

SAS/OR[®] 14.1 User's Guide: Mathematical Programming The Network Solver



This document is an individual chapter from *SAS/OR® 14.1 User's Guide: Mathematical Programming*.

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2015. *SAS/OR® 14.1 User's Guide: Mathematical Programming*. Cary, NC: SAS Institute Inc.

SAS/OR® 14.1 User's Guide: Mathematical Programming

Copyright © 2015, SAS Institute Inc., Cary, NC, USA

All Rights Reserved. Produced in the United States of America.

For a hard-copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the Internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

U.S. Government License Rights; Restricted Rights: The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication, or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a), and DFAR 227.7202-4, and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government's rights in Software and documentation shall be only those set forth in this Agreement.

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

July 2015

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

Chapter 9

The Network Solver

Contents

Overview: Network Solver	376
Getting Started: Network Solver	376
Syntax: Network Solver	381
Functional Summary	382
SOLVE WITH NETWORK Statement	386
General Options	386
Input and Output Options	387
Algorithm Options	390
Details: Network Solver	398
Input Data for the Network Solver	398
Solving over Subsets of Nodes and Links (Filters)	402
Numeric Limitations	406
Biconnected Components and Articulation Points	407
Clique	410
Connected Components	413
Cycle	418
Linear Assignment (Matching)	423
Minimum-Cost Network Flow	424
Minimum Cut	430
Minimum Spanning Tree	434
Shortest Path	436
Transitive Closure	449
Traveling Salesman Problem	452
Macro Variable _OROPTMODEL_	458
Examples: Network Solver	462
Example 9.1: Articulation Points in a Terrorist Network	462
Example 9.2: Cycle Detection for Kidney Donor Exchange	464
Example 9.3: Linear Assignment Problem for Minimizing Swim Times	468
Example 9.4: Linear Assignment Problem, Sparse Format versus Dense Format	471
Example 9.5: Minimum Spanning Tree for Computer Network Topology	474
Example 9.6: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System	476
Example 9.7: Traveling Salesman Tour through US Capital Cities	479
References	485

Overview: Network Solver

The network solver includes a number of graph theory, combinatorial optimization, and network analysis algorithms. The algorithm classes are listed in [Table 9.1](#).

Table 9.1 Algorithm Classes in the Network solver

Algorithm Class	SOLVE WITH NETWORK Option
Biconnected components	BICONCOMP
Maximal cliques	CLIQUE=
Connected components	CONCOMP
Cycle detection	CYCLE=
Linear assignment (matching)	LINEAR_ASSIGNMENT
Minimum-cost network flow	MINCOSTFLOW
Minimum cut	MINCUT=
Minimum spanning tree	MINSPANTREE
Shortest path	SHORTPATH=
Transitive closure	TRANSITIVE_CLOSURE
Traveling salesman	TSP=

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

Unlike other solvers that PROC OPTMODEL uses, the network solver operates directly on arrays and sets. You do not need to explicitly define variables, constraints, and objectives to use the network solver. PROC OPTMODEL declares the appropriate objects internally as needed. You specify the names of arrays and sets that define your inputs and outputs as options in the SOLVE WITH NETWORK statement.

Getting Started: Network Solver

This section shows an introductory example for getting started with the network solver. For more information about the expected input formats and the various algorithms available, see the sections “[Details: Network Solver](#)” on page 398 and “[Examples: Network Solver](#)” on page 462.

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

In this road network (graph), the links are the roads and the nodes are intersections between roads. For each road, you assign a *link attribute* in the parameter `time_to_travel` to describe the number of minutes that it takes to drive from one node to another. The following data were collected using Google Maps (Google 2011):

```

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
;

```

Using the network solver, you want to find the route that yields the shortest path between home (614 Capital Blvd) and the SAS headquarters (SAS Campus Drive). This can be done by using the SHORTPATH= option as follows:

```

proc optmodel;
  set<str,str> LINKS;
  num miles{LINKS};
  num miles_per_hour{LINKS};
  num time_to_travel{<i,j> in LINKS} = miles[i,j] / miles_per_hour[i,j] * 60;
  read data LinkSetInRoadNC10am into
    LINKS=[start_inter end_inter]
    miles miles_per_hour
  ;
  /* You can compute paths between many pairs of source and destination,
     so these parameters are declared as sets */
  set HOME = /"614CapitalBlvd"/;
  set WORK = /"SASCampusDrive"/;

  /* The path is stored as a set of: Start, End, Sequence, Tail, Head */
  set<str,str,num,str,str> PATH;

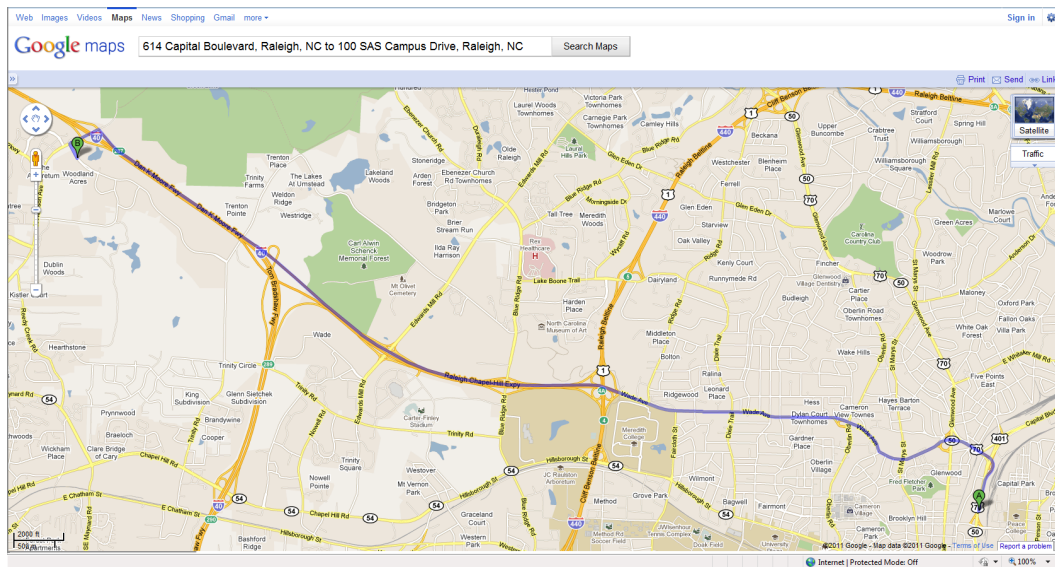
  solve with network /
    links      = ( weight = time_to_travel )
    shortpath = ( source = HOME
                  sink   = WORK )
    out        = ( sppaths = PATH )
  ;
  create data ShortPath from [s t order start_inter end_inter]=PATH
    time_to_travel[start_inter,end_inter];
quit;

```

For more information about shortest path algorithms in the network solver, see the section “[Shortest Path](#)” on page 436. [Figure 9.1](#) displays the output data set ShortPath, which shows the best route to take to minimize travel time at 10:00 a.m. This route is also shown in Google Maps in [Figure 9.2](#).

Figure 9.1 Shortest Path for Road Network at 10:00 A.M.

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

Figure 9.2 Shortest Path for Road Network at 10:00 A.M. in Google Maps

Now suppose that it is rush hour (5:00 p.m.) and the time to traverse the roads has changed because of traffic patterns. You want to find the route that is the shortest path for going home from SAS headquarters under different speed assumptions due to traffic.

The following statements are similar to the first network solver run, except that one *miles_per_hour* value is modified and the SOURCE= and SINK= option values are reversed:

```

proc optmodel;
  set<str,str> LINKS;
  num miles{LINKS};
  num miles_per_hour{LINKS};
  num time_to_travel{<i,j> in LINKS} = miles[i,j] / miles_per_hour[i,j] * 60;
  read data LinkSetInRoadNC10am into
    LINKS=[start_inter end_inter]
    miles miles_per_hour
  ;
  /* high traffic */
  miles_per_hour['Capital/WadeAve', 'WadeAve/RaleighExpy'] = 25;

  /* You can compute paths between many pairs of source and destination,
     so these parameters are declared as sets */
  set HOME = /"614CapitalBlvd"/;
  set WORK = /"SASCampusDrive"/;

  /* The path is stored as a set of: Start, End, Sequence, Tail, Head */
  set<str,str,num,str,str> PATH;

  solve with network /
    links      = ( weight = time_to_travel )
    shortpath = ( source = WORK
                  sink   = HOME )
    out        = ( sppaths = PATH )
  ;
  create data ShortPath from [s t order start_inter end_inter]=PATH
    time_to_travel[start_inter,end_inter];
quit;

```

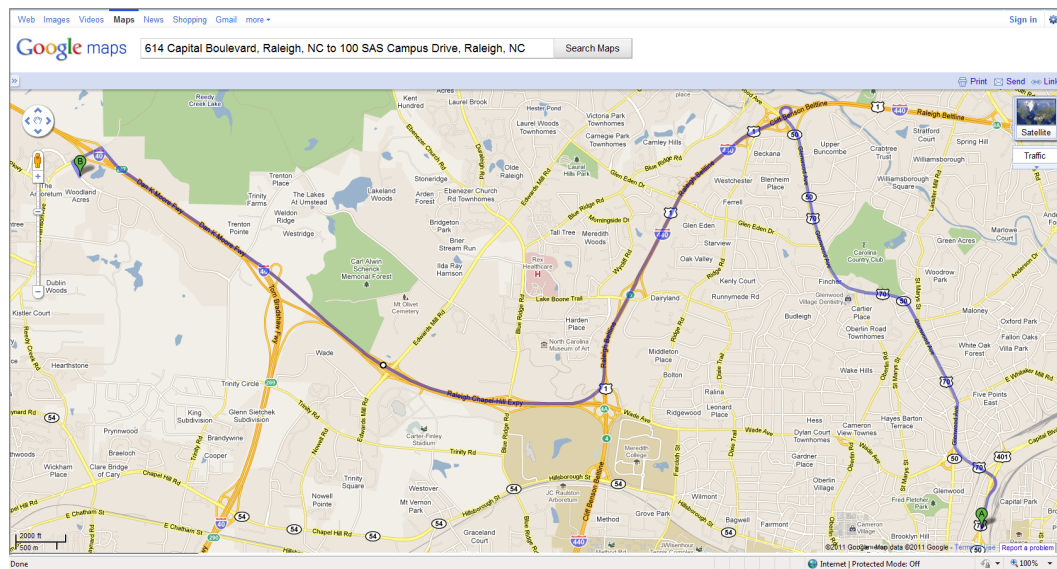
Now, the output data set ShortPath, shown in [Figure 9.3](#), shows the best route for going home at 5:00 p.m. Because the traffic on Wade Avenue is usually heavy at this time of day, the route home is different from the route to work.

Figure 9.3 Shortest Path for Road Network at 5:00 P.M.

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

This new route is shown in Google Maps in Figure 9.4.

Figure 9.4 Shortest Path for Road Network at 5:00 P.M. in Google Maps



Syntax: Network Solver

SOLVE WITH NETWORK /

General and Diagnostic Options:

```
< GRAPH_DIRECTION=DIRECTED | UNDIRECTED >
< INCLUDE_SELFLINK >
< LOGFREQ=number >
< LOGLEVEL=number | string >
< MAXTIME=number >
< TIMETYPE=number | string >
```

Data Input and Output Options:

```
< LINKS=( suboptions ) >
< NODES=( suboptions ) >
< OUT=( suboptions ) >
< SUBGRAPH=( suboptions ) >
```

Algorithm Options:

```
< BICONCOMP<=( ) > >
< CLIQUE<=( suboption ) > >
< CONCOMP<=( suboption ) > >
< CYCLE<=( suboptions ) > >
< LINEAR_ASSIGNMENT<=( ) > >
< MINCOSTFLOW<=( ) > >
< MINCUT<=( suboptions ) > >
< MINSPANTREE<=( ) > >
< SHORTPATH<=( suboptions ) > >
< TRANSITIVE_CLOSURE<=( ) > >
< TSP<=( suboptions ) > > ;
```

There are three types of SOLVE WITH NETWORK statement options:

- **General and diagnostic** options have the same meaning for multiple algorithms.
- **Data input and output** options, such as the LINKS=, NODES=, and OUT= options, control the names of the sets and variables that the network solver uses to build the graph and that the algorithms use for output.
- **Algorithm** options select an algorithm to run, and where available, provide further algorithm-specific configuration directives.

The section “**Functional Summary**” on page 382 provides a quick reference for each of the suboptions for each option. Each option is then described in more detail in its own section, in alphabetical order.

Functional Summary

Table 9.2 summarizes the options and suboptions available in the SOLVE WITH NETWORK statement.

Table 9.2 Functional Summary of SOLVE WITH NETWORK
Options

Description	Option Suboption
General Options	
Specifies directed or undirected graphs	GRAPH_DIRECTION=
Includes self links in the graph definition	INCLUDE_SELFLINK=
Specifies the iteration log frequency	LOGFREQ=
Controls the amount of information that is displayed in the SAS log	LOGLEVEL=
Specifies the maximum time spent calculating results	MAXTIME=
Specifies whether time units are in CPU time or real time	TIMETYPE=
Input and Output Options	
Groups link-indexed data	LINKS=()
Names a set of links to include in the graph definition even if no weights or bounds are available for them	INCLUDE=
Specifies the flow lower bound for each link	LOWER=
Specifies the flow upper bound for each link	UPPER=
Specifies link weights	WEIGHT=
Groups node-indexed data	NODES=()
Names a set of nodes to include in the graph definition even if no weights are available for them	INCLUDE=
Specifies node weights	WEIGHT=
Specifies node supply upper bounds in the minimum-cost network flow problem	WEIGHT2=
Specifies the input sets that enable you to solve a problem over a subgraph	SUBGRAPH=()
Specifies the subset of links to use	LINKS=
Specifies the subset of nodes to use	NODES=
Specifies the output sets or arrays for each algorithm (see Table 9.4 for which OUT= suboptions you can specify for each algorithm)	OUT=()
Specifies the output set for articulation points	ARTPOINTS=
Specifies the output set for linear assignment	ASSIGNMENTS=
Specifies the array to contain the biconnected component of each link	BICONCOMP=
Specifies the output set for cliques	CLIQUE=
Specifies the output array for connected components	CONCOMP=
Specifies the output set for the cut-sets for minimum cuts	CUTSETS=
Specifies the output set for cycles	CYCLES=
Specifies the output array for the flow on each link	FLOW=
Specifies the output set for the minimum spanning tree (forest)	FOREST=
Specifies the output set for the links that remain after the SUBGRAPH= option is applied	LINKS=

Table 9.2 (continued)

Description	Option Suboption
Specifies the output set for the nodes that remain after the SUBGRAPH= option is applied	NODES=
Specifies the output array for the node order in the traveling salesman problem	ORDER=
Specifies the output set for the partitions for minimum cuts	PARTITIONS=
Specifies the set to contain the link sequence for each path	SPPATHS=
Specifies the numeric array to contain the path weight for each source and sink node pair	SPWEIGHTS=
Specifies the output set for the tour in the traveling salesman problem	TOUR=
Specifies the set to contain the pairs (u, v) of nodes where v is reachable from u	TRANSCL=
Algorithm Options and Suboptions	
Finds biconnected components and articulation points of an undirected input graph	BICONCOMP
Finds maximal cliques in the input graph	CLIQUE=
Specifies the maximum number of cliques to return	MAXCLIQUES=
Finds the connected components of the input graph	CONCOMP=
Specifies the algorithm to use for calculating connected components	ALGORITHM=
Finds the cycles (or the existence of a cycle) in the input graph	CYCLE=
Specifies the maximum number of cycles to return	MAXCYCLES=
Specifies the maximum link count for the cycles to return	MAXLENGTH=
Specifies the maximum link weight for the cycles to return	MAXLINKWEIGHT=
Specifies the maximum sum of node weights to allow in a cycle	MAXNODEWEIGHT=
Specifies the minimum link count for the cycles to return	MINLENGTH=
Specifies the minimum link weight for the cycles to return	MINLINKWEIGHT=
Specifies the minimum node weight for the cycles to return	MINNODEWEIGHT=
Specifies whether to stop after finding the first cycle	MODE=
Solves the minimal-cost linear assignment problem	LINEAR_ASSIGNMENT
Solves the minimum-cost network flow problem	MINCOSTFLOW
Finds the minimum link-weighted cut of an input graph	MINCUT=
Specifies the maximum number of cuts to return from the algorithm	MAXNUMCUTS=
Specifies the maximum weight of each cut to return from the algorithm	MAXWEIGHT=
Solves the minimum link-weighted spanning tree problem on an input graph	MINSPANTREE
Calculates shortest paths between sets of nodes on the input graph	SHORTPATH=
Specifies the type of output for shortest paths results	PATHS=
Specifies the set of sink nodes	SINK=
Specifies the set of source nodes	SOURCE=
Specifies whether to use weights in calculating shortest paths	USEWEIGHT=
Calculates the transitive closure of an input graph	TRANSITIVE_CLOSURE

Table 9.2 (continued)

Description	Option Suboption
Solves the traveling salesman problem	TSP=
Requests that the stopping criterion be based on the absolute objective gap	ABSOBJGAP=
Specifies the level of conflict search	CONFLICTSEARCH=
Specifies the cutoff value for branch-and-bound node removal	CUTOFF=
Specifies the level of cutting planes to be generated by the network solver	CUTSTRATEGY=
Emphasizes feasibility or optimality	EMPHASIS=
Specifies the initial and primal heuristics level	HEURISTICS=
Specifies the maximum number of branch-and-bound nodes to be processed	MAXNODES=
Specifies the maximum number of feasible tours to be identified	MAXSOLS=
Specifies whether to use a mixed integer linear programming solver	MILP=
Specifies the branch-and-bound node selection strategy	NODESEL=
Specifies the probing level	PROBE=
Requests that the stopping criterion be 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=
Requests that the stopping criterion be based on the target objective value	TARGET=
Specifies the rule for selecting branching variable	VARSEL=

Table 9.3 lists the valid **GRAPH_DIRECTION=** values for each algorithm option in the SOLVE WITH NETWORK statement.

