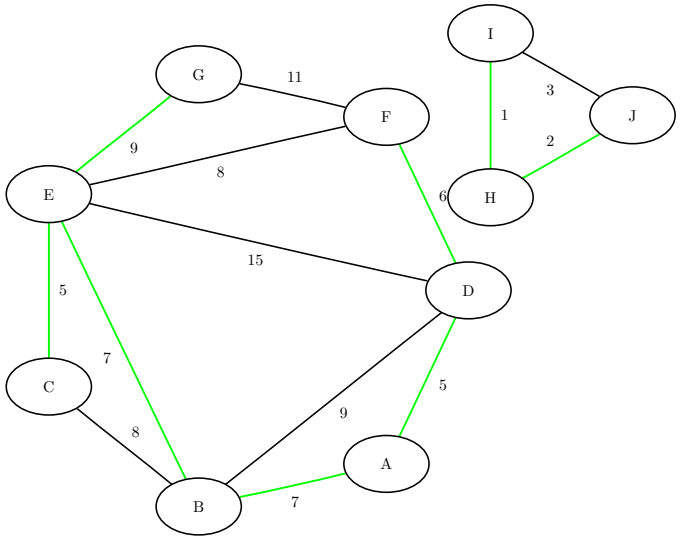
The data set MinSpanForest now contains the links that belong to a minimum spanning forest, which is shown in Figure 1.82.

Figure 1.82 Minimum Spanning Forest

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

The minimal cost links are shown in green in Figure 1.83.

Figure 1.83 Minimum Spanning Forest



For a more detailed example, see Example 1.12.

Minimum-Cost Network Flow

The *minimum-cost network flow problem* (MCF) is a fundamental problem in network analysis that involves sending flow over a network at minimal cost. Let $G = (N, A)$ be a directed graph. For each link $(i, j) \in A$, associate a cost per unit of flow, designated by c_{ij}. The demand (or supply) at each node $i \in N$ is designated as b_i, where $b_i \geq 0$ denotes a supply node and $b_i < 0$ denotes a demand node. These values must be within $[b_i^l, b_i^u]$. Define decision variables x_{ij} that denote the amount of flow sent between node i and node j. The

amount of flow that can be sent across each link is bounded to be within $[l_{ij}, u_{ij}]$. The problem can be modeled as a linear programming problem as follows:

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

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 OPTGRAPH, the minimum-cost network flow solver can be invoked by using the MINCOSTFLOW statement. The options for this statement are described in the section "MINCOSTFLOW Statement" on page 35.

The minimum-cost network flow solver reports status information in a macro variable called _OPTGRAPH_MCF_. See the section "Macro Variable _OPTGRAPH_MCF_" on page 176 for more information about this macro variable.

The algorithm used in PROC OPTGRAPH for solving MCF is a variant of the primal network simplex algorithm (Ahuja, Magnanti, and Orlin 1993). Sometimes the directed graph G is disconnected. In this case, the problem is first decomposed into its weakly connected components, and then each minimum-cost flow problem is solved separately.

The input for the network is the standard graph input described in the section "Graph Input Data" on page 48. The links data set, which is specified in the DATA_LINKS= option in the PROC OPTGRAPH statement, contains the following columns:

- weight defines the link cost c_{ij}
- lower defines the link lower bound l_{ij} (the default is 0)
- upper 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 OPTGRAPH statement, can contain the following columns:

- weight defines the node supply lower bound b_i^l (the default is 0)
- weight2 defines the node supply upper bound b_i^u (the default is ∞)

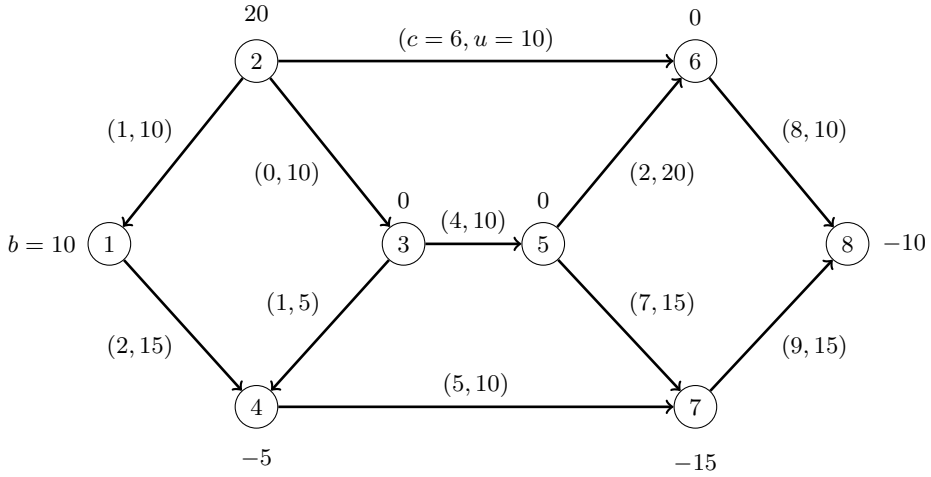
To define a pure network where the node supply must be met exactly, use the weight variable only. You do not need to specify all the node supply bounds. For any missing node, the solver uses its default values.

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 OPTGRAPH statement.

Minimum Cost Network Flow for a Simple Directed Graph

The following example demonstrates how to use the network simplex solver to find a minimum-cost flow in a directed graph. Consider the directed graph in Figure 1.84, which appears in Ahuja, Magnanti, and Orlin (1993).

Figure 1.84 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 OPTGRAPH to find a minimum-cost flow:

```
proc optgraph
   loglevel        = moderate
   graph_direction = directed
   data_links      = LinkSetIn
   data_nodes      = NodeSetIn
   out_links       = LinkSetOut;
   mincostflow
      logfreq      = 1;
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_MCF_;
```

The progress of the procedure is shown in Figure 1.85.

Figure 1.85 PROC OPTGRAPH Log for Minimum-Cost Network Flow

```
NOTE: ------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------
NOTE: Running OPTGRAPH version 12.3.
NOTE: ------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------
NOTE: The OPTGRAPH procedure is executing in single-machine mode.
NOTE: ------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: Reading the nodes data set.
NOTE: There were 5 observations read from the data set WORK.NODESETIN.
NOTE: There were 11 observations read from the data set WORK.LINKSETIN.
NOTE: Data input used 0.01 (cpu: 0.02) seconds.
NOTE: Building the input graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The input graph storage is using 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 MINCOSTFLOW statement.
NOTE: The network has 1 connected component.
                      Primal         Primal           Dual
      Iteration      Objective  Infeasibility  Infeasibility     Time
              1              0     20.0000000     89.0000000     0.00
              2              0     20.0000000     89.0000000     0.00
              3      5.0000000     15.0000000     84.0000000     0.00
              4      5.0000000     15.0000000     83.0000000     0.00
              5     75.0000000     15.0000000     83.0000000     0.00
              6     75.0000000     15.0000000     79.0000000     0.00
              7    130.0000000     10.0000000     76.0000000     0.00
              8    270.0000000              0              0     0.00
NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: The minimum cost network flow is 270.
NOTE: Processing the minimum cost network flow 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
```

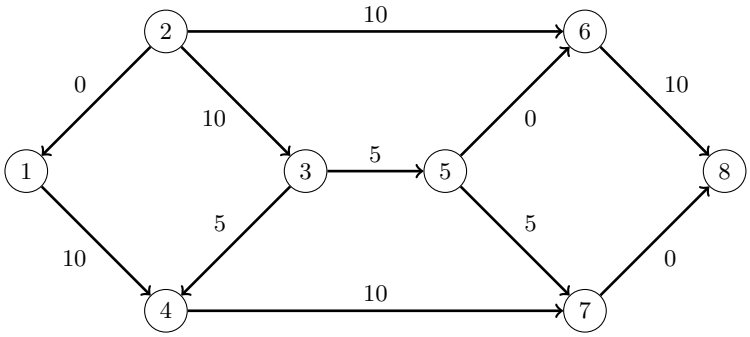
The optimal solution is displayed in Figure 1.86.

Figure 1.86 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 1.87.

Figure 1.87 Minimum-Cost Network Flow Problem: Optimal Solution



Reach (Ego) Network

The *reach network* of a graph $G = (N, A)$ is a graph $G_L^R = (N_L^R, A_L^R)$ that is defined as the induced subgraph over the set of nodes N_L^R that are reachable in L steps (or hops) from a set S of nodes, called the *source nodes*. Reach networks are often referred to as *ego networks* in the context of social networks, since they focus around the neighbors of one (or more) particular individuals.

In PROC OPTGRAPH, reach networks can be calculated by using the REACH statement. The options for this statement are described in the section "REACH Statement" on page 38.

The REACH statement reports status information in a macro variable called _OPTGRAPH_REACH_. See the section "Macro Variable _OPTGRAPH_REACH_" on page 178 for more information about this macro variable.

In most cases, the set of source nodes from which to calculate reach are defined in a node subset data set, as described in the section "Node Subset Input Data" on page 55. The node subset data set can be used to define several sets of sources nodes. Each source node set is used to calculate the reach networks. The reach network identifier is given in the node subset data set's reach column. When you use the EACH_SOURCE option, every node in the original graph's node set N is used to find a reach network from each node separately.

Output Data Sets

Depending on the options selected, the reach network algorithm produces output data sets as described in the following sections.

OUT_NODES= Data Set

This data set describes the nodes in each reach network that are found from each set of source nodes. The data set contains the following columns:

- node: node label for each node in each reach network
- reach: reach network identifier (which defines the set of source nodes that was used)

OUT_LINKS= Data Set

This data set describes the links in each reach network that are found from each set of source nodes. Output of the reach network links can sometimes be more costly computationally, relative to calculating only the nodes or counts in the reach networks. This option does not work when you use the BY_CLUSTER option. The data set contains the following columns:

- from: the *from* node label for each link in each reach network
- to: the *to* node label for each link in each reach network
- reach: reach network identifier (which defines the set of source nodes that was used)

OUT_COUNTS= Data Set

This data set describes the number of nodes in each reach network for each set of sources nodes. The data set contains the following columns:

- node: node label for each node in the source node sets
- reach: reach network identifier (which defines the set of source nodes that was used)
- count: the number of nodes reachable using outgoing links from the source nodes
- count_not: the number of nodes not reachable using outgoing links from the source nodes

If the graph is directed and you use the DIGRAPH option, then the OUT_COUNTS= data set contains the following additional columns:

- count_in: the number of nodes reachable using incoming links from the source node
- count_out: the number of nodes reachable using outgoing links from the source node (equivalent to count)
- count_in_or_out: the number of nodes reachable using incoming or outgoing links (but not both) from the source node
- count_in_and_out: the number of nodes reachable using both incoming and outgoing links from the source node

If node weights are present, the OUT_COUNTS= data set contains the following additional columns:

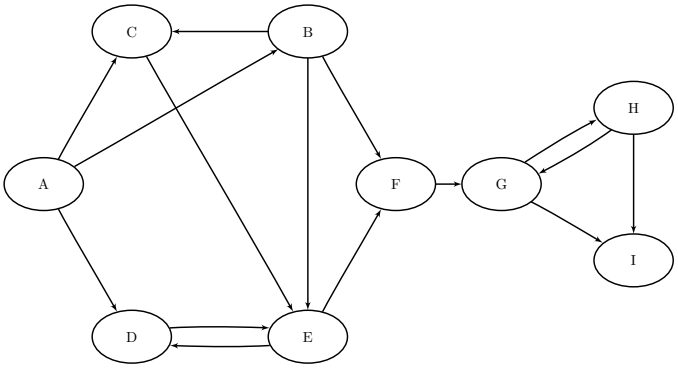
- count_wt: the sum of the weights of the nodes reachable using outgoing links from the source node
- count_not_wt: the sum of the weights of the nodes not reachable from the source node
- count_in_wt: the sum of the weights of the nodes reachable using incoming links from the source node
- count_out_wt: the sum of the weights of the nodes reachable using outgoing links from the source node
- count_in_or_out_wt: the sum of the weights of the nodes reachable using incoming or outgoing links (but not both) from the source node
- count_in_and_out_wt: the sum of the weights of the nodes reachable using both incoming and outgoing links from the source node

When you want to calculate hop limits of 1 and 2 on the same graph, you can use the OUT_COUNTS1= and OUT_COUNTS2= options to do this in one call. This option works only when the EACH_SOURCE and BY_CLUSTER options are specified.

Reach Network of a Simple Directed Graph

This section illustrates the use of the reach networks algorithm on the simple directed graph G shown in Figure 1.88.

Figure 1.88 Simple Directed Graph G



The directed graph G can be represented using the links data set LinkSetIn as follows:

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

Consider two sets of source nodes, $S_1 = \{A, G\}$ and $S_2 = \{B\}$. These can be defined separately in two node subset data sets as follows:

```
data NodeSubSetIn1;
   input node $ reach;
   datalines;
A 1
G 1
;

data NodeSubSetIn2;
   input node $ reach;
   datalines;
B 1
;
```

For the first set of source nodes, you can use the following statements to calculate the reach network with a hop limit of 1:

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   data_nodes_sub  = NodeSubSetIn1;
   reach
      out_nodes    = ReachNodes1
      out_links    = ReachLinks1
      out_counts   = ReachCounts1
      maxreach     = 1;
run;
```

The data sets ReachNodes1, ReachLinks1, and ReachCounts1 now contain the nodes, links, and counts of the reach network, respectively, that come from S_1.

Figure 1.89 Reach Network for $S_1 = \{A, G\}$ with Hop Limit of 1

ReachNodes1

reach	node
1	A
1	B
1	C
1	D
1	G
1	H
1	I

Figure 1.89 *continued*

ReachLinks1

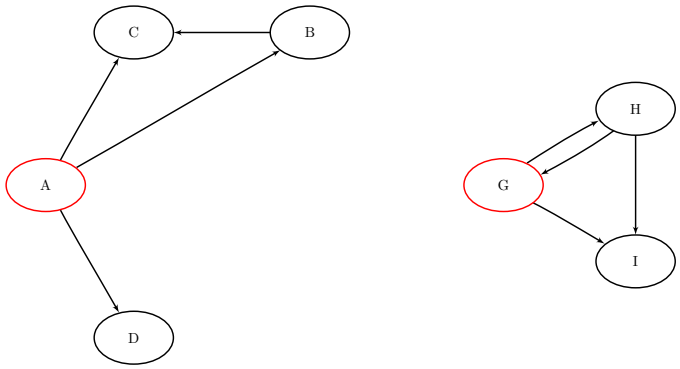
reach	from	to
1	A	B
1	A	C
1	A	D
1	B	C
1	G	H
1	G	I
1	H	G
1	H	I

ReachCounts1

reach	node	count	count_not
1	A	7	2
1	G	7	2

The results are displayed graphically in Figure 1.90.

Figure 1.90 Reach Network for $S_1 = \{A, G\}$ with Hop Limit of 1



For the second set of source nodes, you can use the following statements to calculate the reach network with a hop limit of 2:

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   data_nodes_sub  = NodeSubSetIn2;
   reach
      out_nodes    = ReachNodes2
      out_links    = ReachLinks2
      out_counts   = ReachCounts2
      maxreach     = 2;
run;
```

The data sets ReachNodes2, ReachLinks2, and ReachCounts2 now contain the nodes, links, and counts of the reach network, respectively, that come from S_2.

Figure 1.91 Reach Network for $S_2 = \{B\}$ with Hop Limit of 2

ReachNodes2

reach	node
1	B
1	C
1	D
1	E
1	F
1	G

ReachLinks2

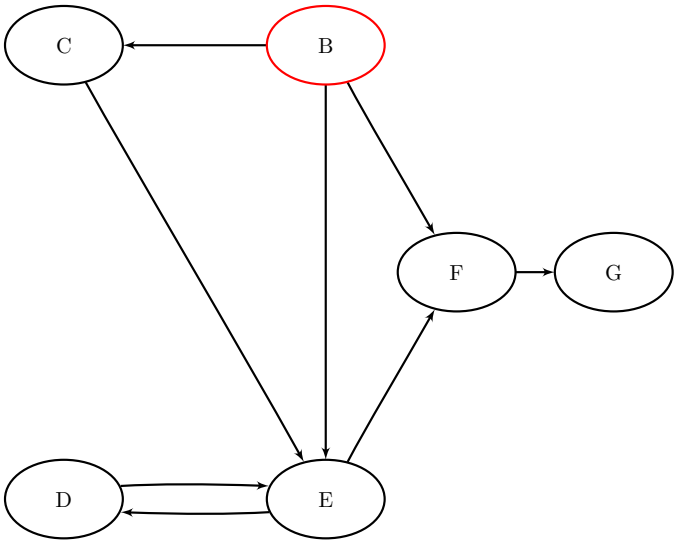
reach	from	to
1	B	C
1	B	E
1	B	F
1	C	E
1	D	E
1	E	D
1	E	F
1	F	G

ReachCounts2

reach	node	count	count_not
1	B	6	3

The results are displayed graphically in Figure 1.92.

Figure 1.92 Reach Network for $S_1 = \{B\}$ with Hop Limit of 2



Processing Multiple Reach Networks in One Pass

You can process a set of reach networks from one graph in one pass using one node subset data set. The MAXREACH= option applies to all of the reach networks requested. If the node subset data set column reach is set to 0 or missing (.), then the node is not processed. If the column reach is set to a value greater than 0, then the node is processed with other nodes by using the same marker.

Consider again the graph shown in Figure 1.88, now with source node sets $S_1 = \{C\}$ and $S_2 = \{A, H\}$. These source node sets can be defined together as follows:

```
data NodeSubSetIn;
   input node $ reach;
   datalines;
A 2
C 1
H 2
;
```

You can use the following statements to process the two one-hop-limit reach networks in one pass:

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   data_nodes_sub  = NodeSubSetIn;
   reach
      out_nodes    = ReachNodes
      out_links    = ReachLinks
      out_counts   = ReachCounts
      maxreach     = 1;
run;
```

The data sets ReachNodes, ReachLinks, and ReachCounts now contain the nodes, links, and counts of the reach networks, respectively, that come from S_1 and S_2.

Figure 1.93 Reach Networks for $S_1 = \{C\}$ and $S_2 = \{A, H\}$ with Hop Limit of 1

ReachNodes

reach	node
1	C
1	E
2	A
2	B
2	C
2	D
2	G
2	H
2	I

ReachLinks

reach	from	to
1	C	E
2	A	B
2	A	C
2	A	D
2	B	C
2	G	H
2	G	I
2	H	G
2	H	I

ReachCounts

reach	node	count	count_not
1	C	2	7
2	A	7	2
2	H	7	2

Processing Reach Networks by Cluster

Similar to the usage for centrality described in the section "Processing by Cluster" on page 80, you can use the BY_CLUSTER option in the REACH statement to process a number of induced subgraphs of a graph with only one call to PROC OPTGRAPH. In this section, you want to work on the subgraphs that are induced by node subsets $N_0 = \{A, C, D, E\}$ and $N_1 = \{B, F, G, H, I\}$ for the directed graph shown in Figure 1.88. The induced subgraphs are shown graphically in Figure 1.94 and Figure 1.95.

Figure 1.94 Induced Subgraph for $N^0 = \{A, C, D, E\}$

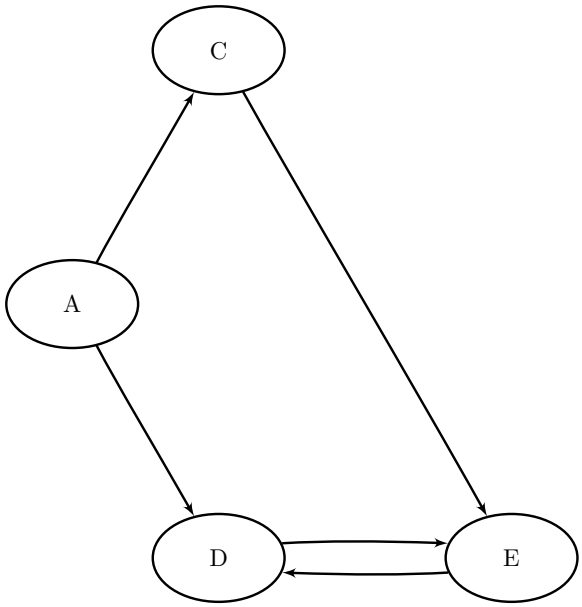
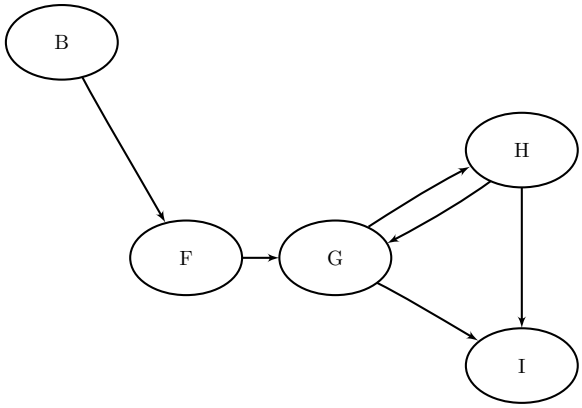


Figure 1.95 Induced Subgraph for $N^1 = \{B, F, G, H, I\}$



Define the subgraphs in the nodes data set by using the cluster variable as follows:

```
data NodeSetIn;
   input node $ cluster @@;
   datalines;
A 0  B 1  C 0  D 0  E 0
F 1  G 1  H 1  I 1
;
```

In the node subset data set, define the source nodes set $S = \{B, C\}$ by using the reach variable as follows:

```
data NodeSubSetIn;
   input node $ reach;
   datalines;
B 1
C 1
;
```

To process the two-hop-limit reach network for each induced subgraph, you can use the following statements:

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   data_nodes      = NodeSetIn
   data_nodes_sub  = NodeSubSetIn;
   performance
      nthreads     = 2;
   reach
      by_cluster
      out_nodes    = ReachNodes
      out_counts   = ReachCounts
      maxreach     = 2;
run;
```

Notice in this example that you can process each subgraph in parallel by using the NTHREADS= option in the PERFORMANCE statement.

The data sets ReachNodes and ReachCounts now contain the nodes and counts of the reach networks, respectively, that come from S for each induced subgraph.

Figure 1.96 Reach Networks for $S = \{B, C\}$ with Hop Limit of 2 for Induced Subgraphs

ReachNodes

reach	node	cluster
1	B	1
1	C	0
1	D	0
1	E	0
1	F	1
1	G	1

ReachCounts

reach	node	cluster	count	count_not
1	C	0	3	1
1	B	1	3	2

Notice that since you are operating on the induced subgraphs (not the original graph), node B cannot reach nodes C and E because they are not in its induced subgraph.

Processing Multiple Reach Networks in One Pass by Cluster

You can also process several reach networks in one pass while looking over decomposed subgraphs. Consider the same original graph and subgraphs from the section "Processing Reach Networks by Cluster" on page 137. Now, suppose you want the one-hop-limit reach network where each original node is its own source node subset. Define nine source sets by using the node subset data set as follows:

```
data NodeSubSetIn;
   input node $ reach @@;
   datalines;
A 1  B 2  C 3  D 4  E 5
F 6  G 7  H 8  I 9
;
```

Then, to calculate the reach networks (including the directed graph counts) for each source node set on the induced subgraphs, use the following statements:

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   data_nodes      = NodeSetIn
   data_nodes_sub  = NodeSubSetIn;
   performance
      nthreads     = 2;
   reach
      by_cluster
      digraph
      out_nodes    = ReachNodes
      out_counts   = ReachCounts
      maxreach     = 1;
run;
```

Notice that you can do the same thing using the EACH_SOURCE option. In this case, you do not need the subset data set.

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   data_nodes      = NodeSetIn;
   performance
      nthreads     = 2;
   reach
      each_source
      by_cluster
      digraph
      out_nodes    = ReachNodes
      out_counts   = ReachCounts
      maxreach     = 1;
run;
```

The resulting data sets ReachNodes and ReachCounts are displayed in Figure 1.97.

Figure 1.97 Reach Networks for Each Source for Induced Subgraphs with a Node Hop Limit of 1

ReachNodes

reach	node	cluster
1	A	0
1	C	0
1	D	0
2	B	1
2	F	1
3	C	0
3	E	0
4	D	0
4	E	0
5	D	0
5	E	0
6	F	1
6	G	1
7	G	1
7	H	1
7	I	1
8	G	1
8	H	1
8	I	1
9	I	1

ReachCounts

reach	node	cluster	count	count_ not	count_in	count_ out	count_ in_or_ out	count_ in_and_ out
1	A	0	3	1	0	3	3	0
2	B	1	2	3	0	2	2	0
3	C	0	2	2	1	2	3	0
4	D	0	2	2	2	2	2	1
5	E	0	2	2	2	2	2	1
6	F	1	2	3	1	2	3	0
7	G	1	3	2	2	3	3	1
8	H	1	3	2	1	3	2	1
9	I	1	1	4	2	1	3	0

Processing Each Source Reach Network for Hop Limits of Both 1 and 2 in One Pass by Cluster

In this section, suppose you want to calculate the one-hop- and two-hop-limit reach counts on the same graph for each source node on a set of induced subgraphs. You can do this in one pass by using the OUT_COUNTS1= and OUT_COUNTS2= options, as follows:

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   data_nodes      = NodeSetIn;
   performance
      nthreads     = 2;
   reach
      each_source
      by_cluster
      out_counts1  = ReachCounts1
      out_counts2  = ReachCounts2;
run;
```

The resulting data sets ReachCounts1 and ReachCounts1 are displayed in Figure 1.98.

Figure 1.98 Reach Counts for Each Source Node for Induced Subgraphs with a Hop Limit of 1 and 2

ReachCounts1

reach	node	cluster	count	count_not
1	A	0	3	1
3	C	0	2	2
4	D	0	2	2
5	E	0	2	2
2	B	1	2	3
6	F	1	2	3
7	G	1	3	2
8	H	1	3	2
9	I	1	1	4

ReachCounts2

reach	node	cluster	count	count_not
1	A	0	4	0
3	C	0	3	1
4	D	0	2	2
5	E	0	2	2
2	B	1	3	2
6	F	1	4	1
7	G	1	3	2
8	H	1	3	2
9	I	1	1	4

For a more detailed example, see Example 1.14.

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 with the lowest total link weight. The starting node is referred to as the *source node*, and the ending node is referred to as the *sink node*.

In PROC OPTGRAPH, shortest paths can be calculated by invoking the SHORTPATH statement. The options for this statement are described in the section "SHORTPATH Statement" on page 40.

The shortest path algorithm reports status information in a macro variable called _OPTGRAPH_SHORTPATH_. See the section "Macro Variable _OPTGRAPH_SHORTPATH_" on page 178 for more information about this macro variable.

By default, PROC OPTGRAPH 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. 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 55. The following sections show examples of these options.

The algorithm used in PROC OPTGRAPH for finding shortest paths is a variant of Dijkstra's algorithm (Ahuja, Magnanti, and Orlin 1993). For unweighted graphs, PROC OPTGRAPH uses a variant of breadth-first search. Dijkstra's algorithm on weighted graphs runs in time $O(|N|\log|N| + |A|)$ for each source node. Breadth-first search runs in time $O(|N| + |A|)$ for each source node.

For weighted graphs, the algorithm uses the weight variable that is defined in the links data set to evaluate a path's total weight (or 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 simply calculated 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 is specified in the OUT_PATHS= option contains the links of a shortest path for each source-sink pair combination. The output data set that is specified in the OUT_WEIGHTS= option contains the total weight for the shortest path for each source-sink pair combination.

OUT_PATHS= Data Set

This data set contains the links present in the shortest path for each of the source-sink pairs. For large graphs and a large requested number of source-sink pairs, this output data set can be extremely large. In this case, the generation of the output can sometimes take longer than the computation of the shortest paths. For example, using the U.S. road network data for the state of New York, the data contain a directed graph with 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

This data set contains the total weight (and total auxiliary weight) for the shortest path for each of the source-sink pair.

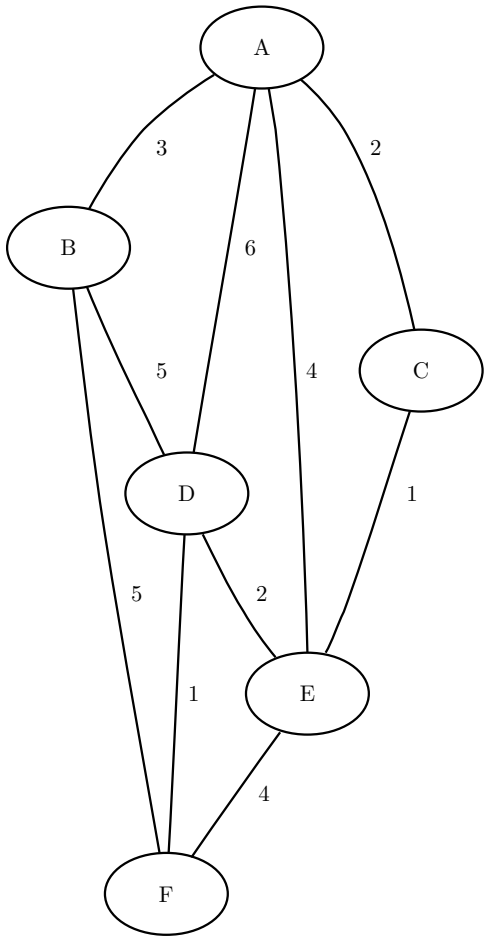
The data set contains the following columns:

- 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 1.99.

Figure 1.99 A Simple Undirected Graph G



The undirected graph G can be represented by the links data set LinkSetIn as follows:

```
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 optgraph
   data_links     = LinkSetIn;
   shortpath
      out_weights = ShortPathW
      out_paths   = ShortPathP;
run;
```

The data set ShortPathP contains the shortest paths and is shown in Figure 1.100.

Figure 1.100 All-Pairs Shortest Paths

ShortPathP

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

The data set ShortPathW contains the path weight for the shortest paths of each source-sink pair and is shown in Figure 1.101.

Figure 1.101 All-Pairs Shortest Paths Summary

ShortPathW

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
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 with the longest shortest path, you can use the PATHS= option. This option affects only the output processing; it does not affect the computation. All of 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 optgraph
   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 1.102.

Figure 1.102 Longest Shortest Path

ShortPathLong

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 for calculating shortest paths between 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 OPTGRAPH then calculates all the source-sink pairs in the cross product of these two sets.

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

```
data NodeSetInSub;
   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 optgraph
   data_nodes_sub = NodeSetInSub
   data_links     = LinkSetIn;
   shortpath
      out_paths   = ShortPath;
run;
```

The data set ShortPath contains the shortest paths and is shown in Figure 1.103.

Figure 1.103 Shortest Paths for a Subset of Source-Sink Pairs

ShortPath

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 for calculating 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 source (or sink). By specifying only one of the variables, you indicate that you want PROC OPTGRAPH to calculate all pairs from a subset of source nodes (or all pairs to a subset of sink nodes).

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

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

You can use the same PROC OPTGRAPH call as is used in the section "Shortest Paths for a Subset of Source-Sink Pairs" on page 149 to calculate all the shortest paths from nodes B and E. The data set ShortPath contains the shortest paths and is shown in Figure 1.104.

Figure 1.104 Shortest Paths for a Subset of Source Pairs

ShortPath

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 NodeSetInSub;
   input node $ sink;
   datalines;
B 1
E 1
;
```

You can use the same PROC OPTGRAPH 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 1.105.

Figure 1.105 Shortest Paths for a Subset of Sink Pairs

ShortPath

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 for calculating 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 optgraph
   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 1.106.

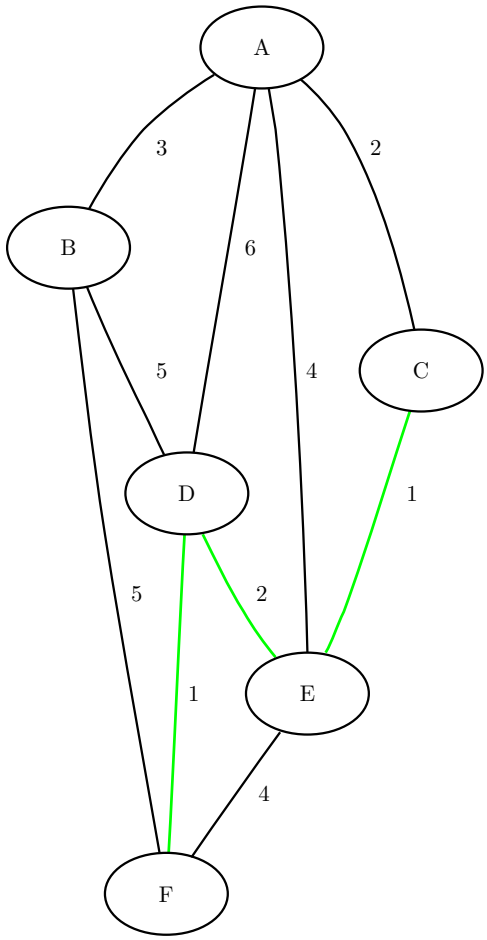
Figure 1.106 Shortest Paths for One Source-Sink Pair

ShortPath

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 1.107.