Table 9.3 Supported Graph Directions by Algorithm

Algorithm	Direction	
	Undirected	Directed
BICONCOMP	x	
CLIQUE	x	
CONCOMP	x	x
CYCLE	x	x
LINEAR_ASSIGNMENT		x
MINCOSTFLOW		x
MINCUT	x	
MINSPANTREE	x	
SHORTPATH	x	x
TRANSITIVE_CLOSURE	x	x
TSP	x	x

Table 9.4 indicates, for each algorithm option in the SOLVE WITH NETWORK statement, which output options you can specify, and what their types can be. The types vary depending on whether nodes are of type STRING or NUMBER.

Table 9.4 Output Suboptions and Types by Algorithm

Algorithm Option OUT= Suboption	OPTMODEL Type
BICONCOMP	
ARTPOINTS=	SET<STRING> or SET<NUMBER>
BICONCOMP=	NUMBER indexed over links (<NUMBER,NUMBER> or <STRING,STRING>)
CLIQUE	
CLIQUE=	SET<NUMBER,NUMBER> or SET<NUMBER,STRING>
CONCOMP	
CONCOMP=	NUMBER indexed over nodes (NUMBER or STRING)
CYCLE	
CYCLES=	SET<NUMBER,NUMBER,NUMBER> or SET<NUMBER,NUMBER,STRING>
LINEAR_ASSIGNMENT	
ASSIGNMENTS=	SET<NUMBER,NUMBER> or SET<STRING,STRING>
MINCOSTFLOW	
FLOW=	NUMBER indexed over links (<NUMBER,NUMBER> or <STRING,STRING>)
MINCUT	
CUTSETS=	SET<NUMBER,NUMBER,NUMBER> or SET<NUMBER,STRING,STRING>
PARTITIONS=	SET<NUMBER,NUMBER> or SET<NUMBER,STRING>
MINSPANTREE	
FOREST=	SET<NUMBER,NUMBER> or SET<STRING,STRING>
SHORTPATH	
SPPATHS=	SET<NUMBER,NUMBER,NUMBER,NUMBER,NUMBER> or SET<STRING,STRING,NUMBER,STRING,STRING>
SPWEIGHTS=	NUMBER indexed over sink and source node pairs (<NUMBER,NUMBER> or <STRING,STRING>)
TRANSITIVE_CLOSURE	
CLOSURE=	SET<NUMBER,NUMBER> or SET<STRING,STRING>
TSP	
ORDER=	NUMBER indexed over nodes (NUMBER or STRING)
TOUR=	SET<NUMBER,NUMBER> or SET<STRING,STRING>

SOLVE WITH NETWORK Statement

SOLVE WITH NETWORK / < options > ;

The SOLVE WITH NETWORK statement invokes the network solver. You can specify the following *options* to define various processing and diagnostic controls, the graph input and output, and the algorithm to run:

General Options

You can specify the following general options, which have the same meaning for multiple algorithms.

GRAPH_DIRECTION=DIRECTED | UNDIRECTED

DIRECTION=DIRECTED | UNDIRECTED

specifies directed or undirected graphs.

Table 9.5 Values for the GRAPH_DIRECTION= Option

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

By default, GRAPH_DIRECTION=UNDIRECTED.

INCLUDE_SELFINK

includes self links in the graph definition—for example, (i, i) —when an input graph is read. By default, when the network solver reads the **LINKS=** data, it removes all self links.

LOGFREQ=number

controls the frequency with which an algorithm reports progress from its underlying solver. This setting is recognized by the **traveling salesman problem** and **minimum-cost flow** algorithms. You can set *number* to 0 to turn off log updates from underlying algorithms.

LOGLEVEL=number | string

controls the amount of information that is displayed in the SAS log. This setting sets the log level for all algorithms. **Table 9.6** describes the valid values for this option.

Table 9.6 Values for LOGLEVEL= Option

<i>number</i>	<i>string</i>	Description
0	NONE	Turns off all procedure-related messages in the SAS log

Table 9.6 (continued)

<i>number</i>	<i>string</i>	Description
1	BASIC	Displays a basic summary of the input, output, and algorithmic processing
2	MODERATE	Displays a summary of the input, output, and algorithmic processing
3	AGGRESSIVE	Displays a detailed summary of the input, output, and algorithmic processing

By default, LOGLEVEL=BASIC.

MAXTIME=number

specifies the maximum time spent calculating results. The type of time (either CPU time or real time) is determined by the value of the TIMETYPE= option. The value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment. The clique, cycle, minimum-cost network flow, and traveling salesman problem algorithms recognize the MAXTIME= option.

TIMETYPE=number | string

specifies whether CPU time or real time is used for the MAXTIME= option for each applicable algorithm. Table 9.7 describes the valid values of the TIMETYPE= option.

Table 9.7 Values for TIMETYPE= Option

<i>number</i>	<i>string</i>	Description
0	CPU	Specifies units of CPU time
1	REAL	Specifies units of real time

By default, TIMETYPE=CPU.

Input and Output Options

The following options enable you to specify the graph to run algorithms on. These options take array and set names. They are known as *identifier expressions* in Chapter 5, “The OPTMODEL Procedure.” Also see Table 9.4 for semantic requirements and the section “Input Data for the Network Solver” on page 398 for use cases.

LINKS=(suboptions)

groups link-indexed data. For more information, see the section “Input Data for the Network Solver” on page 398.

You can specify the following *suboptions*:

INCLUDE=set-name

names a set of links to include in the graph definition even if no weights or bounds are available for them. For more information, see “Example 9.1: Articulation Points in a Terrorist Network” on page 462. The set must be numeric, and it must be indexed over a subset of the links of the graph.

LOWER=*array-name*

specifies the flow lower bound for each link. The array must be numeric, and it must be indexed over a subset of the links of the graph.

UPPER=*array-name*

specifies the flow upper bound for each link. The array must be numeric, and it must be indexed over a subset of the links of the graph.

WEIGHT=*array-name*

specifies link weights. The array must be numeric, and it must be indexed over a subset of the links of the graph. If you specify this suboption, then any link that does not appear in the index set of the WEIGHT= array has weight 0. If you do not specify this suboption, then every link has weight 1.

NODES=(*suboptions*)

groups node-indexed data. For more information, see the section “[Input Data for the Network Solver](#)” on page 398.

You can specify the following *suboptions*:

INCLUDE=*set-name*

names a set of nodes to include in the graph definition even if no weights are available for them. For more information, see the section “[Connected Components](#)” on page 413.

WEIGHT=*array-name*

specifies node weights. The array must be numeric, and it must be indexed over a subset of the nodes of the graph.

WEIGHT2=*array-name*

specifies node supply upper bounds in the minimum-cost network flow problem. The array must be numeric, and it must be indexed over a subset of the nodes of the graph. For more information, see the section “[Minimum-Cost Network Flow](#)” on page 424.

OUT=(*suboptions*)

specifies the output sets or arrays for each algorithm (see [Table 9.4](#) for which OUT= suboptions you can specify for each algorithm). You can use some of these options (even if you do not invoke any algorithm) to see the filtering outcome that is produced by the **SUBGRAPH=** option.

If you do not specify a *suboption* that matches the [algorithm option](#) in the statement, the algorithm runs and only updates the objective.

If you specify a suboption that does not match the [algorithm option](#) in the statement, OPTMODEL issues a warning.

When you declare arrays that are indexed over nodes, over links, or over sets of nodes or links, you must use the same type you used in your node definition.

See the various algorithm sections for examples of the use of these OUT= *suboptions*.

ARTPOINTS=*set-name*

specifies the output set for articulation points. Each element of the set represents a node ID. This suboption is used with the BICONCOMP algorithm option.

ASSIGNMENTS=*set-name*

specifies the output set for linear assignment. This suboption is used with the LINEAR_ASSIGNMENT algorithm option.

BICONCOMP=*array-name*

specifies the array to contain the biconnected component of each link. This suboption is used with the BICONCOMP algorithm option.

CLIQUE=*set-name*

specifies the output set for cliques. Each tuple of the set represents clique ID and node ID. This suboption is used with the CLIQUE algorithm option.

CONCOMP=*array-name*

specifies the output array for connected components. This suboption is used with the CONCOMP algorithm option.

CUTSETS=*set-name*

specifies the output set for the cut-sets for minimum cuts. Each tuple of the set represents the cut ID, the tail node ID, and the head node ID. This suboption is used with the MINCUT algorithm option.

CYCLES=*set-name*

specifies the output set for cycles. Each tuple of the set represents a cycle ID, the order within that cycle, and the node ID. This suboption is used with the CYCLE algorithm option.

FLOW=*array-name*

specifies the output array for the flow on each link. This suboption is used with the MINCOST-FLOW algorithm option.

FOREST=*set-name*

specifies the output set for the minimum spanning tree (forest). This suboption is used with the MINSPANTREE algorithm option.

LINKS=*set-name*

specifies the output set for the links that remain after the **SUBGRAPH=** option is applied. Each tuple of the set represents tail and head nodes, followed by a sequence of numbers which correspond to the attributes you provide in the **LINKS=** suboptions. The length of the tuples must be the number of attributes you specify plus two (for the tail and head node information). The options you specify in the **LINKS=** option will appear in the output in the order: **WEIGHT=**, **LOWER=**, and **UPPER=**. For an example, see Figure 9.12 in “Solving over Subsets of Nodes and Links (Filters)” on page 402.

NODES=*set-name*

specifies the output set for the nodes that remain after the **SUBGRAPH=** option is applied. Each tuple of the set represents a node, followed by a sequence of numbers which correspond to the attributes you provide in the **NODES=** suboptions. The length of the tuples must be the number of attributes you specify plus one (for node information). The options you specify in the **NODES=** option will appear in the output in the order: **WEIGHT=** and **WEIGHT2=**. For an example, see the section “Minimum-Cost Network Flow with Flexible Supply and Demand” on page 428.

ORDER=*array-name*

specifies the numeric array to contain the position of each node within the optimal tour. This suboption is used with the TSP algorithm option.

PARTITIONS=*set-name*

specifies the output set for the partitions for minimum cuts. The set contains, for each partition, the node IDs in the smaller of the two subsets. Each tuple of the set represents a cut ID and a node ID. This suboption is used with the MINCUT algorithm option.

SPPATHS=*set-name*

specifies the set to contain the link sequence for each path. Each tuple of the set represents a source node ID, a sink node ID, a sequence number, a tail node ID, and a head node ID. This suboption is used with the SHORTPATH algorithm option.

SPWEIGHTS=*array-name*

specifies the numeric array to contain the path weight for each source and sink node pair. This suboption is used with the SHORTPATH algorithm option.

TOUR=*set-name*

specifies the output set for the tour in the traveling salesman problem. This suboption is used with the TSP algorithm option.

TRANSCL=*set-name*

specifies the set to contain the pairs (u, v) of nodes where v is reachable from u . This suboption is used with the TRANSITIVE_CLOSURE algorithm option.

SUBGRAPH=(*suboptions*)

specifies the input sets that enable you to solve a problem over a subgraph. For more information, see the section “[Input Data for the Network Solver](#)” on page 398.

You can specify the following *suboptions*:

LINKS=*set-name*

specifies the subset of links to use. If you specify a node pair that is not referenced in any of the suboptions of the **LINKS=** option, then the network solver returns an error.

NODES=*set-name*

specifies the subset of nodes to use. If you specify a node that is not referenced in any of the suboptions of the **LINKS=** option or the **NODES=** option, then the network solver returns an error.

Algorithm Options

BICONCOMP<=*()*>

finds biconnected components and articulation points of an undirected input graph. For more information, see the section “[Biconnected Components and Articulation Points](#)” on page 407.

CLIQUE<=(*suboption*) >

finds maximal cliques in the input graph. For more information, see the section “[Clique](#)” on page 410.

You can specify the following *suboption*:

MAXCLIQUES=*number*

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

CONCOMP<=(*suboption*) >

finds the connected components of the input graph. For more information, see the section “[Connected Components](#)” on page 413.

You can specify the following *suboption*:

ALGORITHM=DFS | UNION_FIND

specifies the algorithm to use for calculating connected components. [Table 9.8](#) describes the valid values for this option.

Table 9.8 Values for the ALGORITHM= Option

Option Value	Description
DFS	Uses the depth-first search algorithm for connected components.
UNION_FIND	Uses the union-find algorithm for connected components. You can use ALGORITHM=UNION_FIND only with undirected graphs.

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

CYCLE<=(*suboptions*) >

finds the cycles (or the existence of a cycle) in the input graph. For more information, see the section “[Cycle](#)” on page 418.

You can specify the following *suboptions* in the CYCLE= option:

MAXCYCLES=*number*

specifies the maximum number of cycles to return. The default 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.

MAXLENGTH=*number*

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

MAXLINKWEIGHT=*number*

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

MAXNODEWEIGHT=*number*

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

MINLENGTH=*number*

specifies the minimum number of links to allow in a cycle. Any cycle that has fewer links than *number* is removed from the results. The default is 1. By default, only self-loops are filtered. This option works only when you also specify **MODE=ALL_CYCLES**.

MINLINKWEIGHT=*number*

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

MINNODEWEIGHT=*number*

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

MODE=ALL_CYCLES | FIRST_CYCLE

specifies whether to stop after finding the first cycle. [Table 9.9](#) describes the valid values for this option.

Table 9.9 Values for the **MODE=** Option

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

By default, **MODE=FIRST_CYCLE**.

LINEAR_ASSIGNMENT<=() >**LAP<=() >**

solves the minimal-cost linear assignment problem. In graph terms, this problem is also known as the minimum link-weighted matching problem on a bipartite directed graph. The input data (the cost matrix) is defined as a directed graph by specifying the **LINKS=** option in the **SOLVE WITH NETWORK** statement, where the costs are defined as link weights. Internally, the graph is treated as a bipartite directed graph.