Figure 1.107 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 for calculating shortest paths between all source-sink pairs.

Consider a links data set where 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 optgraph
   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 1.108. Since the variable count in LinkSetIn is 1 for all links, the value in the output data set variable path_weights2 gives the number of links in each shortest path.

Figure 1.108 Shortest Paths Including Auxiliary Weights in Calculation

ShortPathW

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
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 4 shows an example of using the shortest path algorithm for minimizing travel to and from work based on traffic conditions.

Summary

In PROC OPTGRAPH, various summary statistics for a graph and its nodes can be calculated by invoking the SUMMARY statement. The options for this statement are described in the section "SUMMARY Statement" on page 41.

The SUMMARY statement reports status information in a macro variable called _OPTGRAPH_SUMMARY_. See the section "Macro Variable _OPTGRAPH_SUMMARY_" on page 178 for more information about this macro variable.

Output Data Sets

The summary statistics produced are broken into two categories: statistics on the entire graph and statistics on the nodes of the graph. The latter is appended to the output nodes data set that is specified in the OUT_NODES= option in the PROC OPTGRAPH statement. The former is contained in the data set that is specified in the OUT= option in the SUMMARY statement.

OUT= Data Set

By default, the summary output data set that is specified in the OUT= option in the SUMMARY statement contains the following columns:

- nodes: the number of nodes in the graph ($|N|$)
- links: the number of links in the graph ($|A|$)
- avg_links_per_node: the average number of links per node
- density: the number of links in the graph ($|A|$) divided by the total number of links in a complete graph ($|N|(|N|-1)$)

You can produce statistics about the connectedness of the graph by using the CONCOMP and BICONCOMP options. For more information about connected components and biconnected components, see the sections "Connected Components" on page 100 and "Biconnected Components and Articulation Points" on page 59, respectively. If you use the CONCOMP and BICONCOMP options, the following columns also appear in the summary output data set:

- concomp: the number of (weakly) connected components in the graph
- biconcomp: the number of biconnected components in the graph (undirected graphs only)
- artpoints: the number of articulation points in the graph (undirected graphs only)

You can produce statistics about the shortest paths in the graph by using the SHORTPATH= option. The *diameter* of a graph is the longest shortest path of all possible source-sink pairs in the graph. Calculating the diameter of a graph is an expensive computation, because it involves calculating shortest paths for all pairs. For undirected graphs, an approximate method is available based on Boitmanis et al. (2006).

The algorithm can be invoked by using the DIAMETER_APPROX= option. The exact method runs in $O(|N| \times (|N| \log |N| + |A|))$; the approximate method runs in $O(|A|\sqrt{|N|})$ with an additive error of $O(\sqrt{|N|})$. For more information about shortest paths, see the section "Shortest Path" on page 143. If you use the SHORTPATH= option, the following columns also appear in the summary output data set:

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

Depending on which other options you specify, some of these columns might not appear in the summary output data set.

OUT_NODES= Data Set

In addition, you can produce summary statistics about the nodes of the graph. By default, the following column is appended to the data set specified in the OUT_NODES= option in the PROC OPTGRAPH statement:

- sum_in_and_out_wt: sum of the link weights from and to the node

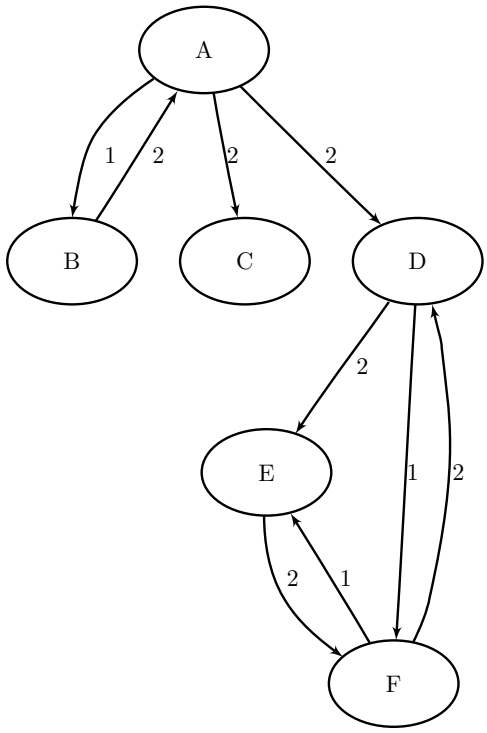
You can produce statistics about the shortest paths to and from nodes in the graph by using the SHORTPATH= option. The *eccentricity* of a node u is the longest shortest path of all possible shortest paths between u and any other node. If you use the SHORTPATH= option, the following columns also appear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

- eccentr_out_wt: the longest weighted shortest path from the node
- eccentr_out_unwt: the longest unweighted shortest path from the node
- eccentr_in_wt: the longest weighted shortest path to the node
- eccentr_in_unwt: the longest unweighted shortest path to the node

Summary Statistics of a Simple Directed Graph

This section illustrates the calculation of summary statistics on the simple directed graph G shown in Figure 1.109.

Figure 1.109 A Simple Directed Graph G



The directed graph G can be represented using the links data set LinkSetIn as follows:

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

The following statements calculate the default summary statistics and output the results in the data set Summary:

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn;
   summary
      out          = Summary;
run;
```

The data set Summary contains the default summary statistics of the input graph and is shown in Figure 1.110.

Figure 1.110 Graph Summary Statistics of a Simple Directed Graph

Summary

nodes	links	avg_links_per_node	density
6	9	1.5	0.3

The following statements calculate the default summary statistics and information about the connectedness of the graph, and they output the results in the data set Summary:

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn;
   summary
      concomp
      out          = Summary;
run;
```

The data set Summary contains the summary statistics of the input graph and is shown in Figure 1.110.

Figure 1.111 Graph Summary and Connectedness Statistics of a Simple Directed Graph

Summary

nodes	links	avg_links_per_node	density	concomp
6	9	1.5	0.3	3

The following statements calculate the default summary statistics and information about shortest paths of the graph, and they output the results in the data set Summary. In addition, node statistics are produced and output in the data set NodeSetOut.

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   out_nodes       = NodeSetOut;
   summary
      out          = Summary
      shortpath    = weight;
run;
```

The data set Summary contains the summary statistics of the input graph and is shown in Figure 1.112.

Figure 1.112 Graph Summary and Shortest Path Statistics of a Simple Directed Graph

Summary

nodes	links	avg_links_ per_node	density	diameter_ wt	avg_shortpath_ wt
6	9	1.5	0.3	6	2.8125

The data set NodeSetOut contains the summary statistics for each node of the input graph and is shown in Figure 1.113.

Figure 1.113 Node Summary and Shortest Path Statistics of a Simple Directed Graph

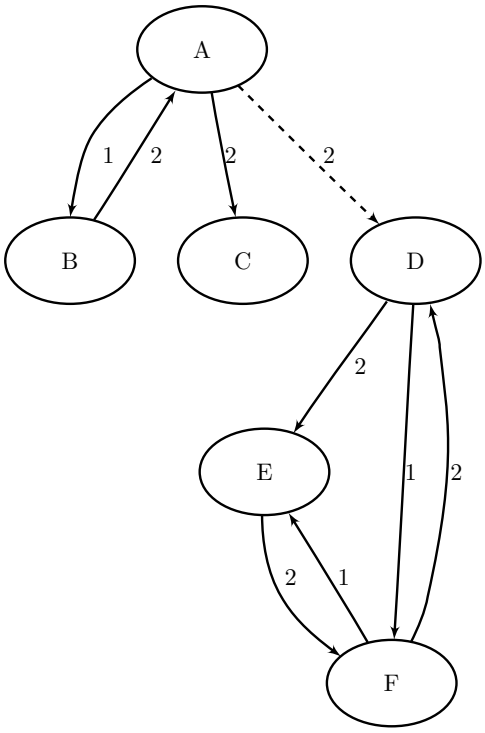
NodeSetOut

node	sum_in_ and_out_ wt	eccentr_ wt_out	eccentr_ wt_in
A	7	4	2
B	3	6	1
C	2	0	4
D	7	2	4
E	5	4	6
F	6	2	5

Summary Statistics of a Simple Directed Graph by Cluster

Similar to how you can use the BY_CLUSTER option in the CENTRALITY statement, as described in the section "Processing by Cluster" on page 80, you can process a number of induced subgraphs of a graph with only one call to PROC OPTGRAPH by using the BY_CLUSTER option in the SUMMARY statement. In this section, you want to work on the subgraphs induced by node subsets $N_0 = \{A, B, C\}$ and $N_1 = \{D, E, F\}$ for the directed graph shown in Figure 1.109. The induced subgraphs are shown graphically in Figure 1.114 (the dashed link is removed).

Figure 1.114 Induced Subgraphs of G



Define the subgraphs in the nodes data set by using the cluster variable as follows:

```
data NodeSetIn;
   input node $ cluster @@;
   datalines;
A 0  B 0  C 0
D 1  E 1  F 1
;
```

The following statements process the summary statistics for each induced subgraph:

```
proc optgraph
   graph_direction = directed
   data_links      = LinkSetIn
   data_nodes      = NodeSetIn
   out_nodes       = NodeSetOut;
   performance
     nthreads      = 2;
   summary
      by_cluster
      concomp
      shortpath    = weight
      out          = Summary;
run;
```

Notice in this example that you can process each subgraph in parallel by using the NTHREADS= option in the PERFORMANCE statement.

The data sets Summary and NodeSetOut now contain the summary statistics for each induced subgraph; they are shown in Figure 1.115.

Figure 1.115 Summary Statistics for Induced Subgraphs of G

Summary

cluster	nodes	links	avg_links_ per_node	density	concomp	diameter_ wt	avg_shortpath_ wt
0	3	3	1.00000	0.50000	2	4	2.25
1	3	5	1.66667	0.83333	1	4	2.00

NodeSetOut

node	cluster	sum_in_ and_out_ wt	eccentr_ wt_out	eccentr_ wt_in
A	0	5	2	2
B	0	3	4	1
C	0	2	0	4
D	1	5	2	4
E	1	5	4	2
F	1	6	2	2

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. This has many applications, including speeding up the processing of structured query languages, which are often used in databases.

In PROC OPTGRAPH, the transitive closure algorithm can be invoked by using the TRANSITIVE_CLOSURE statement. The options for this statement are described in the section "TRANSITIVE_CLOSURE Statement" on page 43.

The results for 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.

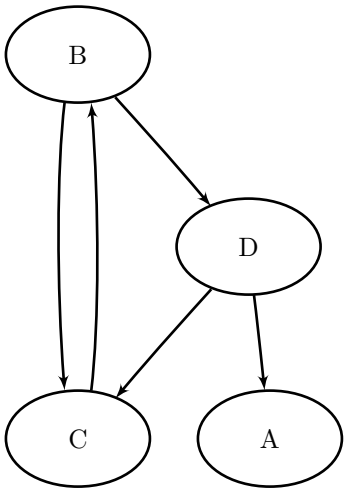
The transitive closure algorithm reports status information in a macro variable called _OPTGRAPH_TRANSCL_. See the section "Macro Variable _OPTGRAPH_TRANSCL_" on page 179 for more information about this macro variable.

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

Transitive Closure of a Simple Directed Graph

This example illustrates the use of the transitive closure algorithm on the simple directed graph G shown in Figure 1.116.

Figure 1.116 A Simple Directed Graph G



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

```
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 optgraph
   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 1.117.

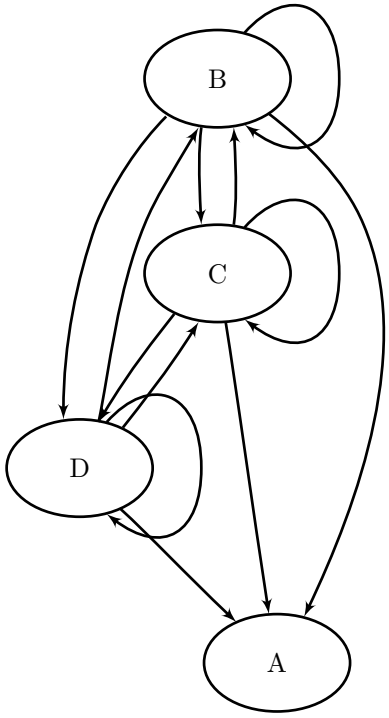
Figure 1.117 Transitive Closure of a Simple Directed Graph

Transitive Closure

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

The transitive closure of G is shown graphically in Figure 1.118.

Figure 1.118 Transitive Closure of G



For a more detailed example, see Example 1.13.

Traveling Salesman Problem

The *traveling salesman problem* (TSP) finds a minimum-cost tour in an undirected graph G with node set N and links set A. A *tour* is a connected subgraph for which each node has degree two. The goal is then to find a tour of minimum total cost, where the total cost is the sum of the costs of the links in the tour. With each link $(i, j) \in A$, a binary variable x_{ij}, which indicates whether link x_{ij} is part of the tour, and a cost c_{ij} are associated. Let $\delta(S) = \{(i, j) \in A \mid i \in S,\ j \notin S\}$. Then an integer linear programming formulation of the TSP 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, and the inequalities (subtour_elim) are the *subtour elimination constraints* (SECs), which enforce connectivity.

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 cities. Given the pairwise distances between each city, 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 OPTGRAPH, the traveling salesman problem solver can be invoked by using the TSP statement. The options for this statement are described in the section "TSP Statement" on page 44.

The traveling salesman problem solver reports status information in a macro variable called _OPTGRAPH_TSP_. See the section "Macro Variable _OPTGRAPH_TSP_" on page 179 for more information about this macro variable.

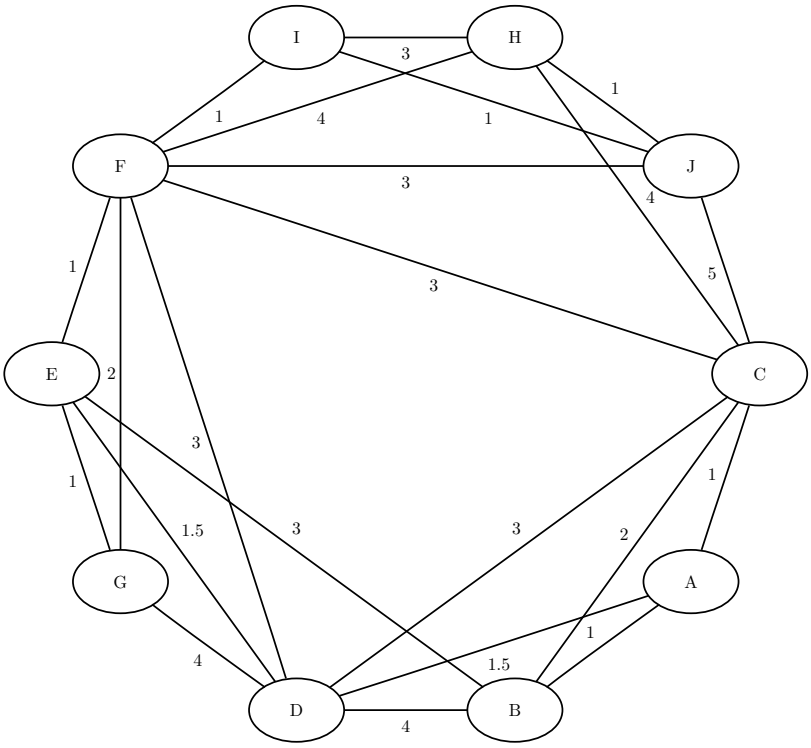
The algorithm used in PROC OPTGRAPH for solving TSP is based on a variant of the branch-and-cut process described in (Applegate et al. 2006).

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

Traveling Salesman Problem of a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 1.119.

Figure 1.119 A Simple Undirected Graph



The links data set can be represented 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 optgraph
   loglevel   = moderate
   data_links = LinkSetIn
   out_nodes  = NodeSetOut;
   tsp
     out      = TSPTour;
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_TSP_;
```

The progress of the procedure is shown in Figure 1.120.

Figure 1.120 PROC OPTGRAPH Log: Optimal Traveling Salesman Tour of a Simple Undirected Graph

```
NOTE: ------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------
NOTE: Running OPTGRAPH version 12.3.
NOTE: ------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------
NOTE: The OPTGRAPH 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: 0.00) seconds.
NOTE: The input graph storage is using 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 TSP statement.
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.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       2     16.0000000     16.0000000    0.00%       0
             0       0       2     16.0000000     16.0000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 16.
NOTE: Processing the traveling salesman problem used 0.00 (cpu: 0.00) seconds.
NOTE: ------------------------------------------------------------------------
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=15  CPU_TIME=0.00  REAL_TIME=0.00
```

The data set NodeSetOut now contains a sequence of nodes in the optimal tour and is shown in Figure 1.121.

Figure 1.121 Nodes in the Optimal Traveling Salesman Tour

Traveling Salesman Problem

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 1.122.

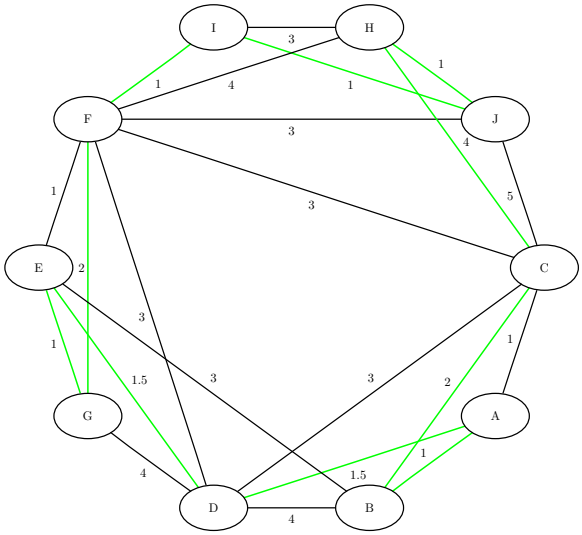
Figure 1.122 Links in the Optimal Traveling Salesman Tour

Traveling Salesman Problem

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 1.123.

Figure 1.123 Optimal Traveling Salesman Tour



ODS Table Names

Each table created by the OPTGRAPH procedure 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 1.53.

Table 1.53 ODS Tables Produced by PROC OPTGRAPH

Table Name	Description	Required Statement / Option
Details	Detailed real times for each phase of the procedure	PERFORMANCE with DETAILS option
PerformanceInfo	Information about the computing environment	Default output

Macro Variable _OPTGRAPH_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_. This variable contains a character string that indicates the status of the OPTGRAPH 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.
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 59. The BICONCOMP term can take one of the following values:

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

CENTR
indicates the status of the centrality algorithms at termination. These algorithms are described in the section "Centrality" on page 63. The CENTR 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.

CLIQUE
indicates the status of the clique-finding algorithms at termination. These algorithms are described in the section "Clique" on page 88. The CLIQUE term can take one of the following values:

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

COMMUNITY
indicates the status of the community algorithms at termination. These algorithms are described in the section "Community" on page 92. The COMMUNITY 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.

CONCOMP
indicates the status of the connected components algorithm at termination. This algorithm is described in the section "Connected Components" on page 100. The CONCOMP term can take one of the following values:

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

CORE
indicates the status of the core decomposition algorithm at termination. This algorithm is described in the section "Core Decomposition" on page 105. The CORE 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 109. 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 indicated by the MAXTIME= option in the CYCLE statement.
SOLUTION_LIM	The algorithm reached its limit on the number of cycles found, which is indicated by the MAXCYCLES= option in the CYCLE statement.
ERROR	The algorithm encountered an error.

EIGEN
indicates the status of the eigenvector solver at termination. This solver is described in the section "Eigenvector Problem" on page 115. The EIGEN term can take one of the following values:

OK	The solver terminated normally.
ERROR	The solver 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 118. 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 125. The MCF term can take one of the following values:

OPTIMAL	The solution is optimal.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
TIMELIMIT	The solver reached its execution time limit, which is indicated by the MAXTIME= option in the MINCOSTFLOW statement.
ERROR	The solver encountered an error.

MINCUT

indicates the status of the minimum cut algorithm at termination. This algorithm is described in the section "Minimum Cut" on page 119. The MINCUT 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.

MINSPANTREE

indicates the status of the minimum spanning tree solver at termination. This solver is described in the section "Minimum Spanning Tree" on page 123. The MINSPANTREE term can take one of the following values:

OPTIMAL	The solution is optimal.
ERROR	The solver encountered an error.

REACH

indicates the status of the reach algorithms at termination. These algorithms are described in the section "Reach (Ego) Network" on page 130. The REACH 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.

SHORTPATH

indicates the status of the shortest path algorithms at termination. These algorithms are described in the section "Shortest Path" on page 143. 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.

SUMMARY

indicates the status of the summary algorithms at termination. These algorithms are described in the section "Summary" on page 155. The SUMMARY 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.

TRANSITIVE_CLOSURE

indicates the status of the transitive closure algorithm at termination. This algorithm is described in the section "Transitive Closure" on page 161. The TRANSITIVE_CLOSURE term can take one of the following values:

OK	The algorithm terminated normally.
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 164. 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 specified in the ABSOBJGAP= option.
OPTIMAL_RGAP	The solution is optimal within the relative gap specified in the RELOBJGAP= option.
OPTIMAL_COND	The solution is optimal, but some infeasibilities (primal, bound, or integer) exceed tolerances due to scaling or choice of a small INTTOL= value.
TARGET	The solution is not worse than the target specified 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 _OPTGRAPH_BICONCOMP_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_BICONCOMP_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used 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 _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

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 _OPTGRAPH_CENTR_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_CENTR_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used to calculate centrality. 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 CENTR in the _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

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 _OPTGRAPH_CLIQUE_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_CLIQUE_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used 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 _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

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 _OPTGRAPH_COMMUNITY_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_COMMUNITY_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used to calculate communities. 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 COMMUNITY in the _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

NUM_COMMUNITIES
indicates the number of communities found by the algorithm.

MODULARITY
indicates the final modularity 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 _OPTGRAPH_CONCOMP_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_CONCOMP_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used 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 _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

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 _OPTGRAPH_CORE_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_CORE_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used to calculate the core decomposition. 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 CORE in the _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

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 _OPTGRAPH_CYCLE_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_CYCLE_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used 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 _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

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 _OPTGRAPH_EIGEN_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_EIGEN_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used to calculate eigenvectors. 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 EIGEN in the _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

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 _OPTGRAPH_LAP_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_LAP_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used 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 _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

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 _OPTGRAPH_MCF_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_MCF_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used 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 _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

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 _OPTGRAPH_MINCUT_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_MINCUT_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used 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 _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

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 _OPTGRAPH_MST_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_MST_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used 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 MINSPANTREE in the _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

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 _OPTGRAPH_REACH_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_REACH_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used to calculate reach networks. 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 REACH in the _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

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 _OPTGRAPH_SHORTPATH_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_SHORTPATH_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used 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 _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

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 _OPTGRAPH_SUMMARY_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_SUMMARY_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used to calculate summary statistics. 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 SUMMARY in the _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

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 _OPTGRAPH_TRANSCL_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_TRANSCL_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used 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 TRANSITIVE_CLOSURE in the _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

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 _OPTGRAPH_TSP_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_TSP_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm used 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 _OPTGRAPH_ macro as defined in the section "Macro Variable _OPTGRAPH_" on page 168.

OBJECTIVE
indicates the objective value obtained by the solver 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

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

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.

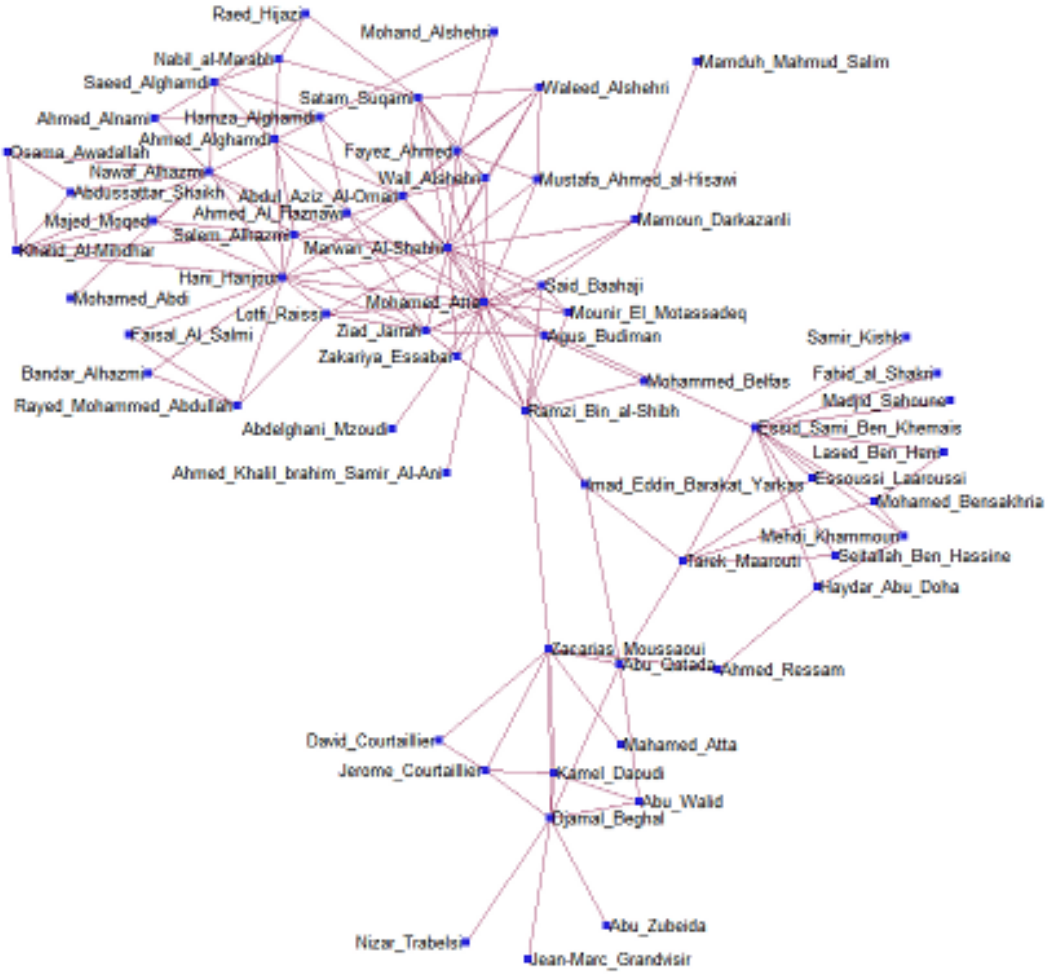
NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time (default) or real time. The type is determined by the TIMETYPE= option.

Examples: OPTGRAPH Procedure

Example 1.1: Articulation Points in a Terrorist Network

This example considers the terrorist communications network from the attacks on the U.S. on September 11, 2001, described in Krebs 2002. Figure 1.124 shows this network, which was constructed after the attacks, based on collected intelligence information. The image was created using SAS/GRAPH® Network Visualization Workshop 2.1 (see the *SAS/GRAPH: Network Visualization Workshop User's Guide*).

Figure 1.124 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;
   length from $25 to $32;
   input from to;
   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 optgraph
   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 1.1.1 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

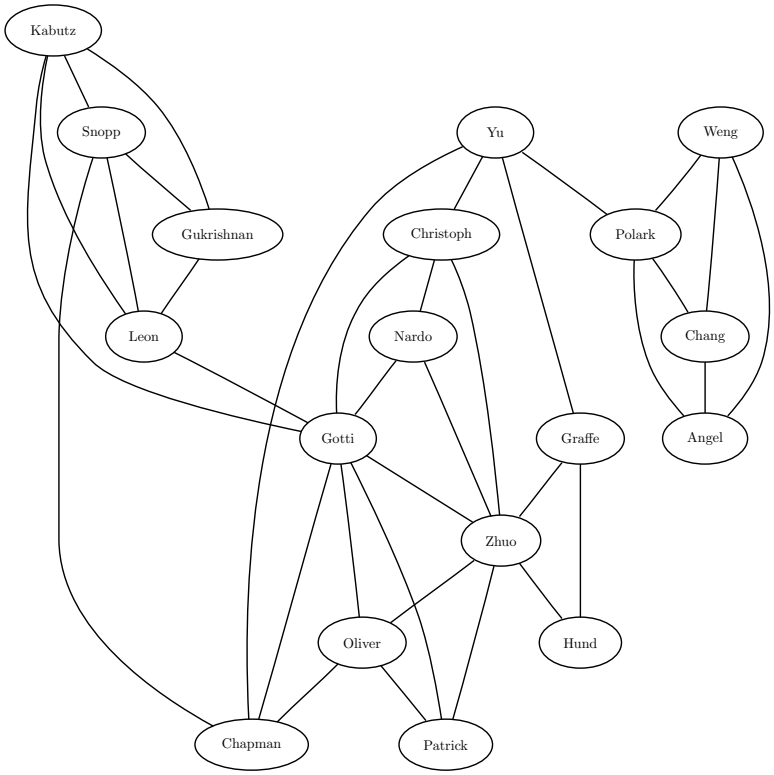
Example 1.2: Influence Centrality for Project Groups in a Research Department

This example looks at an undirected graph that represents a few of the project groups in a hypothetical research department. A link between nodes A and B means that person A and B work together or that person A reports to person B. This graph represents six main projects.

- Department 1 (D1) consists of Snopp, Gukrishnan, Leon, and Kabutz. Snopp reports to Chapman.
- Department 2 (D2) consists of Oliver, Gotti, Patrick, and Zhuo. Oliver reports to Chapman.
- Department 3 (D3) consists of Gotti, Leon, and Kabutz. Gotti reports to Chapman.
- Department 4 (D4) consists of the following project teams who report to Yu. Yu reports to Chapman on this project.
  - Department 4a (D4a) consists of Polark, Chang, Weng, and Angel. Polark reports to Yu.
  - Department 4b (D4b) consists of Christoph, Nardo, Gotti, and Zhuo. Christoph reports to Yu.
  - Department 4c (D4c) consists of Graffe, Zhuo, and Hund. Graffe reports to Yu.

The links are shown in Figure 1.125.

Figure 1.125 Project Groups in a Research Department



The link weights measure the reporting magnitude. In general the higher the weight, the higher the contribution to the influence metric. Chapman is the director of the overall department, and Yu is the manager of a subgroup. The leads for the projects D1, D2, and D3 report to Chapman, and the leads for D4a, D4b, and D4c report to Yu. Reporting links to the director, Chapman, are given a link weight of 3, and reporting links to Yu are given a weight of 2. Links that represent people working together on a project all receive equal weight of 1. The node weights also represent some level of reporting: directors (4), managers (3), leads (2), and all others (1).

The project graph can be represented in the following link and node data sets:

```
data LinkSetInDept;
   input from $1-12 to $13-24 weight;
   datalines;
Yu           Chapman      3
Gotti        Chapman      3
Oliver       Chapman      3
Snopp        Chapman      3
Gukrishnan   Leon         1
Snopp        Gukrishnan   1
Kabutz       Gukrishnan   1
Kabutz       Snopp        1
Snopp        Leon         1
Kabutz       Leon         1
Gotti        Oliver       1
Gotti        Patrick      1
Oliver       Patrick      1
```

```
Zhuo        Oliver      1
Zhuo        Gotti       1
Zhuo        Patrick     1
Kabutz      Gotti       1
Leon        Gotti       1
Polark      Yu          2
Polark      Chang       1
Chang       Angel       1
Polark      Angel       1
Weng        Polark      1
Weng        Chang       1
Weng        Angel       1
Christoph   Yu          2
Christoph   Nardo       1
Christoph   Gotti       1
Christoph   Zhuo        1
Nardo       Gotti       1
Nardo       Zhuo        1
Graffe      Yu          2
Graffe      Hund        1
Graffe      Zhuo        1
Zhuo        Hund        1
;

data NodeSetInDept;
   input node $1-12 weight;
   datalines;
Chapman      4
Yu           3
Gotti        2
Polark       2
Christoph    2
Oliver       2
Snopp        2
Zhuo         1
Nardo        1
Weng         1
Chang        1
Hund         1
Graffe       1
Leon         1
Gukrishnan   1
Kabutz       1
Patrick      1
Angel        1
;
```

The following statements calculate influence centrality (in addition to degree centrality):

```
proc optgraph
   loglevel     = moderate
   data_links   = LinkSetInDept
   data_nodes   = NodeSetInDept
   out_nodes    = NodeSetOut;
```