For more information, see the section “[Linear Assignment \(Matching\)](#)” on page 423.

MINCOSTFLOW<=() >

MCF<=() >

solves the minimum-cost network flow problem.

For more information, see the section “[Minimum-Cost Network Flow](#)” on page 424.

MINCUT<=(*suboptions*) >

finds the minimum link-weighted cut of an input graph. For more information, see the section “[Minimum Cut](#)” on page 430. You can specify the following *suboptions* in the MINCUT= option:

MAXNUMCUTS=*number*

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

MAXWEIGHT=*number*

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

MINSPANTREE<=() >

MST<=() >

solves the minimum link-weighted spanning tree problem on an input graph. For more information, see the section “[Minimum Spanning Tree](#)” on page 434.

SHORTPATH<=(*suboptions*) >

calculates shortest paths between sets of nodes on the input graph. For more information, see the section “[Shortest Path](#)” on page 436.

You can specify the following suboptions:

PATHS=ALL | LONGEST | SHORTEST

specifies the type of output for shortest paths results.

[Table 9.10](#) lists the valid values for this suboption.

Table 9.10 Values for the PATHS= Option

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

By default, SHORTPATH=ALL.

SINK=*set-name*

specifies the set of sink nodes.

SOURCE=*set-name*

specifies the set of source nodes.

USEWEIGHT=YES | NOspecifies whether to use weights in calculating shortest paths as listed in [Table 9.11](#).**Table 9.11** Values for USEWEIGHT= Option

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

By default, USEWEIGHT=YES.

TRANSITIVE_CLOSURE<=() >**TRANSCL**<=() >calculates the transitive closure of an input graph. For more information, see the section “[Transitive Closure](#)” on page 449.**TSP**<= (*suboptions*) >solves the traveling salesman problem. For more information, see the section “[Traveling Salesman Problem](#)” on page 452.

The algorithm that is used to solve this problem is built around the same method as is used in PROC OPTMILP: a branch-and-cut algorithm. Many of the following *suboptions* are the same as those described for the OPTMILP procedure in the *SAS/OR User's Guide: Mathematical Programming*.

You can specify the following *suboptions*:**ABSOBJGAP**=*number*

specifies a stopping criterion. When the absolute difference between the best integer objective and the objective of the best remaining branch-and-bound node becomes less than the value of *number*, the solver stops. The value of *number* can be any nonnegative number. By default, ABSOBJGAP=1E-6.

CONFLICTSEARCH=*number* | *string*

specifies the level of conflict search that the network solver performs. The solver performs a conflict search to find clauses that result from infeasible subproblems that arise in the search tree. [Table 9.12](#) describes the valid values for this option.

Table 9.12 Values for CONFLICTSEARCH= Option

<i>number</i>	<i>string</i>	Description
-1	AUTOMATIC	Performs a conflict search based on a strategy that is determined by the network solver

Table 9.12 (continued)

<i>number</i>	<i>string</i>	Description
0	NONE	Disables conflict search
1	MODERATE	Performs a moderate conflict search
2	AGGRESSIVE	Performs an aggressive conflict search

By default, CONFLICTSEARCH=AUTOMATIC.

CUTOFF=number

cuts off any branch-and-bound nodes in a minimization problem that has an objective value that is greater than *number*. The value of *number* can be any number.

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

CUTSTRATEGY=number | string

specifies the level of cutting planes to be generated by the network solver. TSP-specific cutting planes are always generated. [Table 9.13](#) describes the valid values for this option.

Table 9.13 Values for CUTSTRATEGY= Option

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

By default, CUTSTRATEGY=NONE.

EMPHASIS=number | string

specifies a search emphasis option. [Table 9.14](#) describes the valid values for this option.

Table 9.14 Values for EMPHASIS= Option

<i>number</i>	<i>string</i>	Description
0	BALANCE	Performs a balanced search
1	OPTIMAL	Emphasizes optimality over feasibility
2	FEASIBLE	Emphasizes feasibility over optimality

By default, EMPHASIS=BALANCE.

HEURISTICS=*number* | *string*

controls the level of initial and primal heuristics that the network solver applies. This level determines how frequently the network solver applies primal heuristics during the branch-and-bound tree search. It also affects the maximum number of iterations that are allowed in iterative heuristics. Some computationally expensive heuristics might be disabled by the solver at less aggressive levels. Table 9.15 lists the valid values for this option.

Table 9.15 Values for HEURISTICS= Option

<i>number</i>	<i>string</i>	Description
-1	AUTOMATIC	Applies the default level of heuristics
0	NONE	Disables all initial and primal heuristics
1	BASIC	Applies basic initial and primal heuristics at low frequency
2	MODERATE	Applies most initial and primal heuristics at moderate frequency
3	AGGRESSIVE	Applies all initial primal heuristics at high frequency

By default, HEURISTICS=AUTOMATIC.

MAXNODES=*number*

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

By default, MAXNODES= $2^{31} - 1$.

MAXSOLS=*number*

specifies the maximum number of feasible tours to be identified. If *number* solutions have been found, then the solver stops. The value of *number* can be any positive integer up to the largest four-byte signed integer, which is $2^{31} - 1$.

By default, MAXSOLS= $2^{31} - 1$.

MILP=*number* | *string*

specifies whether to use a mixed integer linear programming (MILP) solver for solving the traveling salesman problem. The MILP solver attempts to find the overall best TSP tour by using a branch-and-bound based algorithm. This algorithm can be expensive for large-scale problems. If MILP=OFF, then the network solver uses its initial heuristics to find a feasible, but not necessarily optimal, tour as quickly as possible. Table 9.16 describes the valid values for this option.

Table 9.16 Values for MILP= Option

<i>number</i>	<i>string</i>	Description
1	ON	Uses a mixed integer linear programming solver
0	OFF	Does not use a mixed integer linear programming solver

By default, MILP=ON

NODESEL=*number* | *string*

specifies the branch-and-bound node selection strategy option. For more information about node selection, see Chapter 13, “The OPTMILP Procedure.” Table 9.17 describes the valid values for this option.

Table 9.17 Values for NODESEL= Option

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

By default, NODESEL=AUTOMATIC.

PROBE=*number* | *string*

specifies a probing option. Table 9.18 describes the valid values for this option.

Table 9.18 Values for PROBE= Option

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

By default, PROBE=NONE.

RELOBJGAP=*number*

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

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

When this value becomes less than the specified gap size *number*, the solver stops. The value of *number* can be any nonnegative number.

By default, RELOBJGAP=1E–4.

STRONGITER=*number* | **AUTOMATIC**

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

STRONGLEN=*number* | AUTOMATIC

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

TARGET=*number*

specifies a stopping criterion for minimization problems. If the best integer objective is better than or equal to *number*, the solver stops. The value of *number* can be any number.

By default, TARGET is the negative number that has the largest absolute value that can be represented in your operating environment.

VARSEL=*number* | *string*

specifies the rule for selecting the branching variable. For more information about variable selection, see Chapter 13, “The OPTMILP Procedure.” Table 9.19 describes the valid values for this option.

Table 9.19 Values for VARSEL= Option

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

By default, VARSEL=AUTOMATIC.

Details: Network Solver

The network solver uses a collection of specialized algorithms that optimize specific types of common problems. When you use the network solver, you specify variable arrays, numeric arrays, and sets, both to define an instance and to get solutions, without explicitly formulating objectives and constraints.

Input Data for the Network Solver

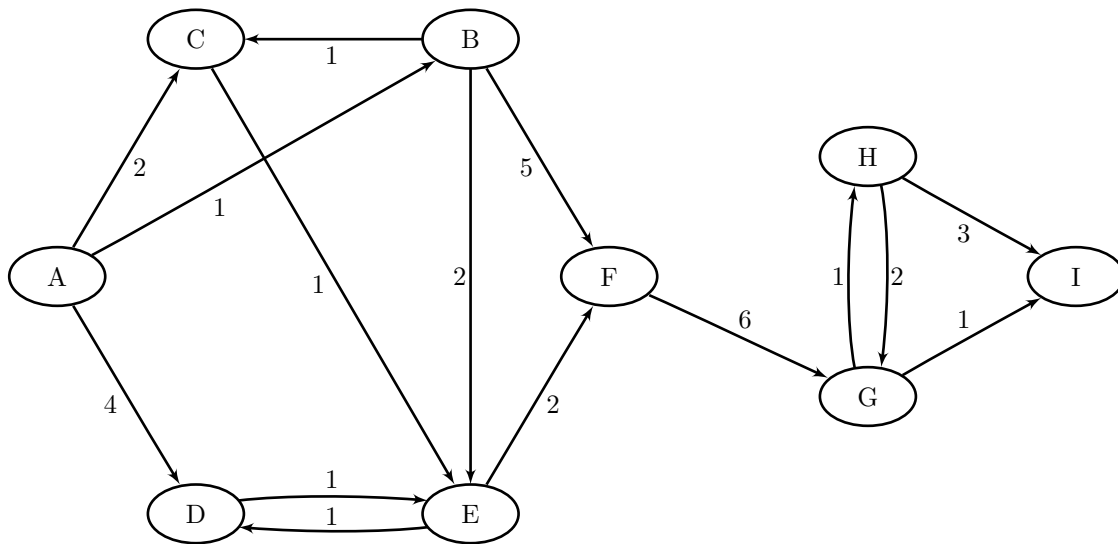
This section describes how you can import and export node, link, and problem data from and to SAS data sets and how you can solve problems over a subgraph without changing your original sets. The section “Graph Input Data” on page 399 describes how to load node and link data in some common formats. The section “Solving over Subsets of Nodes and Links (Filters)” on page 402 describes subgraphs and how to access the objective value of a network problem.

Graph Input Data

This section describes how to input a graph for analysis by the network solver. Because PROC OPTMODEL uses node and link attributes that are indexed over the sets of nodes and links, you need to provide only node and link attributes. PROC OPTMODEL infers the graph from the attributes you provide. When a documented default value exists for the attribute of a link or a node, you need to provide only the values that differ from the default. For example, the section “[Minimum-Cost Network Flow](#)” on page 424 assumes that the link flow upper bound is ∞ . You need to specify only the finite upper bounds.

Consider the directed graph shown in [Figure 9.5](#).

Figure 9.5 A Simple Directed Graph



Each node and link has associated attributes: a node label and a link weight.

None of the algorithms in PROC OPTMODEL support Null Graphs, i.e., graphs with 0 nodes. PROC OPTMODEL will usually raise a semantic error and stop processing any remaining statements in a block if after processing its inputs it determines that the graph is null. If the graph definition itself is not null, but the graph to be passed to the solver after applying the [SUBGRAPH=](#) option is null, then the predeclared parameter `_SOLUTION_STATUS_` will be set to `NULL_GRAPH`. For more information, see “[Solving over Subsets of Nodes and Links \(Filters\)](#)” on page 402.

Data Indexed by Nodes or Links

Nodes often represent entities, and links represent relationships between these entities. Therefore, it is common to store a graph as a link-indexed table. When nodes have attributes beyond their name (label), these attributes are stored in a node-indexed table. This section covers the more complex link-indexed case. The node-indexed case is essentially identical to this one, except that the PROC OPTMODEL set has tuple length of one when node-indexed data are read, whereas the PROC OPTMODEL set has tuple length two when link-indexed data are read.

Let $G = (N, A)$ define a graph with a set N of nodes and a set A of links. A link is an ordered pair of nodes. Each node is defined by using either numeric or string labels.

The directed graph G shown in Figure 9.5 can be represented by the following links data set LinkSetIn:

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

The following statements read in this graph and print the resulting links and nodes sets. These statements do not run any algorithms, so the resulting output contains only the input graph.

```
proc optmodel;
  set<str,str> LINKS;
  set NODES = union{<ni,nj> in LINKS} {ni,nj};
  num weight{LINKS};

  read data LinkSetIn into LINKS=[from to] weight;

  print weight;
  put NODES=; /* computed automatically by OPTMODEL */
quit;
```

The network solver preserves the node order of each link that you provide, even in cases where the link is traversed in the opposite order, such as in paths or tours. For an example, see the tour (1, 4, 2, 3, 5) in Figure 9.8.

The log output in Figure 9.6 shows the nodes that are read from the input link data set. In this example PROC OPTMODEL computed the node set N (NODES) from its definition when it was needed. The ODS output in Figure 9.7 shows the weights that are read from the input link data set, which is indexed by link. PUT is used for NODES because PROC OPTMODEL sets are basic types such as number and string. Thus, you use PUT to quickly inspect a set value. In contrast, you use PRINT to inspect an array, such as weight.

Figure 9.6 Node Set Printout of a Simple Directed Graph

NOTE: There were 15 observations read from the data set WORK.LINKSETIN.
 NODES={'A','B','C','D','E','F','G','H','I'}

Figure 9.7 Link Set of a Simple Directed Graph That Includes Weights**The OPTMODEL Procedure**

[1]	[2]	weight
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

As described in the `GRAPH_DIRECTION=` option, if the graph is undirected, the *from* and *to* labels are interchangeable. If you define this graph as undirected, then reciprocal links (for example, $D \rightarrow E$ and $E \rightarrow D$) are treated as the same link, and duplicates are removed. The network solver takes the first occurrence of the link and ignores the others. By default, `GRAPH_DIRECTION=UNDIRECTED`, so to declare the graph as undirected you can just omit this option.

After you read the data into PROC OPTMODEL sets, you pass link information to the solver by using the `LINKS=` option. Node input is analogous to link input. You pass node information to the solver by using the `NODES=` option.

The `INCLUDE=` suboption is especially useful for algorithms that depend only on the graph topology, (such as the `connected components` algorithm). If an algorithm requires a node or link property and that property is not defined for a node or link that is added by the `INCLUDE=` suboption, the algorithm will not run.

Matrix Input Data

The contents of a table can be represented as a graph. The relationships between two sets of nodes, N_1 and N_2 , can be represented by a $|N_1|$ by $|N_2|$ incidence matrix A , in which N_1 is the set of rows and N_2 is the set of columns.

To read a matrix that is stored in a data set into PROC OPTMODEL, you need to take two extra steps:

1. Determine the name of each numeric variable that you want to use. PROC CONTENTS can be useful for this task.
2. Use an iterated `READ DATA` statement.

For more information, see “Example 9.3: Linear Assignment Problem for Minimizing Swim Times” on page 468.

Solving over Subsets of Nodes and Links (Filters)

You can solve a problem over a subgraph without declaring new link and node sets. You can specify the `LINKS=` and `NODES=` suboptions of the `SUBGRAPH=` option to filter nodes and links before PROC OPTMODEL builds and solves the instance. If you want to see the resulting subgraph, you can specify the `LINKS=` and `NODES=` suboptions of the `OUT=` option. If you just want to produce a subgraph, you do not need to invoke an algorithm.

You can keep all the input and output arrays defined over the original graph and define a subgraph by providing any combination of the `LINKS=` and `NODES=` suboptions of the `SUBGRAPH=` option. If you specify either of the suboptions of the `SUBGRAPH=` option, then union semantics apply. PROC OPTMODEL uses the following rules:

- Only the links that are included in the set named in the `LINKS=` option are used to create the instance.
- Only the nodes that appear either in the `NODES=` suboption of the `SUBGRAPH=` option or that appear as the head or tail of a link in the `LINKS=` suboption are used to create the instance.
- A node or a link that appears only in the `SUBGRAPH=` option, but not in the original graph, is discarded. To add nodes or links that do not have attributes, see the `INCLUDE=` suboption of the `LINKS=` and `NODE=` options.

If the value of the `LOGLEVEL=` suboption is equal to or greater than 3, PROC OPTMODEL issues a message for each of the nodes and links that it discards until the number of messages issued during problem generation reaches the value of the `MSGLIMIT=` option in the PROC OPTMODEL statement. If the value of the `LOGLEVEL=` suboption is greater than 0, PROC OPTMODEL also issues a summary that shows the total count of discarded nodes and links from each input array or set.