```
   centrality
      degree    = out
      influence = weight;
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_CENTR_;
```

The progress of the procedure is shown in Output 1.2.1.

Output 1.2.1 PROC OPTGRAPH Log: Influence Centrality for Project Groups in a Research Department

```
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Running OPTGRAPH version 12.3.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: The OPTGRAPH procedure is executing in single-machine mode.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: Reading the nodes data set.
NOTE: There were 18 observations read from the data set WORK.NODESETINDEPT.
NOTE: There were 35 observations read from the data set WORK.LINKSETINDEPT.
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 of memory.
NOTE: The number of nodes in the input graph is 18.
NOTE: The number of links in the input graph is 35.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Processing CENTRALITY statement.
NOTE: ---------------------------------------------------------------------------
NOTE: Processing degree centrality.
NOTE: The centrality algorithms are using 0.0 MBs of memory.
NOTE: Processing degree centrality used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------
NOTE: Processing influence centrality.
NOTE: The centrality algorithms are using 0.0 MBs of memory.
NOTE: Processing influence centrality used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------
NOTE: Processing centrality used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Creating nodes data set output.
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: The data set WORK.NODESETOUT has 18 observations and 5 variables.
STATUS=OK  CENTR=OK
STATUS=OK  CPU_TIME=0.00  REAL_TIME=0.00
```

The node data set NodeSetOut now contains the weighted influence centrality of the department's graph, including C_1 (variable centr_influence1_wt) and C_2 (variable centr_influence2_wt). This data set is shown in Output 1.2.2.

Output 1.2.2 Influence Centrality for Project Groups in a Research Department

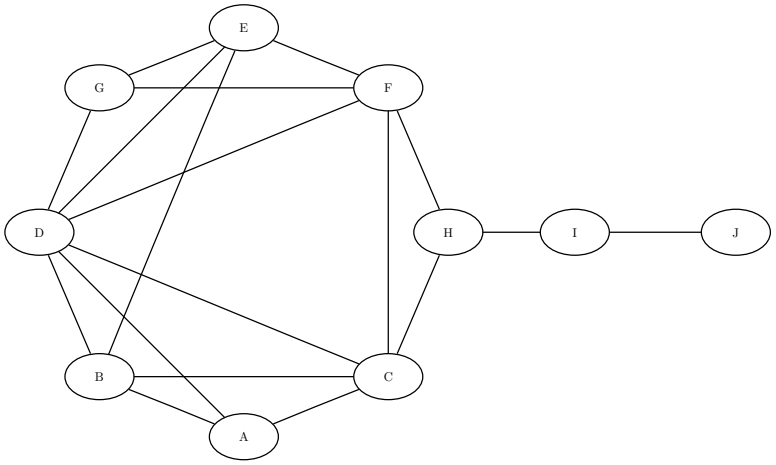
node	weight	centr_ degree_ out	centr_ influence1_ wt	centr_ influence2_ wt
Gotti	2	8	0.35714	1.57143
Zhuo	1	7	0.25000	1.17857
Oliver	2	4	0.21429	1.14286
Chapman	4	4	0.42857	1.10714
Christoph	2	4	0.17857	1.03571
Yu	3	4	0.32143	0.92857
Snopp	2	4	0.21429	0.82143
Leon	1	4	0.14286	0.82143
Patrick	1	3	0.10714	0.82143
Kabutz	1	4	0.14286	0.82143
Nardo	1	3	0.10714	0.78571
Polark	2	4	0.17857	0.64286
Graffe	1	3	0.14286	0.64286
Gukrishnan	1	3	0.10714	0.50000
Weng	1	3	0.10714	0.39286
Chang	1	3	0.10714	0.39286
Hund	1	2	0.07143	0.39286
Angel	1	3	0.10714	0.39286

As expected, the director Chapman has the highest first-order influence, since the weights of the reporting links to him are high. The highest second-order influence is Gotti, who reports to the director but is also involved in three different projects and therefore has a large sphere of influence. This example is revisited with other centrality metrics in other examples.

Example 1.3: Betweenness and Closeness Centrality for Computer Network Topology

Consider a small network of 10 computers spread out across an office. Let a node represent a computer, and let a link represent a direct connection between the machines. For this example, consider the links as Ethernet connections that enable data to transfer between computers. If two computers are not connected directly, then the information must flow through other connected machines. Consider a topology as shown in Figure 1.126. This is an example of the well-known *kite network*, which was popularized by David Krackhardt (1990) for better understanding of social networks in the workplace.

Figure 1.126 Office Computer Network



Define the link data set as follows:

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

To better understand the topology of the computer network, calculate the degree, closeness, and betweenness centrality. It is also interesting to look for articulation points in the computer network to identify places of vulnerability. All of these calculations can be done in one call to PROC OPTGRAPH as follows:

```
proc optgraph
   data_links = LinkSetInCompNet
   out_links  = LinkSetOut
   out_nodes  = NodeSetOut;
   centrality
      degree  = out
      close   = unweight
      between = unweight;
   biconcomp;
run;
```

Output 1.3.1 shows the resulting node data set NodeSetOut sorted by closeness.

Output 1.3.1 Node Closeness and Betweenness Centrality, Sorted by Closeness

node	centr_ degree_ out	centr_ close_ unwt	centr_ between_ unwt	artpoint
C	5	0.64286	0.23148	0
F	5	0.64286	0.23148	0
D	6	0.60000	0.10185	0
H	3	0.60000	0.38889	1
B	4	0.52941	0.02315	0
E	4	0.52941	0.02315	0
A	3	0.50000	0.00000	0
G	3	0.50000	0.00000	0
I	2	0.42857	0.22222	1
J	1	0.31034	0.00000	0

Output 1.3.2 shows the resulting node (NodeSetOut) and link data sets (LinkSetOut) sorted by betweenness.

Output 1.3.2 Node Closeness and Betweenness Centrality, Sorted by Betweenness

Obs	node	centr_ degree_ out	centr_ close_ unwt	centr_ between_ unwt	artpoint
1	H	3	0.60000	0.38889	1
2	C	5	0.64286	0.23148	0
3	F	5	0.64286	0.23148	0
4	I	2	0.42857	0.22222	1
5	D	6	0.60000	0.10185	0
6	E	4	0.52941	0.02315	0
7	B	4	0.52941	0.02315	0
8	A	3	0.50000	0.00000	0
9	G	3	0.50000	0.00000	0
10	J	1	0.31034	0.00000	0

Output 1.3.2 *continued*

Obs	from	to	biconcomp	centr_ between_ unwt
1	H	I	2	0.44444
2	C	H	3	0.29167
3	F	H	3	0.29167
4	I	J	1	0.25000
5	E	F	3	0.12963
6	B	C	3	0.12963
7	A	C	3	0.12500
8	F	G	3	0.12500
9	C	D	3	0.09259
10	D	F	3	0.09259
11	A	D	3	0.08333
12	D	G	3	0.08333
13	C	F	3	0.07407
14	B	E	3	0.07407
15	B	D	3	0.05093
16	D	E	3	0.05093
17	A	B	3	0.04167
18	E	G	3	0.04167

The computers with the highest closeness centrality are C and F, because they have the shortest paths to all other nodes. These computers are key to the efficient distribution of information across the network. Assuming that the entire office has some centralized data that should be shared with all computers, machines C and F would be the best candidates for storing the data on their local hard drives. The computer with the highest betweenness centrality is H. Although machine H has only three connections, it is one of the most important machines in the office because it serves as the only way to reach computers I and J from the other machines in the office. Notice also that machine H is an articulation point because removing it would disconnect the office network. In this setting, computers with high betweenness should be carefully maintained and secured with UPS (uninterruptible power supply) systems to ensure they are always online.

Example 1.4: Betweenness and Closeness Centrality for Project Groups in a Research Department

This example uses the same data as are used in the section "Example 1.2: Influence Centrality for Project Groups in a Research Department" on page 183, which illustrates influence centrality by considering the link weights that represent some measure of reporting magnitude. In Example 1.2, links between managers (or leads) and direct reports had higher link weights than links between non-managers. This interpretation makes sense in the context of influence centrality because weight and the metric are directly related. However, weight and the metric are inversely related for closeness and betweenness centrality.

This example considers the speed of the flow of information between people. In this sense, connections between managers and direct reports have *smaller values*, which cost less in the shortest path calculations. The following DATA step produces a new links data set, based on LinkSetInDept, which uses the inverse of the weight:

```
data LinkSetInDeptInv;
   set LinkSetInDept;
   weight = 1 / weight;
run;
```

The following statements calculate weighted (and unweighted) closeness and betweenness centrality. Notice that this example also uses the NTHREADS= option in the PERFORMANCE statement to specify two threads to allow the computation to be run in parallel. Since these data have 18 nodes, each thread can process 9 nodes simultaneously.

```
proc optgraph
   loglevel     = moderate
   data_links   = LinkSetInDeptInv
   out_links    = LinkSetOut
   out_nodes    = NodeSetOut;
   performance
      nthreads  = 2;
   centrality
      close     = both
      between   = both;
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_CENTR_;
```

The progress of the procedure is shown in Output 1.4.1.

Output 1.4.1 PROC OPTGRAPH Log: Closeness and Node Betweenness Centrality for Project Groups in a Research Department

```
NOTE: ----------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------
NOTE: Running OPTGRAPH version 12.3.
NOTE: ----------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------
NOTE: The OPTGRAPH procedure is executing in single-machine mode.
NOTE: ----------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: There were 35 observations read from the data set WORK.LINKSETINDEPTINV.
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 of memory.
NOTE: The number of nodes in the input graph is 18.
NOTE: The number of links in the input graph is 35.
NOTE: ----------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------
NOTE: Processing CENTRALITY statement.
NOTE: ----------------------------------------------------------------------
NOTE: Processing weighted between/close centrality using 2 threads.
                                                    Cpu     Real   Active
      Algorithm                Nodes   Complete     Time     Time  Threads
      betwNL/close(wt)            18       100%     0.02     0.02        0
NOTE: The centrality algorithms are using 0.0 MBs of memory.
NOTE: Processing weighted between/close centrality used 0.02 (cpu: 0.02)
      seconds.
NOTE: ----------------------------------------------------------------------
NOTE: Processing unweighted between/close centrality using 2 threads.
                                                    Cpu     Real   Active
      Algorithm                Nodes   Complete     Time     Time  Threads
      betwNL/close(unwt)          18       100%     0.00     0.00        0
NOTE: The centrality algorithms are using 0.0 MBs of memory.
NOTE: Processing unweighted between/close centrality used 0.00 (cpu: 0.00)
      seconds.
NOTE: ----------------------------------------------------------------------
NOTE: Processing centrality used 0.02 (cpu: 0.02) seconds.
NOTE: ----------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------
NOTE: Creating nodes data set output.
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 35 observations and 5 variables.
NOTE: The data set WORK.NODESETOUT has 18 observations and 5 variables.
STATUS=OK  CENTR=OK
STATUS=OK  CPU_TIME=0.02  REAL_TIME=0.02
```

The node data set NodeSetOut shows the weighted and unweighted closeness and node betweenness centrality, as shown in Output 1.4.2.

Output 1.4.2 Closeness and Betweenness Centrality for Project Groups in a Research Department

node	centr_ close_wt	centr_ close_ unwt	centr_ between_ wt	centr_ between_ unwt
Yu	0.87179	0.50000	0.50000	0.41262
Chapman	0.88696	0.50000	0.44118	0.23235
Gotti	0.81600	0.51515	0.20956	0.28444
Oliver	0.73913	0.44737	0.04044	0.02230
Snopp	0.75556	0.38636	0.16176	0.08088
Gukrishnan	0.46575	0.32692	0.00000	0.00000
Leon	0.50746	0.38636	0.00000	0.03885
Kabutz	0.50746	0.38636	0.00000	0.03885
Patrick	0.50000	0.37778	0.00000	0.00000
Zhuo	0.58286	0.47222	0.06618	0.15172
Polark	0.69388	0.38636	0.30882	0.30882
Chang	0.44156	0.29310	0.00000	0.00000
Angel	0.44156	0.29310	0.00000	0.00000
Weng	0.44156	0.29310	0.00000	0.00000
Christoph	0.68456	0.48571	0.05882	0.11275
Nardo	0.51777	0.42500	0.00000	0.00000
Graffe	0.67105	0.43590	0.08088	0.06642
Hund	0.45133	0.36957	0.00000	0.00000

The link data set LinkSetOut shows the weighted and unweighted link betweenness centrality, as shown in Output 1.4.3.

Output 1.4.3 Link Betweenness Centrality for Project Groups in a Research Department

from	to	weight	centr_ between_ wt	centr_ between_ unwt
Yu	Chapman	0.33333	0.39706	0.25576
Gotti	Chapman	0.33333	0.20221	0.09767
Oliver	Chapman	0.33333	0.14338	0.07623
Snopp	Chapman	0.33333	0.26471	0.16005
Gukrishnan	Leon	1.00000	0.00735	0.03431
Snopp	Gukrishnan	1.00000	0.11029	0.05637
Kabutz	Gukrishnan	1.00000	0.00735	0.03431
Kabutz	Snopp	1.00000	0.03676	0.03517
Snopp	Leon	1.00000	0.03676	0.03517
Kabutz	Leon	1.00000	0.00735	0.00735
Gotti	Oliver	1.00000	0.00000	0.03431
Gotti	Patrick	1.00000	0.05882	0.06066
Oliver	Patrick	1.00000	0.03676	0.02022
Zhuo	Oliver	1.00000	0.02574	0.03885
Zhuo	Gotti	1.00000	0.05515	0.10184
Zhuo	Patrick	1.00000	0.02941	0.04412
Kabutz	Gotti	1.00000	0.07353	0.12586
Leon	Gotti	1.00000	0.07353	0.12586
Polark	Yu	0.50000	0.41176	0.41176
Polark	Chang	1.00000	0.11029	0.11029
Chang	Angel	1.00000	0.00735	0.00735
Polark	Angel	1.00000	0.11029	0.11029
Weng	Polark	1.00000	0.11029	0.11029
Weng	Chang	1.00000	0.00735	0.00735
Weng	Angel	1.00000	0.00735	0.00735
Christoph	Yu	0.50000	0.13603	0.15870
Christoph	Nardo	1.00000	0.04779	0.04412
Christoph	Gotti	1.00000	0.02574	0.09620
Christoph	Zhuo	1.00000	0.03309	0.05147
Nardo	Gotti	1.00000	0.05515	0.05147
Nardo	Zhuo	1.00000	0.02206	0.02941
Graffe	Yu	0.50000	0.18015	0.12402
Graffe	Hund	1.00000	0.06985	0.04804
Graffe	Zhuo	1.00000	0.03676	0.08578
Zhuo	Hund	1.00000	0.05515	0.07696

Note that Chapman (director) and Yu (manager, reporting to Chapman) both have the highest weighted closeness centrality. However, Yu's weighted betweenness centrality is highest because he serves as more of a *gatekeeper* between his three groups (D4a, D4b, and D4c) and the rest of the department.

Example 1.5: Eigenvector Centrality for Word Sense Disambiguation

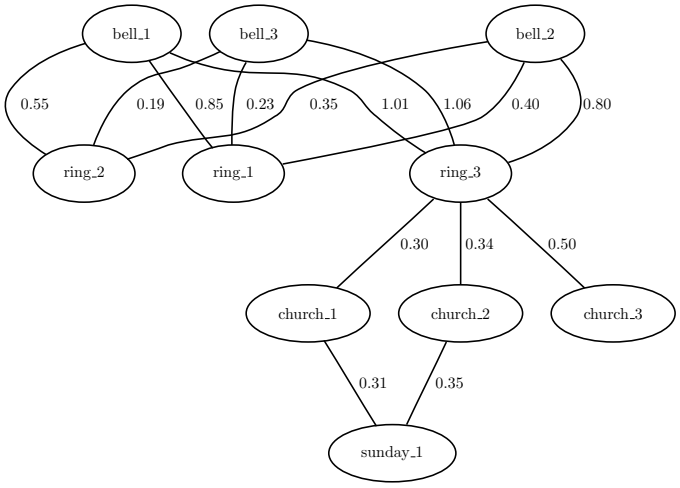
In many languages, numerous words are polysemous (they carry more than one meaning). A common task in information retrieval is to assign the correct meaning to a polysemous word within a given context. Take the word "bass" as an example. It can mean either *a type of fish* (as in the sentence "I went fishing for some sea bass") or *tones of low frequency* (as in the sentence "The bass part of the song is very moving").

The following example from Mihalcea 2005 shows how eigenvector centrality can be used to disambiguate the word sense in the sentence "The church bells no longer ring on Sundays." The following senses of words can be drawn from a dictionary:

- *church*
    1. one of the groups of Christians who have their own beliefs and forms of worship
    2. a place for public (especially Christian) worship
    3. a service conducted in a church
- *bell*
    1. a hollow device made of metal that makes a ringing sound when struck
    2. a push button at an outer door that gives a ringing or buzzing signal when pushed
    3. the sound of a bell
- *ring*
    1. make a ringing sound
    2. ring or echo with sound
    3. make (bells) ring, often for the purposes of musical edification
- *Sunday*
    1. first day of the week; observed as a day of rest and worship by most Christians

Using one of the similarity metrics defined in Sinha and Mihalcea 2007, you can generate a graph in which the nodes correspond to the word senses given above and the weights are determined by the similarity metric. The resulting graph is shown in Figure 1.127.

Figure 1.127 Eigenvector Centrality for Word Sense Disambiguation



To identify the correct senses, you run eigenvector centrality on the graph and select the highest ranking sense for each word:

```
data LinkSetIn;
   input from $ to $ weight;
   datalines;
bell_1   ring_1   0.85
bell_1   ring_2   0.55
bell_1   ring_3   1.01
bell_2   ring_1   0.40
bell_2   ring_2   0.35
bell_2   ring_3   0.80
bell_3   ring_1   0.23
bell_3   ring_2   0.19
bell_3   ring_3   1.06
ring_3   church_1 0.30
ring_3   church_2 0.34
ring_3   church_3 0.50
church_1 sunday_1 0.31
church_2 sunday_1 0.35
;

proc optgraph
   data_links = LinkSetIn
   out_nodes  = NodeSetOut;
   centrality
      eigen   = weight;
run;

data NodeSetOut;
   length word $8 sense $1;
   set NodeSetOut;
   word  = scan(node,1,'_');
   sense = scan(node,2,'_');
run;
```

```
proc sort
   data = NodeSetOut
   out  = WordSenses;
   by word descending centr_eigen_wt;
run;

data WordSenses;
   set WordSenses(drop=centr_eigen_wt);
   by word;
   if first.word then output;
run;
```

The eigenvector scores and the implied word sense are shown in Output 1.5.1.

Output 1.5.1 Eigenvector Centrality for Word Sense Disambiguation

node	centr_eigen_wt
ring_3	1.00000
bell_1	0.77997
bell_3	0.59692
bell_2	0.53889
ring_1	0.48924
ring_2	0.35207
church_3	0.24081
church_2	0.17248
church_1	0.15222
sunday_1	0.05180

word	sense	node
bell	1	bell_1
church	3	church_3
ring	3	ring_3
sunday	1	sunday_1

Example 1.6: Centrality Metrics for Project Groups in a Research Department

The following statements use the WEIGHT2= option, and the project groups in a research department as depicted in Figure 1.125 on page 184. The data set contains the original weight and its inverse, which is used in the calculations of closeness and betweenness.

```
data LinkSetInDept2;
   input from $1-12 to $13-24 weight weightInv;
   datalines;
Yu          Chapman     3  0.33
Gotti       Chapman     3  0.33
Oliver      Chapman     3  0.33
Snopp       Chapman     3  0.33
Gukrishnan  Leon        1  1
```

```
Snopp       Gukrishnan  1  1
Kabutz      Gukrishnan  1  1
Kabutz      Snopp       1  1
Snopp       Leon        1  1
Kabutz      Leon        1  1
Gotti       Oliver      1  1
Gotti       Patrick     1  1
Oliver      Patrick     1  1
Zhuo        Oliver      1  1
Zhuo        Gotti       1  1
Zhuo        Patrick     1  1
Kabutz      Gotti       1  1
Leon        Gotti       1  1
Polark      Yu          2  0.50
Polark      Chang       1  1
Chang       Angel       1  1
Polark      Angel       1  1
Weng        Polark      1  1
Weng        Chang       1  1
Weng        Angel       1  1
Christoph   Yu          2  0.50
Christoph   Nardo       1  1
Christoph   Gotti       1  1
Christoph   Zhuo        1  1
Nardo       Gotti       1  1
Nardo       Zhuo        1  1
Graffe      Yu          2  0.50
Graffe      Hund        1  1
Graffe      Zhuo        1  1
Zhuo        Hund        1  1
;

proc optgraph
   data_nodes   = NodeSetInDept
   data_links   = LinkSetInDept2
   out_nodes    = NodeSetOut;
   performance
      nthreads  = 2;
   centrality
      clustering_coef
      degree    = out
      influence = weight
      close     = weight
      between   = weight
      eigen     = weight
      weight2   = weightInv;
run;
```

The node data set NodeSetOut now shows the resulting centrality metrics given both weight interpretations.

Output 1.6.1 Centrality for Project Groups in a Research Department

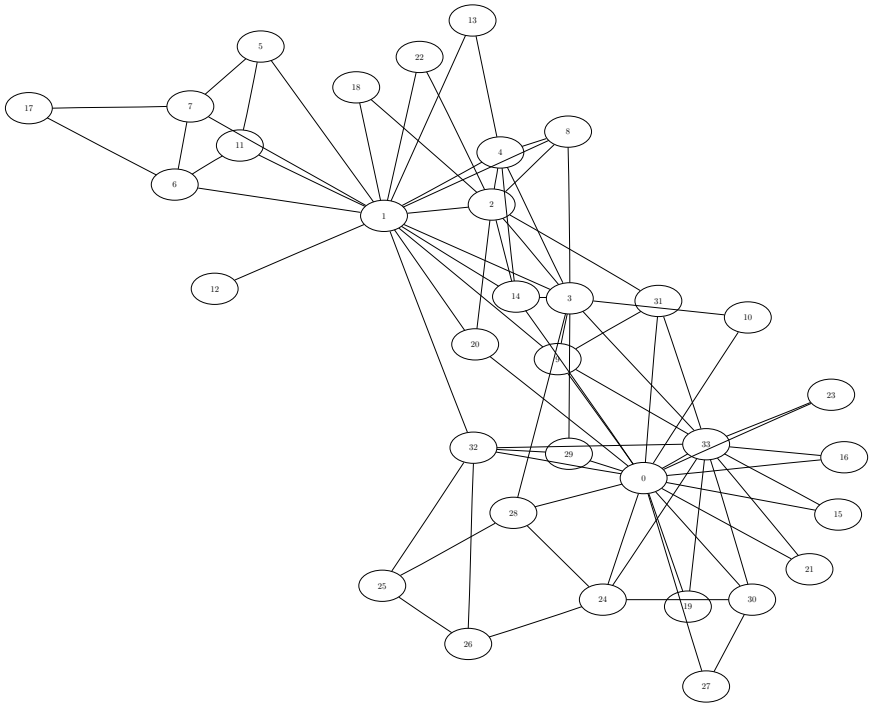
node	weight	centr_ degree_ out	centr_ eigen_wt	centr_ close_wt
Chapman	4	4	1.00000	0.88959
Yu	3	4	0.62475	0.87404
Gotti	2	8	0.70480	0.81849
Polark	2	4	0.18777	0.69530
Christoph	2	4	0.34168	0.68521
Oliver	2	4	0.58858	0.74203
Snopp	2	4	0.49133	0.75859
Zhuo	1	7	0.32567	0.58319
Nardo	1	3	0.18983	0.51813
Weng	1	3	0.03591	0.44213
Chang	1	3	0.03591	0.44213
Hund	1	2	0.07667	0.45153
Graffe	1	3	0.22852	0.67220
Leon	1	4	0.21239	0.50822
Gukrishnan	1	3	0.12674	0.46690
Kabutz	1	4	0.21239	0.50822
Patrick	1	3	0.22398	0.50074
Angel	1	3	0.03591	0.44213

centr_ between_ wt	centr_ influence1_ wt	centr_ influence2_ wt	centr_ cluster
0.44118	0.42857	1.10714	0.16667
0.50000	0.32143	0.92857	0.00000
0.20956	0.35714	1.57143	0.28571
0.30882	0.17857	0.64286	0.50000
0.05882	0.17857	1.03571	0.50000
0.04044	0.21429	1.14286	0.66667
0.16176	0.21429	0.82143	0.50000
0.06618	0.25000	1.17857	0.33333
0.00000	0.10714	0.78571	1.00000
0.00000	0.10714	0.39286	1.00000
0.00000	0.10714	0.39286	1.00000
0.00000	0.07143	0.39286	1.00000
0.08088	0.14286	0.64286	0.33333
0.00000	0.14286	0.82143	0.66667
0.00000	0.10714	0.50000	1.00000
0.00000	0.14286	0.82143	0.66667
0.00000	0.10714	0.82143	1.00000
0.00000	0.10714	0.39286	1.00000

Example 1.7: Community Detection on Zachary's Karate Club Data

This example uses Zachary's Karate Club data (Zachary 1977), which describes social network friendships between 34 members of a karate club at a U.S. university in the 1970s. This is one of the standard publicly available data sets for testing community detection algorithms. It contains 34 nodes and 78 links. The graph is shown in Figure 1.128.

Figure 1.128 Zachary's Karate Club Graph



The graph can be represented using the links data set LinkSetIn as follows:

```
data LinkSetIn;
   input from to weight @@;
   datalines;
 0  9  1  0 10  1  0 14  1  0 15  1  0 16  1  0 19  1  0 20  1  0 21  1
 0 23  1  0 24  1  0 27  1  0 28  1  0 29  1  0 30  1  0 31  1  0 32  1
 0 33  1  2  1  1  3  1  1  3  2  1  4  1  1  4  2  1  4  3  1  5  1  1
 6  1  1  7  1  1  7  5  1  7  6  1  8  1  1  8  2  1  8  3  1  8  4  1
 9  1  1  9  3  1 10  3  1 11  1  1 11  5  1 11  6  1 12  1  1 13  1  1
13  4  1 14  1  1 14  2  1 14  3  1 14  4  1 17  6  1 17  7  1 18  1  1
18  2  1 20  1  1 20  2  1 22  1  1 22  2  1 26 24  1 26 25  1 28  3  1
28 24  1 28 25  1 29  3  1 30 24  1 30 27  1 31  2  1 31  9  1 32  1  1
32 25  1 32 26  1 32 29  1 33  3  1 33  9  1 33 15  1 33 16  1 33 19  1
33 21  1 33 23  1 33 24  1 33 30  1 33 31  1 33 32  1
;
```

The following statements use the RESOLUTION_LIST= option to represent resolution levels (1, 0.5) in community detection on the Karate Club data. For more information about resolution levels, see the section "Resolution List" on page 95.

```
proc optgraph
   data_links            = LinkSetIn
   out_nodes             = NodeSetOut
   graph_internal_format = thin;
   community
      resolution_list    = 1.0 0.5
      out_level          = CommLevelOut
      out_community      = CommOut
      out_overlap        = CommOverlapOut
      out_comm_links     = CommLinksOut;
run;
```

The data set NodeSetOut contains the community identifier of each node. It is shown in Output 1.7.1.

Output 1.7.1 Community Nodes Output

node	community_ 1	community_ 2
0	0	0
9	0	0
10	1	1
14	1	1
15	0	0
16	0	0
19	0	0
20	1	1
21	0	0
23	0	0
24	2	0
27	0	0
28	2	0
29	2	0
30	0	0
31	0	0
32	2	0
33	0	0
2	1	1
1	1	1
3	1	1
4	1	1
5	3	1
6	3	1
7	3	1
8	1	1
11	3	1
12	1	1
13	1	1
17	3	1
18	1	1
22	1	1
26	2	0
25	2	0

Column community_1 contains the community identifier of each node when the resolution value is 1.0; column community_2 contains the community identifier of each node when the resolution value is 0.5. Different node colors are used to represent different communities in Figure 1.129 and Figure 1.130. As you can see from the figures, four communities at resolution 1.0 are merged to two communities at resolution 0.5.

Figure 1.129 Karate Club Communities (Resolution = 1.0)

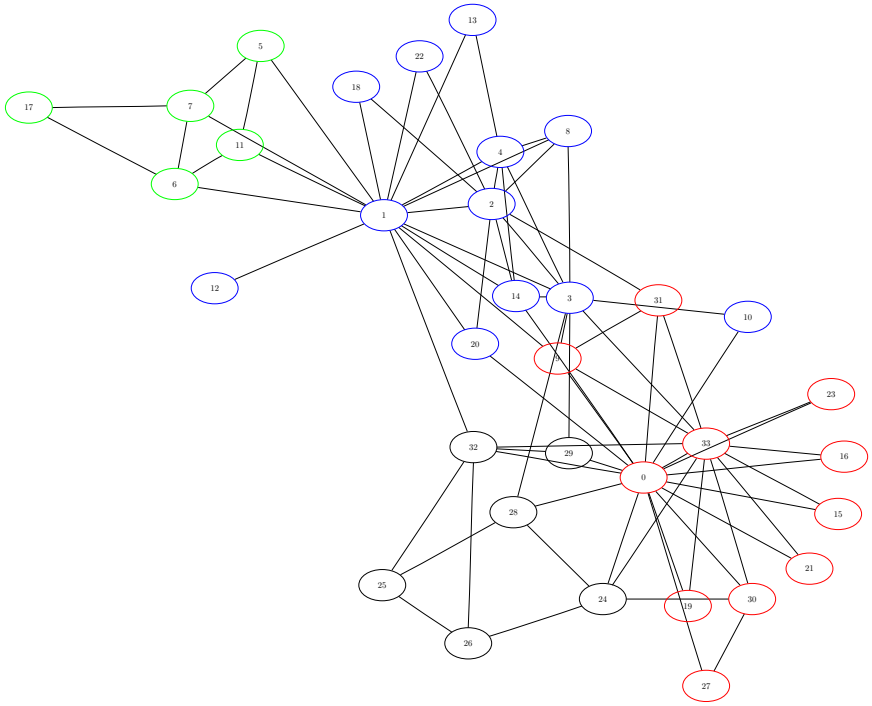
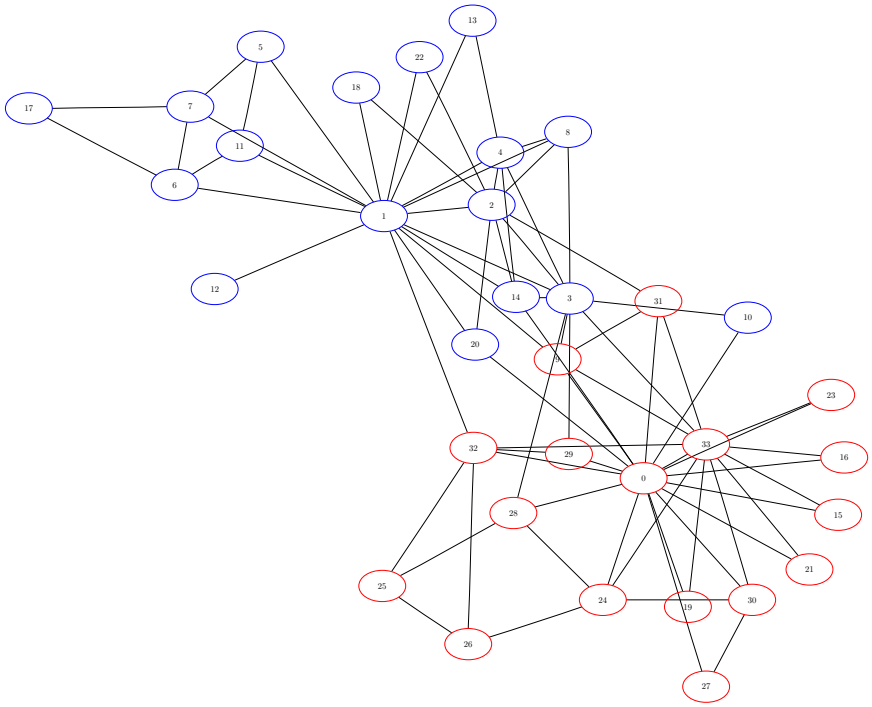


Figure 1.130 Karate Club Communities (Resolution = 0.5)



The data set CommLevelOut contains the number of communities and the corresponding modularity values found at each resolution level. It is shown in Output 1.7.2.

Output 1.7.2 Community Level Summary Output

level	resolution	communities	modularity
1	1.0	4	0.41880
2	0.5	2	0.37179

The data set CommOut contains the number of nodes contained in each community. It is shown in Output 1.7.3.

Output 1.7.3 Community Number of Nodes Output

level	resolution	community	nodes
1	1.0	0	11
1	1.0	1	12
1	1.0	2	6
1	1.0	3	5
2	0.5	0	17
2	0.5	1	17

The data set CommOverlapOut contains the intensity of each node that belongs to multiple communities. It is shown in Output 1.7.4. Note that only the communities in the last resolution level (the smallest resolution value) are output in this data set. In this example, Node 0 belongs to two communities, with 82.3% of its links connecting to Community 0, and 17.6% of its links connecting to Community 1.

Output 1.7.4 Community Overlap Output

node	community	intensity
0	0	0.82353
0	1	0.17647
9	0	0.60000
9	1	0.40000
10	0	0.50000
10	1	0.50000
14	0	0.20000
14	1	0.80000
15	0	1.00000
16	0	1.00000
19	0	1.00000
20	0	0.33333
20	1	0.66667
21	0	1.00000
23	0	1.00000
24	0	1.00000
27	0	1.00000
28	0	0.75000
28	1	0.25000
29	0	0.66667
29	1	0.33333
30	0	1.00000
31	0	0.75000
31	1	0.25000
32	0	0.83333
32	1	0.16667
33	0	0.91667
33	1	0.08333
2	0	0.11111
2	1	0.88889
1	0	0.12500
1	1	0.87500
3	0	0.40000
3	1	0.60000
4	1	1.00000
5	1	1.00000
6	1	1.00000
7	1	1.00000
8	1	1.00000
11	1	1.00000
12	1	1.00000
13	1	1.00000
17	1	1.00000
18	1	1.00000
22	1	1.00000
26	0	1.00000
25	0	1.00000

The data set CommLinksOut shows how the communities are interconnected. It is shown in Output 1.7.5. In this example, when the resolution value is 1, the link weight between Communities 0 and 1 is 7, and the link weight between Communities 1 and 2 is 4.

Output 1.7.5 Community Links Output

level	resolution	from_ community	to_community	link_ weight
1	1.0	0	1	7
1	1.0	0	2	7
1	1.0	1	2	3
1	1.0	1	3	4
2	0.5	0	1	10

Example 1.8: Recursive Community Detection on Zachary's Karate Club Data

This example illustrates the use of the RECURSIVE option in community detection on Zachary's Karate Club data. The data set appears in Example 1.7. This example forces each community to contain no more than five nodes and the number of links between any pair of nodes within any community to be no greater than 2.