The following statements call PROC OPTMODEL and declare a five-node complete undirected graph; a subset of links that contains all links between nodes 1, 2, 3, and 4; and a subset of nodes that contains nodes 3, 4, and 5:

```
proc optmodel;
  set NODES = 1..5;
  set LINKS = {vi in NODES, vj in NODES: vi < vj};
  num distance {<vi,vj> in LINKS} = 10*vi + vj;

  set <num,num> TOUR;

  /* Build a link set using only nodes 1..4 nodes */
  set <num,num> LINKS_BETWEEN_1234 = {vi in 1..3, vj in (vi+1)..4};
  /* Build a node subset consisting of nodes 3..5 */
  set NODES_345 = 3..5;
```

After the sets are declared, the statements in the following steps solve several traveling salesman problems (TSPs) on subgraphs. For more information about TSPs, see the section “[Traveling Salesman Problem](#)” on page 452.

1. The first SOLVE statement solves a TSP on the original graph. Note that the links in the tour (see [Figure 9.8](#)) are returned with the same orientation that you provide in the input. For example, the second step on the tour goes from node 4 to node 2 using link (2, 4). This guarantees that you do not need to do extra processing of output to check for link orientation. You can just use the output directly.

```

/* Implicit network 1: solve over nodes 1..5 -- The original network*/
solve with NETWORK /
    links=( weight=distance )
    out=( tour=TOUR )
    tsp
;
put TOUR=;

```

As shown in [Figure 9.8](#), all links implied by the WEIGHT= suboption of the LINKS= option become part of the graph.

Figure 9.8 SOLVE WITH NETWORK Log: Traveling Salesman Tour of an Unfiltered Graph

```

NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 10.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 111 using 0.00 (cpu:
      0.00) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Branch and Cut algorithm is used.
NOTE: Optimal.
NOTE: Objective = 111.
NOTE: Processing the traveling salesman problem used 0.00 (cpu: 0.00) seconds.
TOUR={<1,4>,<2,4>,<2,3>,<3,5>,<1,5>}

```

To access the objective value of a network problem, use the `_OROPTMODEL_NUM_` predefined array. The network solver ignores the `_OBJ_` predefined symbol, which is part of the current named problem. The current named problem is independent of the network solver, because the network solver uses sets and numeric arrays for input and output. For more information, see the sections “Multiple Subproblems” on page 150 and “Solver Status Parameters” on page 161 in Chapter 5, “The OPTMODEL Procedure.”

```
put _OROPTMODEL_NUM_['OBJECTIVE'];
```

2. The next SOLVE statement solves a TSP on the subgraph that is defined by the link set LINKS_BETWEEN_1234.

```

/* Filter on LINKS: solve over nodes 1..4 */
solve with NETWORK /
    links=( weight=distance )
    subgraph=( links=LINKS_BETWEEN_1234 )
    out=( tour=TOUR )
    tsp
;
put TOUR=;

```

As shown in Figure 9.9, the network solver now ignores node 5.

Figure 9.9 SOLVE WITH NETWORK Log: Traveling Salesman Tour over Nodes $N = \{1, 2, 3, 4\}$

```
111
NOTE: The SUBGRAPH= option filtered 4 elements from 'distance.'
NOTE: The number of nodes in the input graph is 4.
NOTE: The number of links in the input graph is 6.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 74 using 0.00 (cpu:
      0.00) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Branch and Cut algorithm is used.
NOTE: Optimal.
NOTE: Objective = 74.
NOTE: Processing the traveling salesman problem used 0.00 (cpu: 0.00) seconds.
TOUR={<1,3>,<2,3>,<2,4>,<1,4>}
```

3. The next SOLVE statement solves a TSP on the subgraph that is defined by the node set NODES_345.

```
/* Filter on NODES: solve over nodes 3..5 */
solve with NETWORK /
  links=( weight=distance )
  subgraph=( nodes=NODES_345 )
  out=( tour=TOUR )
  tsp
;
put TOUR=;
```

As shown in Figure 9.10, the network solver now ignores nodes 1 and 2, along with any links incident to them.

Figure 9.10 SOLVE WITH NETWORK Log: Traveling Salesman Tour over Nodes $N = \{3, 4, 5\}$

```
NOTE: The SUBGRAPH= option filtered 7 elements from 'distance.'
NOTE: The number of nodes in the input graph is 3.
NOTE: The number of links in the input graph is 3.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 114 using 0.00 (cpu:
      0.00) seconds.
NOTE: Optimal.
NOTE: Objective = 114.
NOTE: Processing the traveling salesman problem used 0.00 (cpu: 0.00) seconds.
TOUR={<3,4>,<4,5>,<3,5>}
```

4. The next SOLVE statement attempts to solve a TSP on the subgraph that is defined by the node set NODES_345 and the link set that is defined by the links on the nodes $\{1, 2, 3, 4\}$. This subgraph creates an infeasible instance because the links $\{(1, 5), (2, 5), (3, 5), (4, 5)\}$ that were defined in the original graph have been filtered out. Thus, node 5 is disconnected and no tour can exist.


```

/* Explicit nodes and links: semantic error over nodes 1..5
 * Links <u,5> are undefined and no documented default exists. */
solve with NETWORK /
  links=( weight=distance )
  subgraph=( nodes=NODES_345 links=LINKS_BETWEEN_1234 )
  out=( tour=TOUR )
  tsp
;

```

As shown in Figure 9.11, the network solver identifies that no tour exists over the surviving nodes and links.

Figure 9.11 SOLVE WITH NETWORK Log: Infeasible Traveling Salesman Problem after Filtering

```

NOTE: The SUBGRAPH= option filtered 4 elements from 'distance.'
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 6.
NOTE: The number of singleton nodes in the input graph is 1.
NOTE: Processing the traveling salesman problem.
NOTE: Infeasible.
NOTE: Processing the traveling salesman problem used 0.00 (cpu: 0.00) seconds.

```

5. The last SOLVE statement uses the **LINKS=** suboption of the **OUT=** option to capture exactly which nodes and links were generated and with which attributes. In this case, because the only attribute defined is link weight, the set **LINKS_OUT** has tuples of length three.

```

/* make room for tail, head, and weight */
set<num,num,num> LINKS_OUT;
solve with NETWORK /
  links=( weight=distance )
  subgraph=( nodes=NODES_345 links=LINKS_BETWEEN_1234 )
  out=( tour=TOUR links=LINKS_OUT )
  tsp
;
put LINKS_OUT=;
quit;

```

As shown in Figure 9.12, the network solver can return the graph after filtering. This feature can sometimes help you identify why you might get counterintuitive results.

Figure 9.12 SOLVE WITH NETWORK Log: Remaining Links after Filtering

```

NOTE: The SUBGRAPH= option filtered 4 elements from 'distance.'
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 6.
NOTE: The number of singleton nodes in the input graph is 1.
NOTE: Processing the traveling salesman problem.
NOTE: Infeasible.
NOTE: Processing the traveling salesman problem used 0.00 (cpu: 0.00) seconds.
LINKS_OUT={<1,2,12>,<1,3,13>,<1,4,14>,<2,3,23>,<2,4,24>,<3,4,34>}
NOTE: The PROCEDURE OPTMODEL printed pages 5-14.

```

Numeric Limitations

Extremely large or extremely small numerical values might cause computational difficulties for some of the algorithms in the network solver. For this reason, each algorithm restricts the magnitude of the data values to a particular threshold number. If the user data values exceed this threshold, the network solver issues an error message. The value of the threshold limit is different for each algorithm and depends on the operating environment. The threshold limits are listed in Table 9.20, where M is defined as the largest absolute value representable in your operating environment.

Table 9.20 Threshold Limits by Algorithm

Algorithm	Graph Links				Graph Nodes	
	weight	weight2	lower	upper	weight	weight2
CYCLE	\sqrt{M}				\sqrt{M}	
LINEAR_ASSIGNMENT	\sqrt{M}					
MINCOSTFLOW	1e15		1e15	1e15	1e15	1e15
MINCUT	\sqrt{M}					
MINSPANTREE	\sqrt{M}					
SHORTPATH	\sqrt{M}	\sqrt{M}				
TSP	1e20					

To obtain these limits, use the SAS CONSTANT function. For example, the following PROC OPTMODEL code assigns \sqrt{M} to a variable x and prints that value to the log:

```
proc optmodel;
  num c = constant('SQRTBIG');
  put c=;
quit;
```

Missing Values

A missing value has no valid interpretation for most of the algorithms in the network solver. If the user data contain a missing value, the network solver issues an error message. There is only one exception: the minimum-cost network flow algorithm interprets a missing value in the lower or upper bound option as the default bound value. For more information about this algorithm, see the section “[Minimum-Cost Network Flow](#)” on page 424.

Negative Link Weights

For certain algorithms in the network solver, a negative link weight is not allowed. The following algorithms issue an error message if a negative link weight is provided:

- MINCUT

Biconnected Components and Articulation Points

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

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

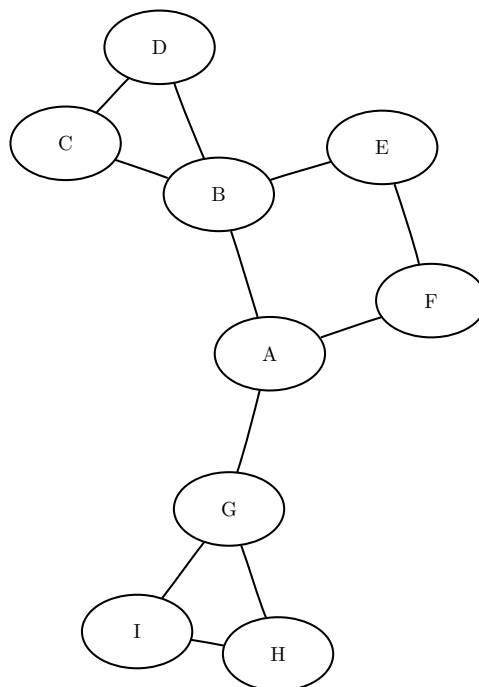
The results for the biconnected components algorithm are written to the link-indexed numeric array that is specified in the BICONCOMP= suboption of the OUT= option. For each link in the links array, the value in this array identifies its component. The component identifiers are numbered sequentially starting from 1. The articulation points are written to the set that is specified in the ARTPOINTS= suboption of the OUT= option.

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

Biconnected Components of a Simple Undirected Graph

This section illustrates the use of the biconnected components algorithm on the simple undirected graph G that is shown in Figure 9.13.

Figure 9.13 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 optmodel;
  set<str,str> LINKS;
  read data LinkSetInBiCC into LINKS=[from to];
  set NODES = union{<i,j> in LINKS} {i,j};
  num bicomponent{LINKS};
  set<str> ARTPOINTS;

  solve with NETWORK /
    loglevel = moderate
    links    = (include=LINKS)
    biconcomp
    out      = (biconcomp=bicomponent artpoints=ARTPOINTS)
  ;

  print bicomponent;
  put ARTPOINTS;
  create data LinkSetOut from [from to] biconcomp=bicomponent;
  create data NodeSetOut from [node]=ARTPOINTS artpoint=1;
quit;
```

The data set LinkSetOut now contains the biconnected components of the input graph, as shown in [Figure 9.14](#).

Figure 9.14 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 9.15.

Figure 9.15 Articulation Points of a Simple Undirected Graph

node	artpoint
A	1
B	1
G	1

The biconnected components are shown graphically in Figure 9.16 and Figure 9.17.

Figure 9.16 Biconnected Components C^1 and C^2

$$C^1 = \{B, C, D\}$$

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

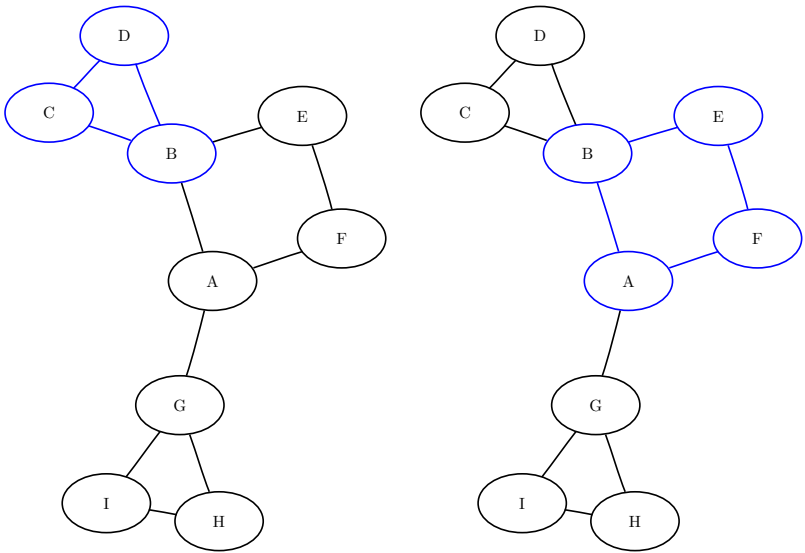
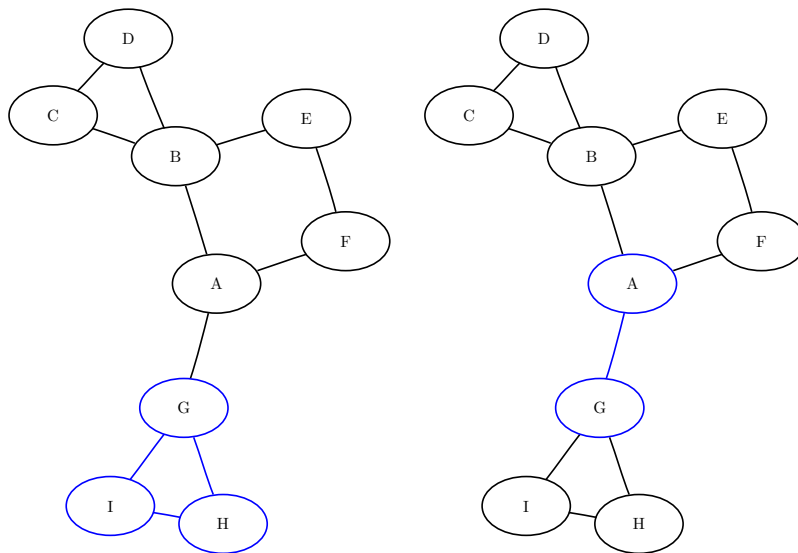


Figure 9.17 Biconnected Components C^3 and C^4

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

$$C^4 = \{A, G\}$$



For a more detailed example, see “Example 9.1: Articulation Points in a Terrorist Network” on page 462.

Clique

A *clique* of a graph $G = (N, A)$ is an induced subgraph that is a complete graph. Every node in a clique is connected to every other node in that clique. A *maximal clique* is a clique that is not a subset of the nodes of any larger clique. That is, it is a set C of nodes such that every pair of nodes in C is connected by a link and every node not in C is missing a link to at least one node in C . The number of maximal cliques in a particular graph can be very large and can grow exponentially with every node added. Finding cliques in graphs has applications in numerous industries including bioinformatics, social networks, electrical engineering, and chemistry.

You can find the maximal cliques of an input graph by invoking the **CLIQUE=** option. The clique algorithm works only with undirected graphs.

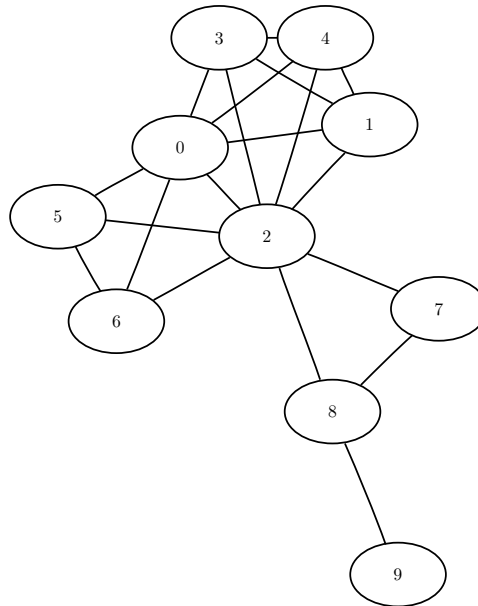
The results for the clique algorithm are written to the set that is specified in the **CLIQUE=** suboption of the **OUT=** option. Each node of each clique is listed in the set along with a clique ID (the first argument of the tuple) to identify the clique to which it belongs. A node can appear multiple times in this set if it belongs to multiple cliques.

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

Maximal Cliques of a Simple Undirected Graph

This section illustrates the use of the clique algorithm on the simple undirected graph G that is shown in Figure 9.18.

Figure 9.18 A Simple Undirected Graph G



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

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

The following statements calculate the maximal cliques, output the results in the data set Cliques, and use the CARD function and SLICE operator as a convenient way to compute the clique sizes, which are output to a data set called CliqueSizes:

```

proc optmodel;
  set<num,num> LINKS;
  read data LinkSetIn into LINKS=[from to];
  set<num,num> CLIQUES;

  solve with NETWORK /
    links = (include=LINKS)
    clique
    out    = (cliques=CLIQUES)
  ;

  put CLIQUES;
  create data Cliques from [clique node]=CLIQUES;
  num num_cliques = card(setof {<cid,node> in CLIQUES} cid);
  set CLIQUE_IDS = 1..num_cliques;
  num size {cid in CLIQUE_IDS} = card(slice(<cid,*>, CLIQUES));
  create data CliqueSizes from [clique] size;
quit;

```

The data set Cliques now contains the maximal cliques of the input graph; it is shown in [Figure 9.19](#).

Figure 9.19 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 9.20](#).

Figure 9.20 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 9.21 and Figure 9.22.

Figure 9.21 Maximal Cliques C^1 and C^2

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

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

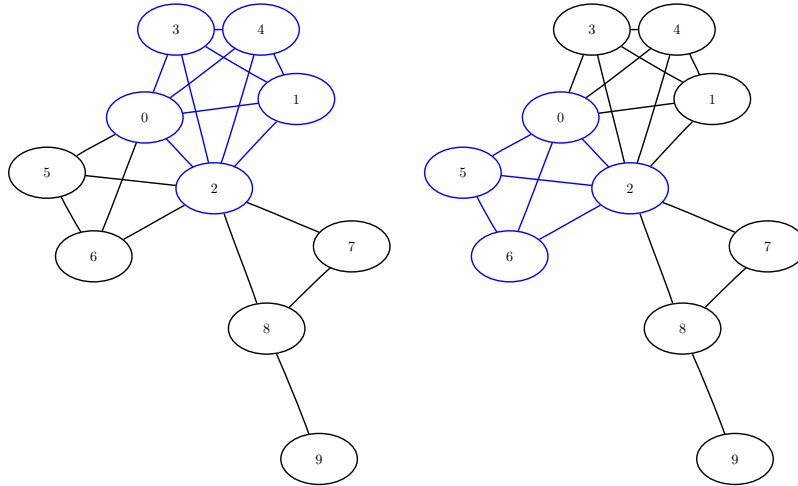
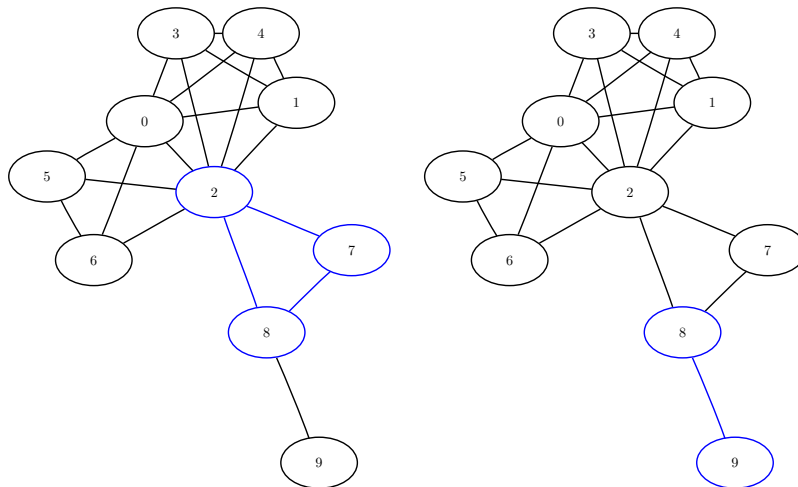


Figure 9.22 Maximal Cliques C^3 and C^4

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

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



Connected Components

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

In the network solver, you can invoke connected components by using the **CONCOMP=** option.

There are two main algorithms for finding connected components in an undirected graph: a depth-first search algorithm (**ALGORITHM=DFS**) and a union-find algorithm (**ALGORITHM=UNION_FIND**). For a graph

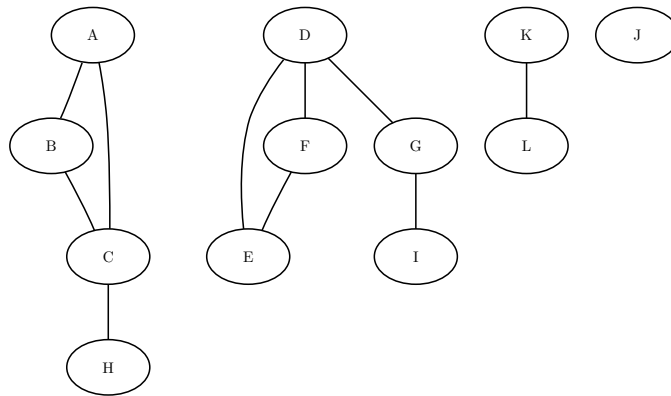
$G = (N, A)$, both algorithms run in time $O(|N| + |A|)$ and can usually scale to very large graphs. The default is the union-find algorithm. For directed graphs, only the depth-first search algorithm is available.

The results of the connected components algorithm are written to the node-indexed numeric array that you specify in the CONCOMP= suboption of the OUT= option. For each node in the set, the value of this array identifies its component. The component identifiers are numbered sequentially starting from 1.

Connected Components of a Simple Undirected Graph

This section illustrates the use of the connected components algorithm on the simple undirected graph G that is shown in Figure 9.23.

Figure 9.23 A Simple Undirected Graph G



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

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

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

```
proc optmodel;
  set<str,str> LINKS;
  read data LinkSetIn into LINKS=[from to];
  set NODES = union {<i,j> in LINKS} {i,j};
  num component{NODES};

  solve with NETWORK /
    links    = (include=LINKS)
    concomp
    out      = (concomp=component)
  ;

  print component;
  create data NodeSetOut from [node] concomp=component;
quit;
```

The data set NodeSetOut contains the connected components of the input graph and is shown in [Figure 9.24](#).

Figure 9.24 Connected Components of a Simple Undirected Graph

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

Notice that the graph is defined by using only the links array. As seen in [Figure 9.23](#), this graph also contains a singleton node labeled J, which has no associated links. By definition, this node defines its own component. But because the input graph was defined by using only the links array, node J did not show up in the results data set. To define a graph by using nodes that have no associated links, you should also define the input nodes set. In this case, you can define a nodes data set NodeSetIn as follows:

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

You could also have defined the set directly in PROC OPTMODEL, but in this case, a separate data set nicely preserves the independence between the model and the data.

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

```
proc optmodel;
  set<str,str> LINKS;
  read data LinkSetIn into LINKS=[from to];
  set<str> NODES;
  read data NodeSetIn into NODES=[node];
  num component{NODES};

  solve with NETWORK /
    links    = (include=LINKS)
    nodes    = (include=NODES)
    concomp
    out      = (concomp=component)
  ;

  print component;
  create data NodeSetOut from [node] concomp=component;
quit;
```

The resulting data set, NodeSetOut, includes the singleton node J as its own component, as shown in Figure 9.25.

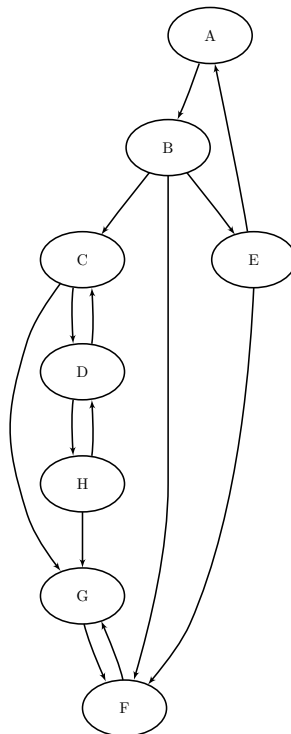
Figure 9.25 Connected Components of a Simple Undirected Graph

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

Connected Components of a Simple Directed Graph

This section illustrates the use of the connected components algorithm on the simple directed graph G that is shown in Figure 9.26.

Figure 9.26 A Simple Directed Graph G



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

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

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

```
proc optmodel;
  set<str,str> LINKS;
  read data LinkSetIn into LINKS=[from to];
  set NODES = union {<i,j> in LINKS} {i,j};
  num component{NODES};

  solve with NETWORK /
    graph_direction = directed
    links           = (include=LINKS)
    concomp
    out             = (concomp=component)
  ;

  print component;
  create data NodeSetOut from [node] concomp=component;
quit;
```

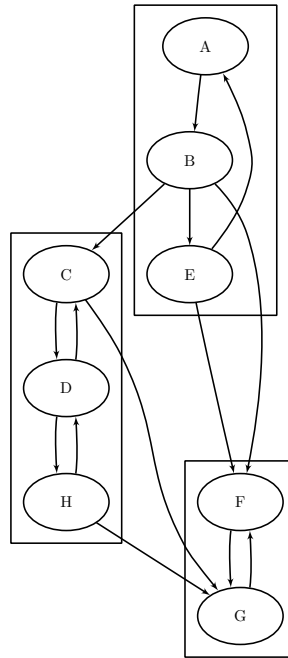
The data set NodeSetOut, shown in [Figure 9.27](#), now contains the connected components of the input graph.

Figure 9.27 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 9.28.

Figure 9.28 Strongly Connected Components of Graph G



Cycle

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

In the network solver, you can find (or just count) the elementary cycles of an input graph by invoking the **CYCLE=** algorithm option. To find the cycles and report them in a set, use the **CYCLES=** suboption in the **OUT=** option. You do not need to use the **CYCLES=** suboption to simply count the cycles.

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

The results of the cycle detection algorithm are written to the set that you specify in the **CYCLES=** suboption in the **OUT=** option. Each node of each cycle is listed in the **CYCLES=** set along with a cycle ID (the first argument of the tuple) to identify the cycle to which it belongs. The second argument of the tuple defines the order (sequence) of the node in the cycle.

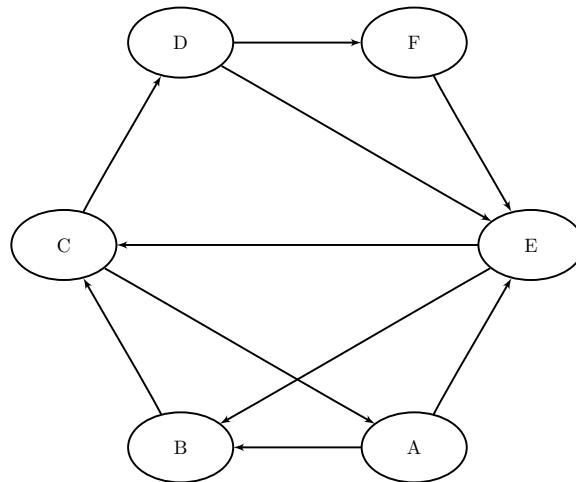
The algorithm that the network solver uses to compute all cycles is a variant of the algorithm in Johnson (1975). This algorithm runs in time $O((|N| + |A|)(c + 1))$, where c is the number of elementary cycles in the graph. So the algorithm should scale to large graphs that contain few cycles. However,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

If **MODE=ALL_CYCLES** and there are many cycles, the **CYCLES=** set can become very large. It might be beneficial to check the number of cycles before you try to create the **CYCLES=** set. When you specify **MODE=FIRST_CYCLE**, the algorithm returns the first cycle that it finds and stops processing. This should run relatively quickly. For large-scale graphs, the **MINLINKWEIGHT=** and **MAXLINKWEIGHT=** suboptions might increase the computation time.

Cycle Detection of a Simple Directed Graph

This section provides a simple example of using the cycle detection algorithm on the simple directed graph G that is shown in Figure 9.29. Two other examples are “[Example 9.2: Cycle Detection for Kidney Donor Exchange](#)” on page 464, which shows the use of cycle detection for optimizing a kidney donor exchange, and “[Example 9.6: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System](#)” on page 476, which shows an application of cycle detection to dependencies between bug reports.

Figure 9.29 A Simple Directed Graph G



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

```

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

```

The following statements check whether the graph has a cycle:

```

proc optmodel;
  set<str,str> LINKS;
  read data LinkSetIn into LINKS=[from to];
  set<num,num,str> CYCLES;

  solve with NETWORK /
    graph_direction = directed
    links           = (include=LINKS)
    cycle           = (mode=first_cycle)
  ;
quit;

```

The result is written to the log of the procedure, as shown in [Figure 9.30](#).

Figure 9.30 Network Solver Log: Check the Existence of a Cycle in a Simple Directed Graph

```
NOTE: There were 10 observations read from the data set WORK.LINKSETIN.
NOTE: The number of nodes in the input graph is 6.
NOTE: The number of links in the input graph is 10.
NOTE: Processing cycle detection.
NOTE: The graph does have a cycle.
NOTE: Processing cycle detection used 0.00 (cpu: 0.00) seconds.
```

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

```
proc optmodel;
  set<str,str> LINKS;
  read data LinkSetIn into LINKS=[from to];
  set<num,num,str> CYCLES;

  solve with NETWORK /
    graph_direction = directed
    links           = (include=LINKS)
    cycle           = (mode=all_cycles)
  ;
quit;
```

The result is written to the log of the procedure, as shown in [Figure 9.31](#).

Figure 9.31 Network Solver Log: Count the Number of Cycles in a Simple Directed Graph

```
NOTE: There were 10 observations read from the data set WORK.LINKSETIN.
NOTE: The number of nodes in the input graph is 6.
NOTE: The number of links in the input graph is 10.
NOTE: Processing cycle detection.
NOTE: The graph has 7 cycles.
NOTE: Processing cycle detection used 0.00 (cpu: 0.00) seconds.
```

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

```
proc optmodel;
  set<str,str> LINKS;
  read data LinkSetIn into LINKS=[from to];
  set<num,num,str> CYCLES;

  solve with NETWORK /
    graph_direction = directed
    links           = (include=LINKS)
    cycle           = (mode=first_cycle)
    out             = (cycles=CYCLES)
  ;

  put CYCLES;
  create data Cycles from [cycle order node]=CYCLES;
quit;
```

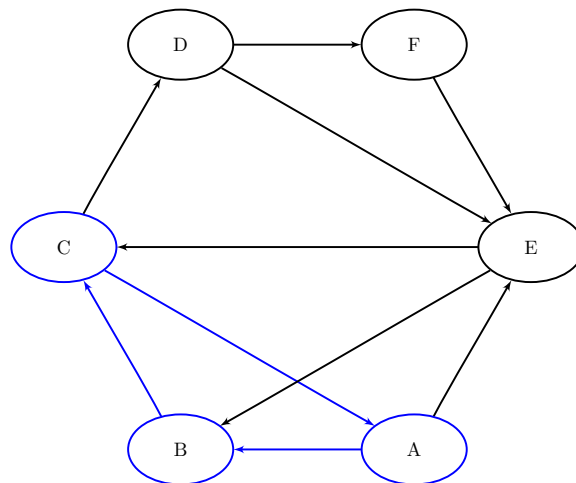

The data set Cycles now contains the first cycle found in the input graph; it is shown in Figure 9.32.

Figure 9.32 First Cycle Found in a Simple Directed Graph

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

The first cycle that is found in the input graph is shown graphically in Figure 9.33.

Figure 9.33 $A \rightarrow B \rightarrow C \rightarrow A$



The following statements return all the cycles in the graph:

```

proc optmodel;
  set<str,str> LINKS;
  read data LinkSetIn into LINKS=[from to];
  set<num,num,str> CYCLES;

  solve with NETWORK /
    graph_direction = directed
    links           = (include=LINKS)
    cycle           = (mode=all_cycles)
    out             = (cycles=CYCLES)
  ;

  put CYCLES;
  create data Cycles from [cycle order node]=CYCLES;
quit;
```

The data set Cycles now contains all the cycles in the input graph; it is shown in Figure 9.34.

Figure 9.34 All Cycles in a Simple Directed Graph

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

The six additional cycles are shown graphically in Figure 9.35 through Figure 9.37.

Figure 9.35 Cycles

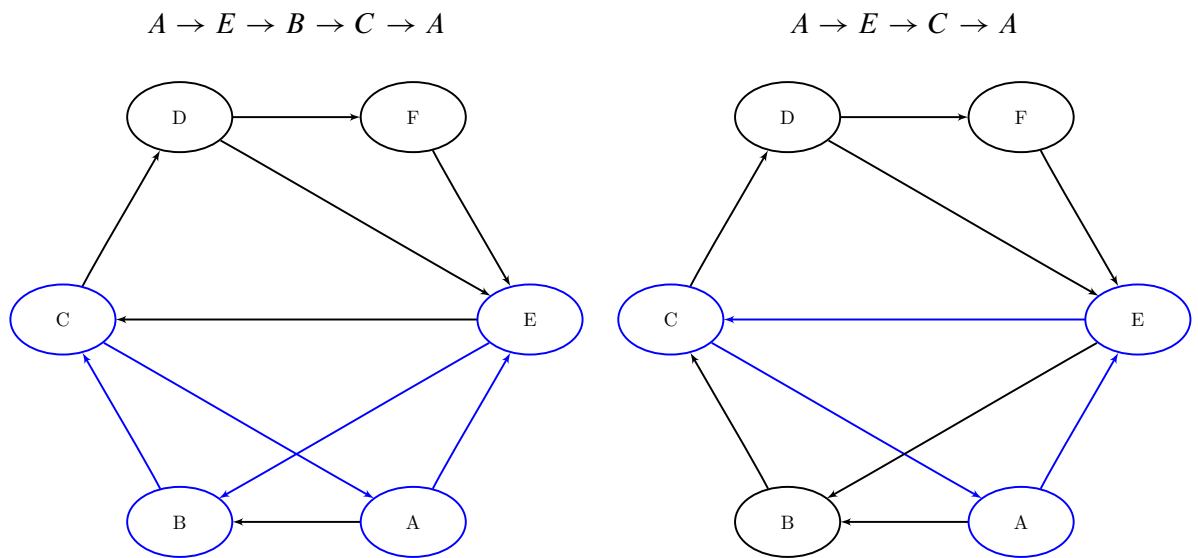
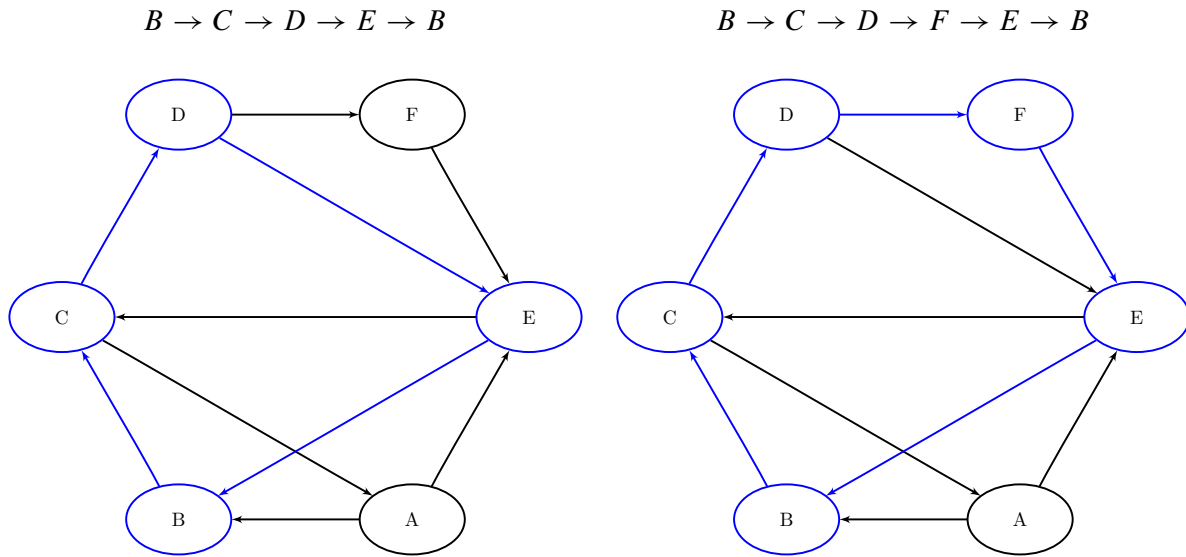
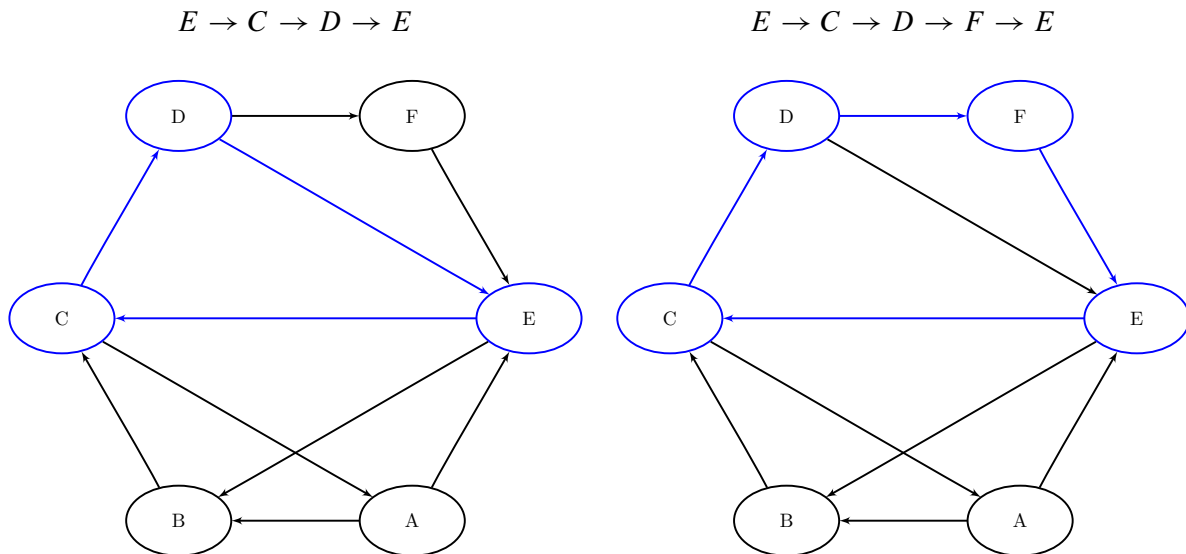


Figure 9.36 Cycles**Figure 9.37** Cycles

Linear Assignment (Matching)

The *linear assignment problem* (LAP) is a fundamental problem in combinatorial optimization that involves assigning workers to tasks at minimal costs. In graph theoretic terms, the LAP is equivalent to finding a minimum-weight matching in a weighted bipartite directed graph. In a *bipartite graph*, the nodes can be divided into two disjoint sets S (workers) and T (tasks) such that every link connects a node in S to a node in T . That is, the node sets S and T are independent. The concept of assigning workers to tasks can be generalized to the assignment of any abstract object from one group to some abstract object from a second group.

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

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

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

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

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

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

In the network solver, you can invoke the linear assignment problem solver by using the **LIN-EAR_ASSIGNMENT** option. The algorithm that the network solver uses for solving a LAP is based on augmentation of shortest paths (Jonker and Volgenant 1987). This algorithm can be applied as long as the graph is bipartite.

The resulting assignment (or matching) is contained in the set that is specified in the **ASSIGNMENTS=** suboption of the **OUT=** option.

For a detailed example, see “**Example 9.3: Linear Assignment Problem for Minimizing Swim Times**” on page 468.

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 from node i to node j . The amount of flow that can be sent across each link is bounded to be within $[l_{ij}, u_{ij}]$. The problem can be modeled as a

linear programming problem as follows:

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

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 the network solver, you can invoke the minimum-cost network flow solver by using the **MINCOSTFLOW** option.

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

The input for the network is the standard graph input, which is described in the section “**Input Data for the Network Solver**” on page 398. The MCF option uses the following suboptions of the **LINKS=** input option that specify link-indexed numeric arrays:

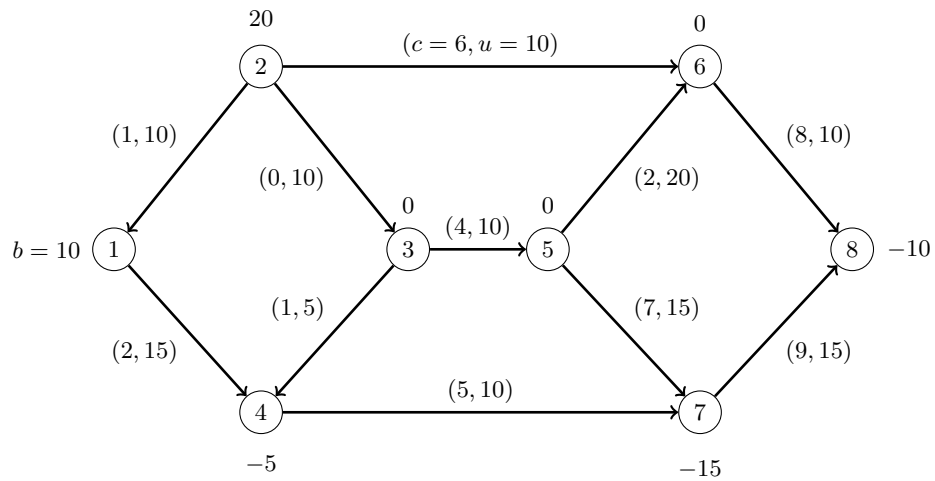
- The **WEIGHT=** suboption defines the link cost c_{ij} per unit of flow. (The default is 0, but if the **WEIGHT=** suboption is not specified, then the default is 1.)
- The **LOWER=** suboption defines the link flow lower bound l_{ij} . (The default is 0.)
- The **UPPER=** suboption defines the link flow upper bound u_{ij} . (The default is ∞ .)

The MCF option uses the **WEIGHT=** suboption of the **NODES=** option to specify supply. The parameter is a numeric array that is positive for supply nodes and negative for demand nodes.

The resulting optimal flow through the network is written to the link-indexed numeric array that is specified in the **FLOW=** suboption of the **OUT=** option in the **SOLVE WITH NETWORK** statement.

Minimum Cost Network Flow for a Simple Directed Graph

The following example demonstrates how to use the network simplex algorithm to find a minimum-cost flow in a directed graph. Consider the directed graph in [Figure 9.38](#), which appears in Ahuja, Magnanti, and Orlin (1993).

Figure 9.38 Minimum-Cost Network Flow Problem: Data

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

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

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

You can use the following call to the network solver to find a minimum-cost flow:

```
proc optmodel;
  set <num,num> LINKS;
  num cost{LINKS};
  num upper{LINKS};
  read data LinkSetIn into LINKS=[from to] cost=weight upper;
  set NODES = union {<i,j> in LINKS} {i,j};
  num supply{NODES} init 0;
  read data NodeSetIn into [node] supply=weight;
  num flow{LINKS};

  solve with network /
    loglevel          = moderate
    logfreq           = 1
    graph_direction   = directed
    links              = (upper=upper weight=cost)
    nodes              = (weight=supply)
    mcf                =
    out                = (flow=flow)
  ;

  print flow;
  create data LinkSetOut from [from to] upper cost flow;
quit;
```

The progress of the procedure is shown in [Figure 9.39](#).

Figure 9.39 Network Solver Log for Minimum-Cost Network Flow

```
NOTE: There were 11 observations read from the data set WORK.LINKSETIN.
NOTE: There were 5 observations read from the data set WORK.NODESETIN.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 11.
NOTE: Processing the minimum-cost network flow problem.
NOTE: The network has 1 connected component.
```

	Primal	Primal	Dual	
Iteration	Objective	Infeasibility	Infeasibility	Time
1	0.000000E+00	2.000000E+01	8.900000E+01	0.00
2	0.000000E+00	2.000000E+01	8.900000E+01	0.00
3	5.000000E+00	1.500000E+01	8.400000E+01	0.00
4	5.000000E+00	1.500000E+01	8.300000E+01	0.00
5	7.500000E+01	1.500000E+01	8.300000E+01	0.00
6	7.500000E+01	1.500000E+01	7.900000E+01	0.00
7	1.300000E+02	1.000000E+01	7.600000E+01	0.00
8	2.700000E+02	0.000000E+00	0.000000E+00	0.00

```
NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: Objective = 270.
NOTE: Processing the minimum-cost network flow problem used 0.00 (cpu: 0.00)
      seconds.
NOTE: The data set WORK.LINKSETOUT has 11 observations and 5 variables.
```

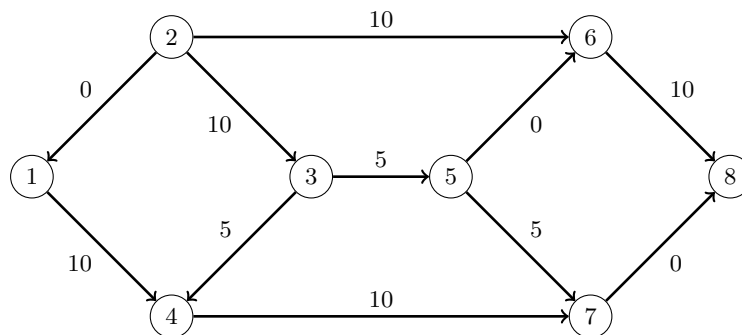
The optimal solution is displayed in Figure 9.40.

Figure 9.40 Minimum-Cost Network Flow Problem: Optimal Solution

Obs	from	to	upper	cost	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 9.41.

Figure 9.41 Minimum-Cost Network Flow Problem: Optimal Solution



Minimum-Cost Network Flow with Flexible Supply and Demand

Using the same directed graph shown in Figure 9.38, this example demonstrates a network that has a flexible supply and demand. Consider the following adjustments to the node bounds:

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

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


```

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

```

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

```

proc optmodel;
  set <num,num> LINKS;
  num cost{LINKS};
  num upper{LINKS};
  read data LinkSetIn into LINKS=[from to] cost=weight upper;
  set <num> NODES;
  num supply{NODES};
  num supplyUB{NODES}; /* also demand lower bound, if negative */
  read data NodeSetIn into NODES=[node] supply=weight supplyUB=weight2;
  num flow{LINKS};

  solve with NETWORK /
    direction = directed
    links      = ( upper = upper weight = cost )
    nodes      = ( weight = supply weight2 = supplyUB )
    mcf
    out        = ( flow = flow )
  ;
  print flow;
  create data LinkSetOut from [from to] upper cost flow;
quit;

```

The progress of the procedure is shown in [Figure 9.42](#).

Figure 9.42 PROC OPTMODEL Log for Minimum-Cost Network Flow