```
proc optgraph
   data_links            = LinkSetIn
   out_nodes             = NodeSetOut
   graph_internal_format = thin;
   community
      resolution_list    = 1.0
      recursive (max_comm_size = 5 max_diameter = 2 relation = AND)
      out_community      = CommOut;
run;
```

The data set NodeSetOut contains the community identifier of each node. It is shown in Output 1.8.1.

Output 1.8.1 Community Nodes Output

node	community_ 1
0	3
9	1
10	7
14	7
15	3
16	3
19	3
20	4
21	3
23	3
24	8
27	2
28	8
29	9
30	2
31	1
32	9
33	3
2	4
1	5
3	7
4	6
5	0
6	0
7	0
8	6
11	0
12	5
13	6
17	0
18	5
22	4
26	8
25	8

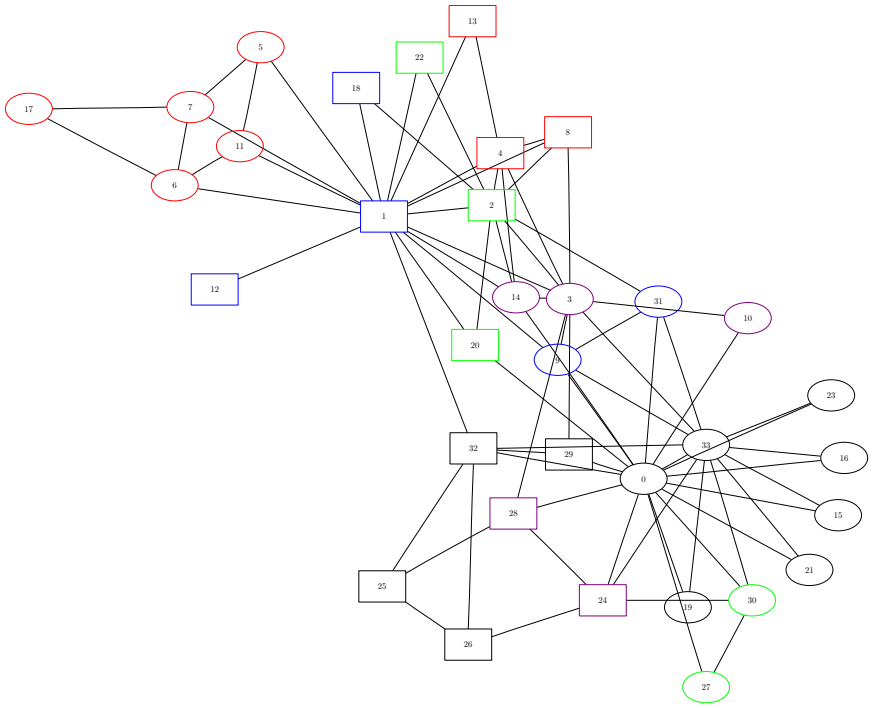
The data set CommOut contains the number of nodes contained in each community. It is shown in Output 1.8.2.

Output 1.8.2 Community Number of Nodes Output

level	resolution	community	nodes
1	1	0	5
1	1	1	2
1	1	2	2
1	1	3	7
1	1	4	3
1	1	5	3
1	1	6	3
1	1	7	3
1	1	8	4
1	1	9	2

The community graph is shown in Figure 1.131, with different node shapes and colors representing different communities.

Figure 1.131 Karate Club Recursive Communities

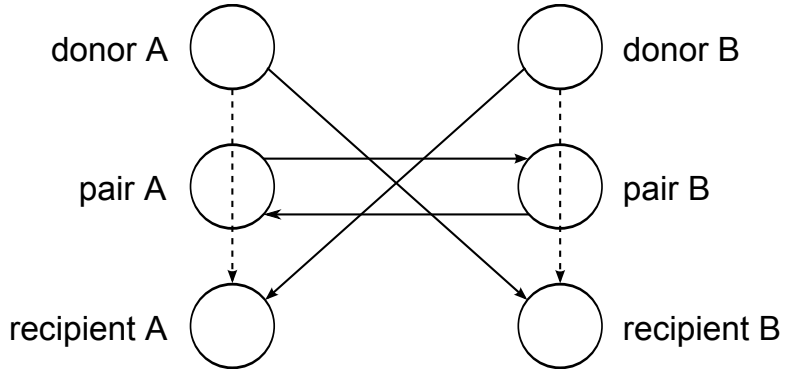


As you can see from Output 1.8.2, Community 3, whose nodes are drawn as black ellipses in Figure 1.131, contains seven nodes even though the maximum number of nodes in any community is set to be 5. This is because Community 3 has a symmetric shape: Nodes 0 and 33 are in the center, and they symmetrically connect to Nodes 21, 15, 19, 16, and 23. Therefore, this community cannot be further split.

Example 1.9: 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 1.132. More generally, an n-way swap can be performed involving n donors and n recipients (Willingham 2009).

Figure 1.132 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 with weight 0, you can also include altruistic donors with no recipients, or recipients without 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 118. 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 of at most L length and then solve a set packing problem. You can use PROC OPTGRAPH to generate the cycles and then PROC OPTMODEL (see *SAS/OR User's Guide: Mathematical Programming*) to read the PROC OPTGRAPH 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 OPTGRAPH to generate all cycles with length 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 optgraph
   loglevel        = moderate
   graph_direction = directed
   data_links      = LinkSetIn;
   cycle
      minLength    = 2
      maxLength    = &max_length
      out          = Cycles
      mode         = all_cycles;
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_CYCLE_;
```

PROC OPTGRAPH finds 224 cycles of the appropriate length, as shown in Output 1.9.1.

Output 1.9.1 Cycles for Kidney Donor Exchange PROC OPTGRAPH Log

```
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Running OPTGRAPH version 12.3.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: The OPTGRAPH 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 0.02 (cpu: 0.02) seconds.
NOTE: Building the input graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The input graph storage is using 0.0 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 statement.
NOTE: The graph has 224 cycles.
NOTE: Processing cycles used 6.04 (cpu: 6.04) seconds.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Creating cycle data set output.
NOTE: Data output used 0.00 (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=6.04  REAL_TIME=6.04
```

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:

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

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 1.9.2, it was able to find a total weight (quality level) of 26.02.

Output 1.9.2 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 MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 35 constraints.
NOTE: The MILP presolver removed 518 constraint coefficients.
NOTE: The MILP presolver modified 116 constraint coefficients.
NOTE: The presolved problem has 224 variables, 28 constraints, and 1382
      constraint coefficients.
NOTE: The MILP solver is called.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       3     22.7780692   1080.2049611   97.89%       0
             0       1       3     22.7780692     26.5638757   14.25%       0
             0       1       4     23.2747070     26.0203249   10.55%       0
             0       1       4     23.2747070     26.0203023   10.55%       0
             0       1       4     23.2747070     26.0202987   10.55%       0
             0       1       6     26.0202871     26.0202871    0.00%       0
NOTE: The MILP solver added 5 cuts with 599 cut coefficients at the root.
NOTE: Optimal.
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=0 BOUND_INFEASIBILITY=0
INTEGER_INFEASIBILITY=0 BEST_BOUND=26.020287142 NODES=1 ITERATIONS=110
PRESOLVE_TIME=0.05 SOLUTION_TIME=0.14
```

The data set Solution, shown in Output 1.9.3, now contains the cycles that define the best exchange and their associated weight (quality).

Output 1.9.3 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 1.10: 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 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 optgraph
   data_matrix = RelayTimesF;
   linear_assignment
      out      = LinearAssignF
      id       = (name sex);
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_LAP_;

proc optgraph
   data_matrix = RelayTimesM;
   linear_assignment
      out      = LinearAssignM
      id       = (name sex);
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_LAP_;
```

The progress of the two PROC OPTGRAPH calls is shown in Output 1.10.1 and Output 1.10.2.

Output 1.10.1 PROC OPTGRAPH Log: Linear Assignment for Female Swim Times

```
NOTE: ------------------------------------------------------------------------
NOTE: Running OPTGRAPH version 12.3.
NOTE: ------------------------------------------------------------------------
NOTE: The OPTGRAPH 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: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: ------------------------------------------------------------------------
NOTE: Processing LINEAR_ASSIGNMENT statement.
NOTE: The minimum cost linear assignment is 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 1.10.2 PROC OPTGRAPH Log: Linear Assignment for Male Swim Times

```
NOTE: ------------------------------------------------------------------------
NOTE: Running OPTGRAPH version 12.3.
NOTE: ------------------------------------------------------------------------
NOTE: The OPTGRAPH 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: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: ------------------------------------------------------------------------
NOTE: Processing LINEAR_ASSIGNMENT statement.
NOTE: The minimum cost linear assignment is 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 1.10.3 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 1.10.4 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

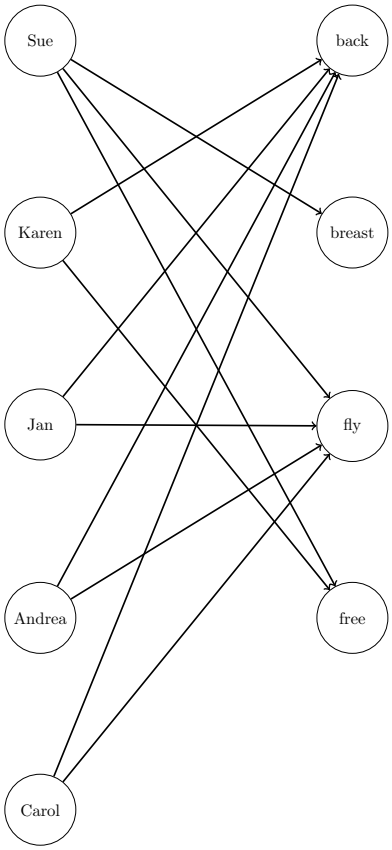
Example 1.11: Linear Assignment Problem, Sparse Format versus Dense Format

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 1.133.

Figure 1.133 Bipartite Graph for Linear Assignment Problem



Because of this, you can represent the same data in RelayTimesMatrix with 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 OPTGRAPH returns an error.

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

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

proc optgraph
   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 1.11.1 and Output 1.11.2.

Output 1.11.1 Optimal Assignments for Swim Times (Dense Input)

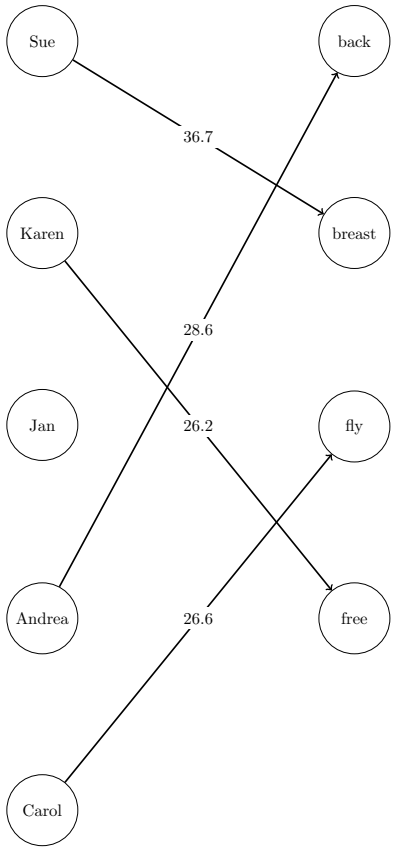
name	sex	assign	cost
Sue	F	breast	36.7
Karen	F	free	26.2
Andrea	F	back	28.6
Carol	F	fly	26.6
			=====
			118.1

Output 1.11.2 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 1.134.

Figure 1.134 Optimal Assignments for Swim Times

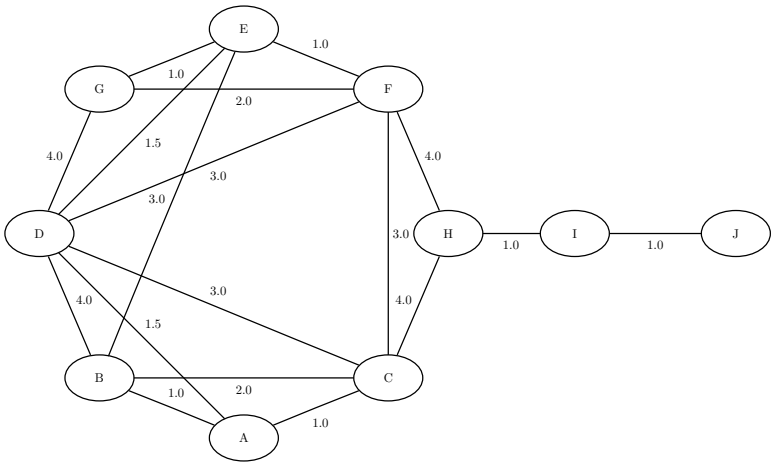


For large problems where a number of links are forbidden, the sparse format can be faster and can save a great deal of memory. Consider an example that uses the format of the DATA_MATRIX= option with 15,000 columns ($|S| = 15,000$) and 4,000 rows ($|T| = 4,000$). To store the dense matrix in memory, PROC OPTGRAPH needs to allocate approximately $|S| \cdot |T| \cdot 8/1024/1024 = 457$ 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 $|S| \cdot |T| \cdot 0.05 \cdot 12/1024/1024 = 34$ 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 1.12: Minimum Spanning Tree for Computer Network Topology

Consider again the small network of computers described in the section "Example 1.3: Betweenness and Closeness Centrality for Computer Network Topology" on page 187. Suppose that this network has not yet been formed, but for structural reasons the connections between the machines shown in Figure 1.126 are the only possible links. 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 1.135. 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 1.135 Potential Office Computer Network



Define the link data set as follows:

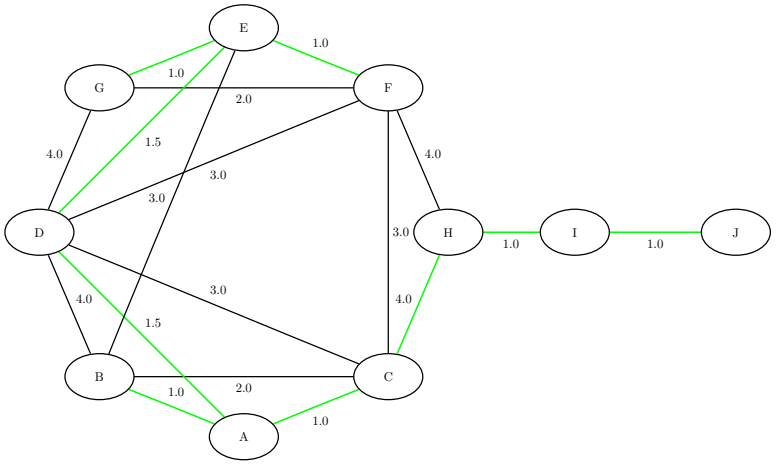
```
data LinkSetInCompNet;
   input from $ to $ weight @@;
   datalines;
A B 1.0  A C 1.0  A D 1.5  B C 2.0  B D 4.0
B E 3.0  C D 3.0  C F 3.0  C H 4.0  D E 1.5
D F 3.0  D G 4.0  E F 1.0  E G 1.0  F G 2.0
F H 4.0  H I 1.0  I J 1.0
;
```

The following statements find a minimum spanning tree:

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

Output 1.12.1 shows the resulting data set MinSpanTree, which is displayed graphically in Figure 1.136 with the minimal cost links shown in green.