```

NOTE: There were 11 observations read from the data set WORK.LINKSETIN.
NOTE: There were 5 observations read from the data set WORK.NODESETIN.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 11.
NOTE: Processing the minimum-cost network flow problem.
NOTE: Objective = 226.
NOTE: Processing the minimum-cost network flow problem used 0.00 (cpu: 0.00)
      seconds.
NOTE: The data set WORK.LINKSETOUT has 11 observations and 5 variables.

```

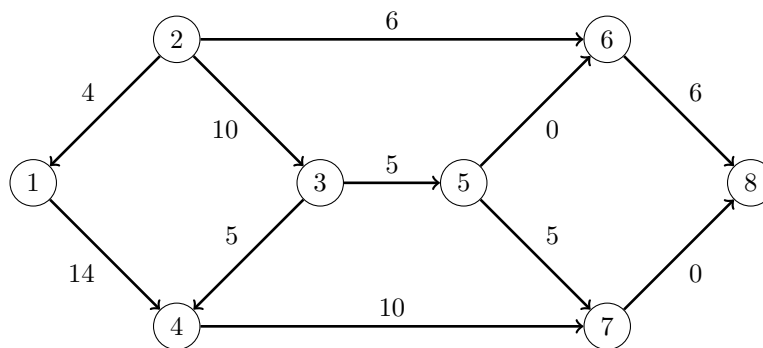
The optimal solution is displayed in Figure 9.43.

Figure 9.43 Minimum-Cost Network Flow Problem: Optimal Solution

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

The optimal solution is represented graphically in Figure 9.44.

Figure 9.44 Minimum-Cost Network Flow Problem: Optimal Solution



Minimum Cut

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

In the network solver, you can invoke the minimum cut algorithm by using the **MINCUT=** option. This algorithm can be used only on undirected graphs.

If the value of the **MAXNUMCUTS=** suboption is greater than 1, then the algorithm can return more than one set of cuts. The resulting cuts can be described in terms of partitions of the nodes of the graph or the links in the cut-sets. The example in the next section illustrates several ways to manipulate the output from the minimum cut algorithm. The node partition is specified in the **PARTITIONS=** suboption of the **OUT=** option in the **SOLVE WITH NETWORK** statement. Each tuple in this set has a cut ID and a node. **PROC OPTMODEL** provides only the smaller of the two subsets that form each partition. You can use the **DIFF** set

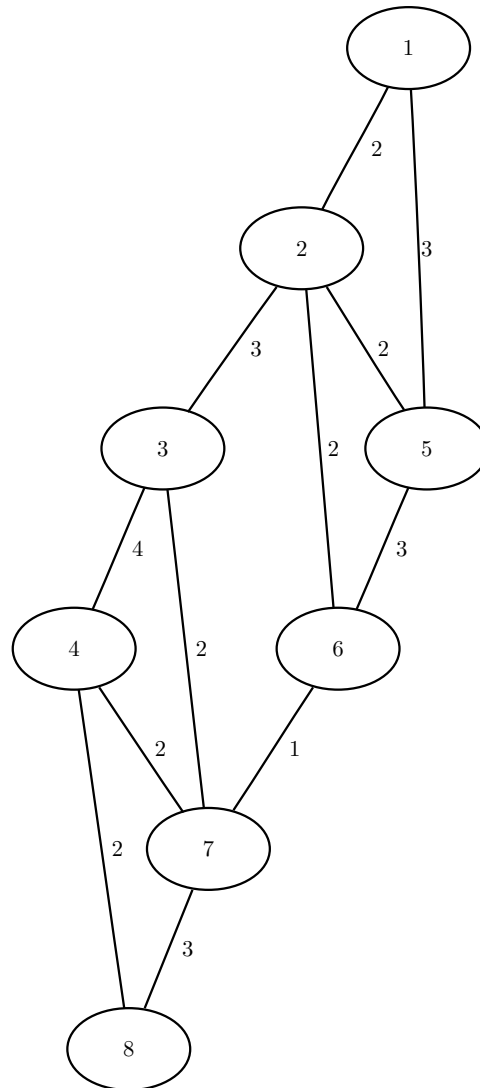
operator to get the complement of each partition. The cut-set is specified in the CUTSETS= suboption of the OUT= option. This set contains the cut ID and the corresponding list of links.

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

Minimum Cut for a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 9.45.

Figure 9.45 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 optmodel;
  set<num,num> LINKS;
  num weight{LINKS};
  read data LinkSetIn into LINKS=[from to] weight;
  set<num> NODES = union {<i,j> in LINKS} {i,j};
  set<num,num> PARTITIONS;
  set<num,num,num> CUTSETS;

  solve with NETWORK /
    loglevel = moderate
    links      = (weight=weight)
    mincut     = (maxnumcuts=3)
    out        = (partitions=PARTITIONS cutsets=CUTSETS)
  ;
  set CUTS = setof {<cut,i,j> in CUTSETS} cut;
  num minCutWeight {cut in CUTS} = sum {<(cut),i,j> in CUTSETS} weight[i,j];
  print minCutWeight;

  for { cut in CUTS }
    put "Cut ID: "      cut
      "Partition: "    ( slice( <cut,*>, PARTITIONS ) )
      "and "           ( NODES diff slice( <cut,*>, PARTITIONS ) )
      "Cut: "          ( slice( <cut,*>,<*>, CUTSETS ) )
      "Weight: "       minCutWeight[cut];

  create data MinCut from [mincut from to]=CUTSETS weight[from,to];
  num mincut {cut in CUTS, node in NODES} =
    if <cut,node> in PARTITIONS then 0 else 1;
  print mincut;
  create data NodeSetOut from [node]=NODES
    {cut in CUTS} <col('mincut_'||cut)=mincut[cut,node]>;
quit;
```

The progress of the procedure is shown in [Figure 9.46](#).

Figure 9.46 Network Solver Log for Minimum Cut

```

NOTE: There were 12 observations read from the data set WORK.LINKSETIN.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 12.
NOTE: Processing the minimum-cut problem.
NOTE: The minimum-cut algorithm found 3 cuts.
NOTE: The cut 1 has weight 4.
NOTE: The cut 2 has weight 5.
NOTE: The cut 3 has weight 5.
NOTE: Processing the minimum-cut problem used 0.00 (cpu: 0.00) seconds.
Cut ID: 1 Partition: {3,4,7,8} and {1,2,5,6} Cut: {<2,3>,<6,7>} Weight: 4
Cut ID: 2 Partition: {8} and {1,2,5,3,6,4,7} Cut: {<4,8>,<7,8>} Weight: 5
Cut ID: 3 Partition: {1} and {2,5,3,6,4,7,8} Cut: {<1,2>,<1,5>} Weight: 5
NOTE: The data set WORK.MINCUT has 6 observations and 4 variables.
NOTE: The data set WORK.NODESETOUT has 8 observations and 4 variables.

```

The data set NodeSetOut now contains the partition of the nodes for each cut, shown in [Figure 9.47](#).

Figure 9.47 Minimum Cut Node Partition

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

The data set MinCut contains the links in the cut-sets for each cut. This data set is shown in [Figure 9.48](#), which also shows each cut separately.

Figure 9.48 Minimum Cut-sets

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

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

Figure 9.48 *continued*

mincut=2		
from	to	weight
4	8	2
7	8	3
mincut		5
mincut=3		
from	to	weight
1	2	2
1	5	3
mincut		5
		14

Minimum Spanning Tree

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

In the network solver, you can invoke the minimum spanning tree algorithm by using the [MINSPANTREE](#) option. This algorithm can be used only on undirected graphs.

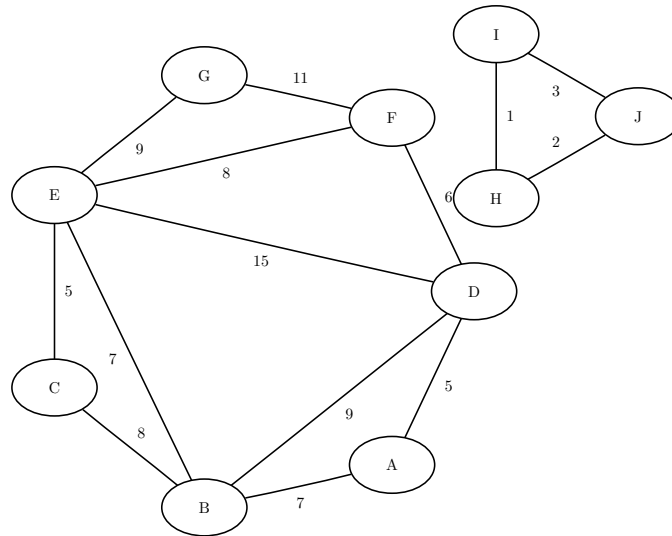
The resulting minimum spanning tree is contained in the set that is specified in the FOREST= suboption of the OUT= option in the SOLVE WITH NETWORK statement.

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

Minimum Spanning Tree for a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 9.49.

Figure 9.49 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 optmodel;
  set<str,str> LINKS;
  num weight{LINKS};
  read data LinkSetIn into LINKS=[from to] weight;
  set<str,str> FOREST;

  solve with NETWORK /
    links      = (weight=weight)
    minspantree
    out        = (forest=FOREST)
  ;

  put FOREST;
  create data MinSpanForest from [from to]=FOREST weight;
quit;
  
```

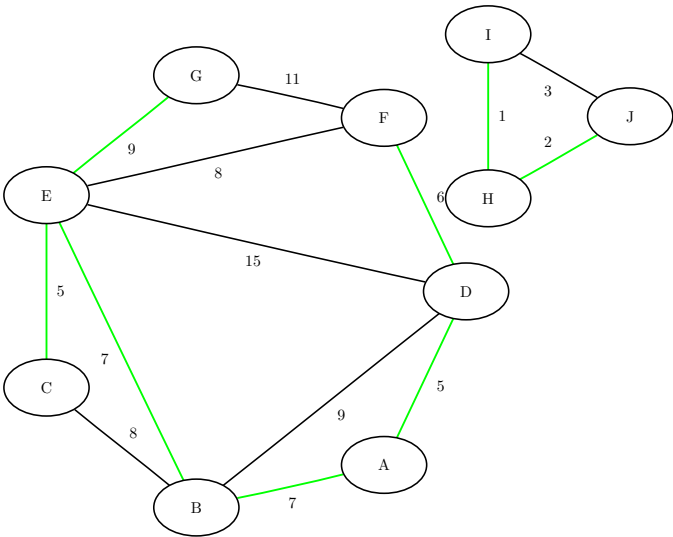
The data set MinSpanForest now contains the links that belong to a minimum spanning forest, which is shown in Figure 9.50.

Figure 9.50 Minimum Spanning Forest

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

The minimal cost links are shown in green in Figure 9.51.

Figure 9.51 Minimum Spanning Forest



For a more detailed example, see “Example 9.5: Minimum Spanning Tree for Computer Network Topology” on page 474.

Shortest Path

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

In the network solver, you can calculate shortest paths by using the **SHORTPATH=** option.

By default, the network solver finds shortest paths for all pairs. That is, it finds a shortest path for each possible combination of source and sink nodes. Alternatively, you can use the **SOURCE=** suboption to fix a particular source node and find shortest paths from the fixed source node to all possible sink nodes. Conversely, by using the **SINK=** suboption, you can fix a sink node and find shortest paths from all possible source nodes to the fixed sink node. By using both suboptions together, you can request one particular

shortest path for a specific source-sink pair. In addition, you can use the **SOURCE=** and **SINK=** suboptions to define a list of source-sink pairs to process. The following sections show examples of these suboptions.

Which algorithm the network solver uses to find shortest paths depends on the data. The algorithm and run-time complexity for each graph type is shown in [Table 9.21](#).

Table 9.21 Algorithms for Shortest Paths

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

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

For weighted graphs, the algorithm uses the parameter that is specified in the **WEIGHT=** suboption of the **SHORTPATH=** option to evaluate a path's total weight (cost).

Outputs

The shortest path algorithm produces up to two outputs. The output set that you specify in the **SPPATHS=** suboption contains the links of a shortest path for each source-sink pair combination. The output parameter that you specify in the **SPWEIGHTS=** suboption contains the total weight for the shortest path for each source-sink pair combination.

SPPATHS= Set

The **SPPATHS=** set contains the links present in the shortest path for each source-sink pair.

The individual links in this set always appear in the order that you provide. If you provide link (u, v) and solve a shortest path problem on an undirected graph, and that path visits node v before node u , then this set will contain link (u, v) , not link (v, u) , which is not part of the network. This approach simplifies indexing throughout your model. In certain use cases, especially for producing output, you might prefer to see the links in the order in which the nodes are visited. For one way to do that, see “[Example 9.7: Traveling Salesman Tour through US Capital Cities](#)” on page 479.

For large graphs and a large requested number of source-sink pairs, this set can be extremely large. For extremely large graphs, generating the output can sometimes take longer than computing the shortest paths. For example, using the US road network data for the state of New York, the data contain a directed graph that has 264,346 nodes. Finding the shortest path for all pairs from only one source node results in 140,969,120 observations, which is a set of size 11 GB. Finding shortest paths for all pairs from all nodes would produce an enormous set.

The **SPPATHS=** set contains the following tuple members:

1. the source node of this shortest path
2. the sink node of this shortest path
3. for this source-sink pair, the order of this link in a shortest path
4. the *tail* node of this link in a shortest path
5. the *head* node of this link in a shortest path

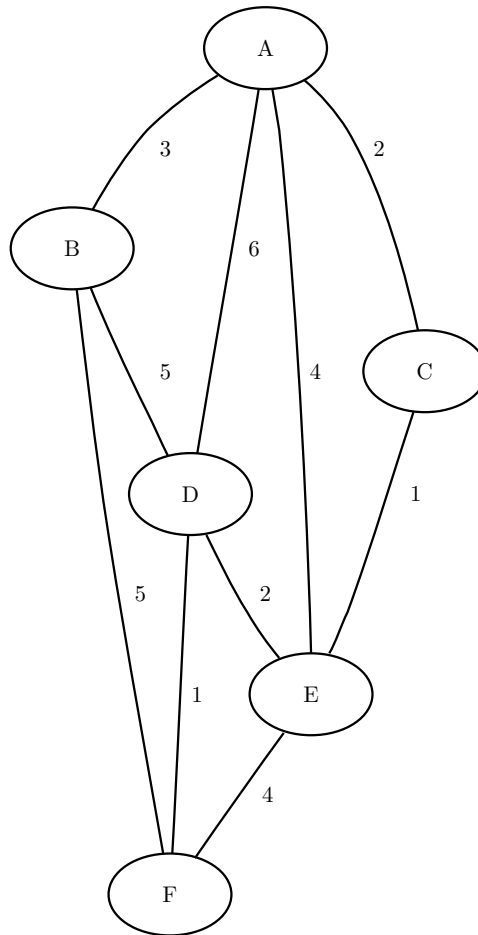
SPWEIGHTS= Array

This array contains the total weight for the shortest path for each of the source-sink pairs.

Shortest Paths for All Pairs

This example illustrates the use of the shortest path algorithm for all source-sink pairs on the simple undirected graph G that is shown in Figure 9.52.

Figure 9.52 A Simple Undirected Graph G



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

```

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

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

```

proc optmodel;
    set <str,str> LINKS;
    num weight{LINKS};
    read data LinkSetIn into LINKS=[from to] weight;
  
```

```

set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
set NODES = union{<i,j> in LINKS} {i,j};
num path_length{NODES, NODES};

solve with NETWORK /
  links      = (weight=weight)
  shortpath
  out        = (sppaths=PATHS spweights=path_length)
;

put PATHS;
print path_length;
create data ShortPathP from [source sink order from to]=PATHS
  weight[from,to];
create data ShortPathW from [source sink]
  path_weight=path_length;
quit;

```

The data set ShortPathP contains the shortest paths and is shown in [Figure 9.53](#).