Figure 1.136 Minimum Spanning Tree for Office Computer Network



Output 1.12.1 Minimum Spanning Tree of a Computer Network Topology

```
from    to    weight

 H      I       1.0
 E      G       1.0
 E      F       1.0
 A      B       1.0
 A      C       1.0
 I      J       1.0
 D      E       1.5
 A      D       1.5
 C      H       4.0
                ======
                 13.0
```

Example 1.13: 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 as 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 109. 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 those links in G^T that are not in G.

```
proc optgraph
   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 &_OPTGRAPH_;
%put &_OPTGRAPH_CYCLE_;
%put &_OPTGRAPH_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 1.13.1.

Output 1.13.1 PROC OPTGRAPH Log: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System

```
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Running OPTGRAPH version 12.3.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: The OPTGRAPH 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 0.01 (cpu: 0.02) seconds.
NOTE: Building the input graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The input graph storage is using 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 statement.
NOTE: The graph has 1 cycle.
NOTE: Processing cycles used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Processing TRANSITIVE_CLOSURE statement.
NOTE: Processing the transitive closure used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Creating transitive closure data set output.
NOTE: Creating cycle 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  TRANSITIVE_CLOSURE=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 1.13.2 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 1.13.3 Chains in Bug Tracking System

defectId	linked Defect
S0152674	S0152674
S0152674	S0153307
S0153280	S0152674
S0153280	S0153280
S0153307	S0153280
S0153307	S0153307
S0162973	S0165405
S0325026	S0643230

Example 1.14: Reach Networks for Computation of Market Coverage of a Terrorist Network

The problem of finding an efficient method for *covering* a market (a set of entities) is important in numerous industries. For example, consider that you are an advertising company with access to data that are collected from your customers' social networks. To keep costs at a minimum in some new promotion, you want to find a minimal set of customers to whom you need to advertise in order to reach the entire market. To solve this, you could first generate all the reach networks for each customer using PROC OPTGRAPH. These networks can then be used in a *set-covering problem*, which can be solved as an integer linear program using PROC OPTMODEL. Let N be the set of customers that you want to reach, and let the links A define the social network between those customers. If you use a one-hop reach network, you assume that if an advertisement is sent to customer i, then customer i will promote the advertisement to all his friends (those he is connected to in A). If you use two-hop reach networks, you assume that customer i's friends will also promote to their friends. So the question is: to which subset of customers should you advertise to reach all customers through the promotion mechanism?

This problem can be generalized as follows:

> Given a graph $G = (N, A)$, choose a node set N^* of minimal size such that there is a path of length less than or equal to L to every node in N from a node in N^*.

To illustrate an application of this problem, consider again the terrorist communications network from the section "Example 1.1: Articulation Points in a Terrorist Network" on page 180. In this case, customers are (alleged) terrorists. Solving the covering problem here can tell you a subset of people to focus on in an investigation in order to cover all members of the network.

The following macro **%GenerateReach** runs PROC OPTGRAPH to generate the reach network for each person in the terrorist network for a variable hop limit:

```
%macro GenerateReach(limit=);
proc optgraph
   out_nodes      = NodeSetOut
   data_links     = LinkSetInTerror911;
   reach
      each_source
      out_nodes   = ReachNode
      maxreach    = &limit;
run;
%mend GenerateReach;
```

The following macro **%SolverCover** runs PROC OPTMODEL to solve the set-covering problem:

```
%macro SolverCover();
proc optmodel;
   string      tmpLabel;
   set<num>    NODE_ID;
   set<string> NODE_LABEL init {};
   string      nodeIdToLabel{NODE_ID};
   num         nodeLabelToId{NODE_LABEL};

   set<num> REACH_SET{NODE_ID} init {};
   set<string,num> PAIRS;

   /* read data */
   read data NodeSetOut into NODE_ID=[_n_] nodeIdToLabel=node;
   read data ReachNode into PAIRS=[node reach];
   for{i in NODE_ID} do;
      tmpLabel   = nodeIdToLabel[i];
      NODE_LABEL = NODE_LABEL union {tmpLabel};
      nodeLabelToId[tmpLabel] = i;
   end;
   for{<label,i> in PAIRS} do;
      REACH_SET[i] = REACH_SET[i] union {nodeLabelToId[label]};
   end;

   /* declare decision variables */
   var x {NODE_ID} binary;

   /* declare objective */
   minimize numNodes = sum{j in NODE_ID} x[j];

   /* cover constraint */
   con cover {i in NODE_ID}:
      sum{j in REACH_SET[i]} x[j] >= 1;
```

```
   /* solve */
   solve;

   create data Solution from [label]=
      (setof{j in NODE_ID : round(x[j].sol)=1}nodeIdToLabel[j]);
quit;
%mend SolverCover;
```

The following statements calculate the minimal cover for the one-hop limit:

```
%GenerateReach(limit=1);
%SolverCover();
```

In order to cover the network, assuming a one-hop limit, the investigators would need to investigate the people listed in the data set Solution, shown in Output 1.14.1.

Output 1.14.1 Minimal One-Hop Cover for Terrorist Communications Network

Obs	label
1	Djamal_Beghal
2	Zacarias_Moussaoui
3	Essid_Sami_Ben_Khemais
4	Mohamed_Atta
5	Agus_Budiman
6	Mamduh_Mahmud_Salim
7	Fayez_Ahmed
8	Satam_Suqami
9	Nawaf_Alhazmi
10	Hani_Hanjour

The following statements calculate the minimal cover for the two-hop limit:

```
%GenerateReach(limit=2);
%SolverCover();
```

If investigators assume a two-hop limit, they could focus their attention to the two people shown in Output 1.14.2. Then by following their links (and their links' links) they could cover the entire network.

Output 1.14.2 Minimal Two-Hop Cover for Terrorist Communications Network

Obs	label
1	Jerome_Courtaillier
2	Mohamed_Atta

Example 1.15: 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 164.

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 OPTGRAPH statements find the optimal tour through each of the capital cities:

```
/* Find optimal tour using OPTGRAPH */
proc optgraph
   loglevel   = moderate
   data_links = CitiesDist
   out_nodes  = TSPTourNodes;
   data_links_var
      from    = city1
      to      = city2
      weight  = distance;
   tsp
      out     = TSPTourLinks;
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_TSP_;
```

The progress of the procedure is shown in Output 1.15.1. The total mileage needed to optimally traverse the capital cities is 10,627.75 miles.

Output 1.15.1 PROC OPTGRAPH Log: Traveling Salesman Tour through US Capital Cities

```
NOTE: ----------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------
NOTE: Running OPTGRAPH version 12.3.
NOTE: ----------------------------------------------------------------------------
NOTE: ----------------------------------------------------------------------------
NOTE: The OPTGRAPH 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 0.01 (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 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 TSP statement.
NOTE: The initial TSP heuristics found a tour with cost 10645.918753 using 0.22
      (cpu: 0.16) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
          Node  Active    Sols    BestInteger      BestBound       Gap     Time
             0       1       1  10645.9187534  10040.5139714    6.03%        0
             0       1       1  10645.9187534  10241.6970024    3.95%        0
             0       1       1  10645.9187534  10262.9074205    3.73%        0
             0       1       1  10645.9187534  10293.2995080    3.43%        0
             0       1       1  10645.9187534  10350.0790852    2.86%        0
             0       1       1  10645.9187534  10549.5506188    0.91%        0
             0       1       1  10645.9187534  10576.0823291    0.66%        0
             0       1       1  10645.9187534  10590.3709358    0.52%        0
             0       1       1  10645.9187534  10590.8162090    0.52%        0
             0       1       1  10645.9187534  10590.9748294    0.52%        0
             0       1       1  10645.9187534  10607.8528157    0.36%        0
             0       1       6  10645.9187534  10607.8528157    0.36%        0
NOTE: The MILP solver added 16 cuts with 4213 cut coefficients at the root.
             1       1       7  10627.7543183  10607.8528157    0.19%        0
             2       0       7  10627.7543183  10627.7543183    0.00%        0
NOTE: Optimal.
NOTE: Objective = 10627.754318.
NOTE: Processing the traveling salesman problem used 0.35 (cpu: 0.28) 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.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=10627.754318  RELATIVE_GAP=0  ABSOLUTE_GAP=0
PRIMAL_INFEASIBILITY=0  BOUND_INFEASIBILITY=0  INTEGER_INFEASIBILITY=0
BEST_BOUND=10627.754318  NODES=3  ITERATIONS=169  CPU_TIME=0.28  REAL_TIME=0.35
```

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;
```

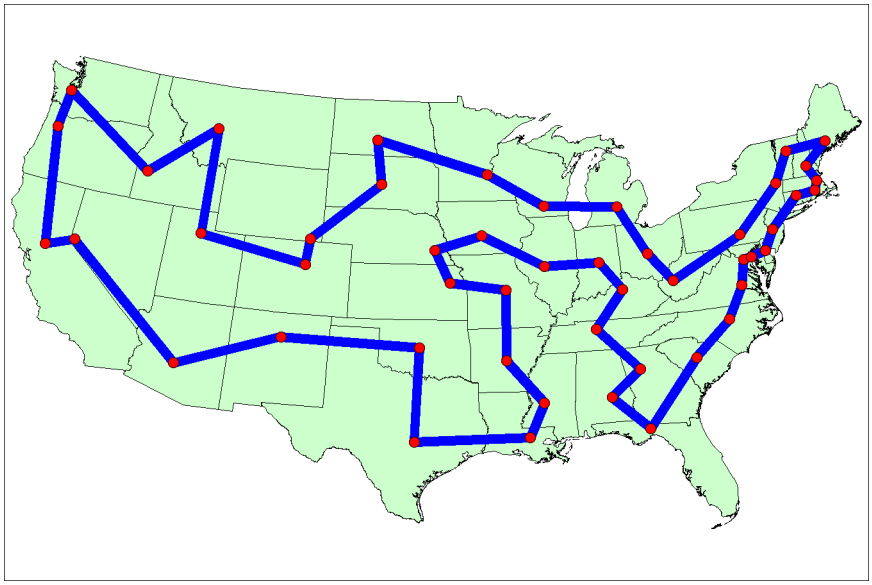
```
/* Get a list of the endpoints locations */
proc sql;
   create table anno_dots as
   select unique x, y from anno_path;
quit;

/* 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 1.15.2.

Output 1.15.2 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 1.137.

Figure 1.137 Links in the Optimal Traveling Salesman Tour

City Name	City Name	distance
Montgomery	Tallahassee	177.14
Tallahassee	Columbia	311.23
Columbia	Raleigh	182.99
Raleigh	Richmond	135.58
Richmond	Washington	97.96
Washington	Annapolis	27.89
Annapolis	Dover	54.01
Dover	Trenton	83.88
Trenton	Hartford	151.65
Hartford	Providence	65.56
Providence	Boston	38.41
Boston	Concord	66.30
Concord	Augusta	117.36
Augusta	Montpelier	139.32
Montpelier	Albany	126.19
Albany	Harrisburg	230.24
Harrisburg	Charleston	287.34
Charleston	Columbus	134.64
Columbus	Lansing	205.08
Lansing	Madison	246.88
Madison	Saint Paul	226.25
Saint Paul	Bismarck	391.25
Bismarck	Pierre	170.27
Pierre	Cheyenne	317.90
Cheyenne	Denver	98.33
Denver	Salt Lake City	373.05
Salt Lake City	Helena	403.40
Helena	Boise City	291.20
Boise City	Olympia	401.31
Olympia	Salem	146.00
Salem	Sacramento	447.40
Sacramento	Carson City	101.51
Carson City	Phoenix	577.84
Phoenix	Santa Fe	378.27
Santa Fe	Oklahoma City	474.92
Oklahoma City	Austin	357.38
Austin	Baton Rouge	394.78
Baton Rouge	Jackson	139.75
Jackson	Little Rock	206.87
Little Rock	Jefferson City	264.75
Jefferson City	Topeka	191.67
Topeka	Lincoln	132.94
Lincoln	Des Moines	168.10
Des Moines	Springfield	243.02
Springfield	Indianapolis	186.46
Indianapolis	Frankfort	129.90
Frankfort	Nashville-Davidson	175.58
Nashville-Davidson	Atlanta	212.61
Atlanta	Montgomery	145.39
		==========
		10,627.75

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 Series in Applied Mathematics, Princeton, NJ: Princeton University Press.

Batagelj, V. and Zaversnik, M. (2003), "An O(m) Algorithm for Cores Decomposition of Networks," *Computing Research Repository*, cs.DS/0310049.

Blondel, V. D., Guillaume, J. L., Lambiotte, R., and Lefebvre, E. (2008), "Fast Unfolding of Communities in Large Networks," *Journal of Statistical Mechanics: Theory and Experiment*, 10, 10000–10014.

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

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

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

Fowler, J. H. and Joen, S. (2008), "The Authority of Supreme Court Precedent," *Social Networks*, 30, 16–30.
URL http://jhfowler.ucsd.edu/judicial.htm

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

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.

Kleinberg, J. (1998), "Authoritative Sources in a Hyperlinked Environment," in *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, Philadelphia: Society for Industrial and Applied Mathematics.

Krackhardt, D. (1990), "Assessing the Political Landscape: Structure, Cognition, and Power in Organizations," *Administrative Science Quarterly*, 35, 342–369.

Krebs, V. (2002), "Uncloaking Terrorist Networks," *First Monday*, 7, available at 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.

Lancichinetti, A. and Fortunato, S. (2009), "Community Detection Algorithms: A Comparative Analysis," *Physical Review E*, 80, 56,117–56,128.

Landes, W. M. and Posner, R. A. (1976), "Legal Precedent: A Theoretical and Empirical Analysis," *Journal of Law and Economics*, 19, 249–307.

Mihalcea, R. (2005), "Unsupervised Large-Vocabulary Word Sense Disambiguation with Graph-Based Algorithms for Sequence Data Labeling," in *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*, 411–418, Vancouver.

Newman, M. E. J. (2010), *Networks: An Introduction*, Oxford: Oxford University Press.

Raghavan, U. N., Albert, R., and Kumara, S. (2007), "Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks," *Physical Review E*, 76, 36106–36117.

Ronhovde, P. and Nussinov, Z. (2010), "Local Resolution-Limit-Free Potts Model for Community Detection," *Physical Review E*, 81, 46114–46129.

Sinha, R. and Mihalcea, R. (2007), "Unsupervised Graph-Based Word Sense Disambiguation Using Measures of Word Semantic Similarity," in *Proceedings of the IEEE International Conference on Semantic Computing*, Los Alamitos, CA: IEEE Computer Society Press.

Sleijpen, G. L. G. and van der Vorst, H. A. (2000), "A Jacobi-Davidson Iteration Method for Linear Eigenvalue Problems," *SIAM Review*, 42, 267–293.

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.

Traag, V. A., Van Dooren, P., and Nesterov, Y. (2011), "Narrow Scope for Resolution-Limit-Free Community Detection," *Physical Review E*, 84, 016114 (1–9).
URL http://pre.aps.org/abstract/PRE/v84/i1/e016114

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

Zachary, W. W. (1977), "An Information Flow Model for Conflict and Fission in Small Groups," *Journal of Anthropological Research*, 33, 452–473.

Index

Your Turn

We welcome your feedback.

- If you have comments about this book, please send them to **yourturn@sas.com**. Include the full title and page numbers (if applicable).
- If you have comments about the software, please send them to **suggest@sas.com**.