Figure 9.53 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	D	E	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	D	E	2
A	F	4	D	F	1
B	A	1	A	B	3
B	C	1	A	B	3
B	C	2	A	C	2
B	D	1	B	D	5
B	E	1	A	B	3
B	E	2	A	C	2
B	E	3	C	E	1
B	F	1	B	F	5
C	A	1	A	C	2
C	B	1	A	C	2
C	B	2	A	B	3
C	D	1	C	E	1
C	D	2	D	E	2
C	E	1	C	E	1
C	F	1	C	E	1
C	F	2	D	E	2
C	F	3	D	F	1

source	sink	order	from	to	weight
D	A	1	D	E	2
D	A	2	C	E	1
D	A	3	A	C	2
D	B	1	B	D	5
D	C	1	D	E	2
D	C	2	C	E	1
D	E	1	D	E	2
D	F	1	D	F	1
E	A	1	C	E	1
E	A	2	A	C	2
E	B	1	C	E	1
E	B	2	A	C	2
E	B	3	A	B	3
E	C	1	C	E	1
E	D	1	D	E	2
E	F	1	D	E	2
E	F	2	D	F	1
F	A	1	D	F	1
F	A	2	D	E	2
F	A	3	C	E	1
F	A	4	A	C	2
F	B	1	B	F	5
F	C	1	D	F	1
F	C	2	D	E	2
F	C	3	C	E	1
F	D	1	D	F	1
F	E	1	D	F	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 9.54.

Figure 9.54 All-Pairs Shortest Paths Summary

ShortPathW

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

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

The following statements display only the longest shortest paths:

```
proc optmodel;
  set <str,str> LINKS;
  num weight{LINKS};
  read data LinkSetIn into LINKS=[from to] weight;
  set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */

  solve with NETWORK /
    links      = ( weight = weight )
    shortpath  = ( paths = longest )
    out        = ( sppaths = PATHS )
  ;

  put PATHS;
  create data ShortPathLong from [source sink order from to]=PATHS
    weight[from,to];
quit;
```

The data set ShortPathLong now contains the longest shortest paths and is shown in Figure 9.55.

Figure 9.55 Longest Shortest Paths

ShortPathLong

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

Shortest Paths for a Subset of Source-Sink Pairs

This section illustrates the use of the SOURCE= and SINK= suboptions and the shortest path algorithm to calculate shortest paths for a subset of source-sink pairs. If S denotes the nodes in the SOURCE= set and T denotes the nodes in the SINK= set, the network solver calculates all the source-sink pairs in the crossproduct of these two sets.

For example, the following statements calculate a shortest path for the four combinations of source-sink pairs in $S \times T = \{A, C\} \times \{B, F\}$:

```
proc optmodel;
  set <str,str> LINKS;
  num weight{LINKS};
  read data LinkSetIn into LINKS=[from to] weight;
  set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
  set SOURCES = / A C /;
  set SINKS    = / B F /;

  solve with NETWORK /
    links      = (weight=weight)
    shortpath  = (source=SOURCES sink=SINKS)
    out        = (sppaths=PATHS)
  ;

  put PATHS;
  create data ShortPath from [source sink order from to]=PATHS weight[from,to];
quit;
```

The data set ShortPath contains the shortest paths and is shown in Figure 9.56.

Figure 9.56 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	D	E	2
A	F	4	D	F	1
C	B	1	A	C	2
C	B	2	A	B	3
C	F	1	C	E	1
C	F	2	D	E	2
C	F	3	D	F	1

Shortest Paths for a Subset of Source or Sink Pairs

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

In this case, you designate the subset of source (or sink) nodes in the node set by specifying the SOURCE= (or SINK=) suboption. By specifying only one of the suboptions, you indicate that you want the network solver to calculate all pairs from a subset of source nodes (or to calculate all pairs to a subset of sink nodes).

For example, the following statements calculate all the shortest paths from nodes B and E.:

```
proc optmodel;
  set <str,str> LINKS;
  num weight{LINKS};
  read data LinkSetIn into LINKS=[from to] weight;
  set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
  set SOURCES = / B E /;

  solve with NETWORK /
    links      = (weight=weight)
    shortpath  = (source=SOURCES)
    out        = (sppaths=PATHS)
  ;

  put PATHS;
  create data ShortPath from [source sink order from to]=PATHS weight[from,to];
quit;
```

The data set ShortPath contains the shortest paths and is shown in Figure 9.57.

Figure 9.57 Shortest Paths for a Subset of Source Pairs

ShortPath

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

Conversely, the following statements calculate all the shortest paths to nodes B and E.:

```
proc optmodel;
  set <str,str> LINKS;
  num weight{LINKS};
  read data LinkSetIn into LINKS=[from to] weight;
  set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
  set SINKS = / B E /;

  solve with NETWORK /
    links      = (weight=weight)
    shortpath  = (sink=SINKS)
    out        = (sppaths=PATHS)
  ;

  put PATHS;
  create data ShortPath from [source sink order from to]=PATHS weight[from,to];
quit;
```

The data set ShortPath contains the shortest paths and is shown in Figure 9.58.

Figure 9.58 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	A	B	3
B	E	2	A	C	2
B	E	3	C	E	1
C	B	1	A	C	2
C	B	2	A	B	3
C	E	1	C	E	1
D	B	1	B	D	5
D	E	1	D	E	2
E	B	1	C	E	1
E	B	2	A	C	2
E	B	3	A	B	3
F	B	1	B	F	5
F	E	1	D	F	1
F	E	2	D	E	2

Shortest Paths for One Source-Sink Pair

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

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

```
proc optmodel;
  set <str,str> LINKS;
  num weight{LINKS};
  read data LinkSetIn into LINKS=[from to] weight;
  set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
  set SOURCES = / C /;
  set SINKS    = / F /;

  solve with NETWORK /
    links      = (weight=weight)
    shortpath  = (source=SOURCES sink=SINKS)
    out        = (sppaths=PATHS)
  ;

  put PATHS;
  create data ShortPath from [source sink order from to]=PATHS weight[from,to];
quit;
```

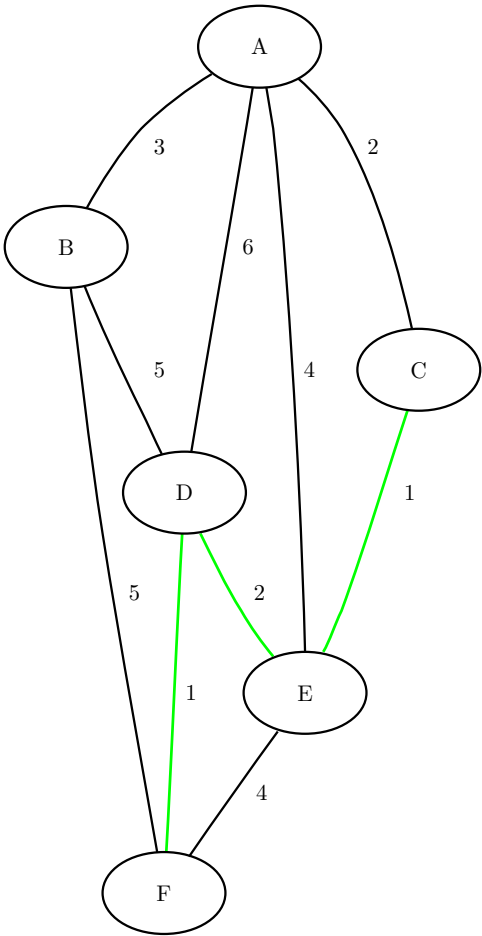

The data set ShortPath contains this shortest path and is shown in Figure 9.59.

Figure 9.59 Shortest Paths for One Source-Sink Pair

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

The shortest path is shown graphically in Figure 9.60.

Figure 9.60 Shortest Path between Nodes C and F



Shortest Paths with Auxiliary Weight Calculation

This section illustrates the use of the shortest path algorithm, where auxiliary weights are used to calculate the shortest paths between all source-sink pairs.

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

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

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

```
proc optmodel;
  set <str,str> LINKS;
  num weight{LINKS};
  num count{LINKS};
  read data LinkSetIn into LINKS=[from to] weight count;
  set <str,str,num,str,str> PATHS; /* source, sink, order, from, to */
  set NODES = union{<i,j> in LINKS} {i,j};
  num path_length{i in NODES, j in NODES: i ~= j};

  solve with NETWORK /
    links      = (weight=weight)
    shortpath
    out        = (sppaths=PATHS spweights=path_length)
  ;

  put PATHS;
  num path_weight2{source in NODES, sink in NODES: source ~= sink} =
    sum {<(source), (sink), order, from, to> in PATHS} count[from,to];
  print path_length path_weight2;
  create data ShortPathW from [source sink]
    path_weight=path_length path_weight2;
quit;
```

The data set ShortPathW contains the total path weight for shortest paths in each source-sink pair and is shown in [Figure 9.61](#). Because the variable count in LinkSetIn has a value of 1 for all links, the value in the output data set variable path_weights2 contains the number of links in each shortest path.

Figure 9.61 Shortest Paths Including Auxiliary Weights in Calculation**ShortPathW**

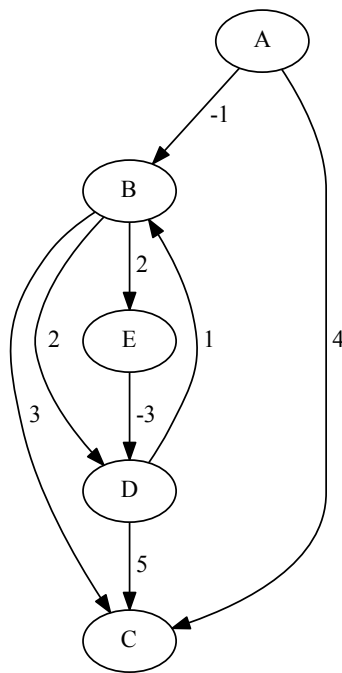
source	sink	path_weight	path_weight2
A	B	3	1
A	C	2	1
A	D	5	3
A	E	3	2
A	F	6	4
B	A	3	1
B	C	5	2
B	D	5	1
B	E	6	3
B	F	5	1
C	A	2	1
C	B	5	2
C	D	3	2
C	E	1	1
C	F	4	3
D	A	5	3
D	B	5	1
D	C	3	2

source	sink	path_weight	path_weight2
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 “[Getting Started: Network Solver](#)” on page 376 shows an example of using the shortest path algorithm to minimize travel to and from work based on traffic conditions.

Shortest Paths with Negative Link Weights

This section illustrates the use of the shortest path algorithm on a simple directed graph G with negative link weights, shown in [Figure 9.62](#).

Figure 9.62 A Simple Directed Graph G with Negative Link Weights

The following statements call PROC OPTMODEL and declare the directed graph G by using set and array literals. For more information about literals, see the section “[NUMBER, STRING, and SET Parameter Declarations](#)” on page 47 in Chapter 5, “[The OPTMODEL Procedure](#).”

```

proc optmodel;
  set LINKS          = / <A B> <A C> <B C> <B D> <B E> <D B> <D C> <E D> /;
  num weight{LINKS} init [  -1    4    3    2    2    1    5   -3  ];

```

The next statements declare a set of the correct type for path output and calculate the shortest paths between source node E and sink node B :

```

  set NODES = union{<i,j> in LINKS} {i,j};
  /* Use the type (in this case, STRING) of NODES but leave PATHS empty */
  set PATHS init {NODES,NODES,/0/,NODES,NODES:0};
  set SOURCE = /E/, SINK = /B/;
  solve with NETWORK /
    links      = ( weight = weight )
    direction = directed
    shortpath = ( source = SOURCE sink = SINK )
    out        = ( sppaths = PATHS )
  ;
  put "Path and Weight: " (setof{<s,t,i,u,v> in PATHS} <u,v,weight[u,v]> );

```

As shown in Figure 9.63, the network solver identifies a shortest path that has negative weights.

Figure 9.63 SOLVE WITH NETWORK Log: Shortest Paths with Negative Link Weights

```
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 8.
NOTE: Processing the shortest paths problem.
NOTE: Processing the shortest paths problem used 0.00 (cpu: 0.00) seconds.
Path and Weight: {<'E','D',-3>,<'D','B',1>}
```

If you reduce the weight on link (B, E) from 2 units to 1 unit, there is a negative weight cycle $(E \rightarrow D \rightarrow B \rightarrow E)$. The Bellman-Ford algorithm catches this and produces an error, as shown in Figure 9.64.

```
weight['B','E'] = 1;
solve with NETWORK /
  links      = (weight=weight)
  direction  = directed
  shortpath  = ( source = SOURCE sink = SINK )
  out        = ( sppaths = PATHS )
;
put _SOLUTION_STATUS_=;
quit;
```

Figure 9.64 SOLVE WITH NETWORK Log: Negative Weight Cycle

```
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 8.
NOTE: Processing the shortest paths problem.
ERROR: The graph contains a negative weight cycle.
NOTE: Processing the shortest paths problem used 0.00 (cpu: 0.00) seconds.
_SOLUTION_STATUS_=BAD_PROBLEM_TYPE
```

Transitive Closure

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

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

In the network solver, you can invoke the transitive closure algorithm by using the [TRANSITIVE_CLOSURE](#) option.

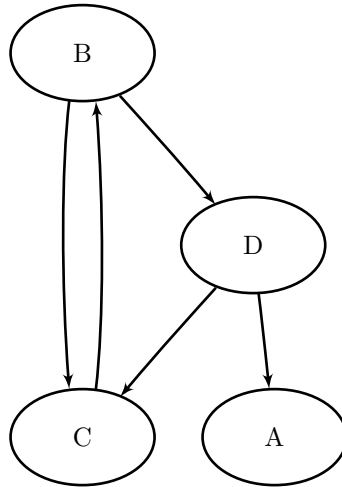
The results for the transitive closure algorithm are written to the set that is specified in the `CLOSURE=` suboption of the `OUT=` option.

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

Transitive Closure of a Simple Directed Graph

This example illustrates the use of the transitive closure algorithm on the simple directed graph G that is shown in Figure 9.65.

Figure 9.65 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 optmodel;
  set<str,str> LINKS;
  read data LinkSetIn into LINKS=[from to];
  set<str,str> CAN_REACH;

  solve with NETWORK /
    links = ( include = LINKS )
    transc
    out = ( closure = CAN_REACH )
  ;

  put CAN_REACH;
  create data TransClosure from [from to]=CAN_REACH;
quit;

```

The data set TransClosure contains the transitive closure of G and is shown in Figure 9.66.

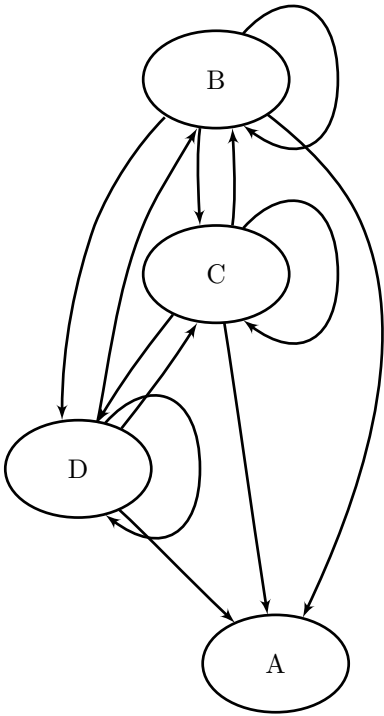
Figure 9.66 Transitive Closure of a Simple Directed Graph

Transitive Closure

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

The transitive closure of G is shown graphically in Figure 9.67.

Figure 9.67 Transitive Closure of G



For a more detailed example, see Example 9.6.

Traveling Salesman Problem

The *traveling salesman problem* (TSP) finds a minimum-cost tour in an undirected graph, G , that has a node set, N , and a link set, A . A *path* in a graph is a sequence of nodes, each of which has a link to the next node in the sequence. An *elementary cycle* is a path in which the start node and end node are the same and otherwise no node appears more than once in the sequence. A *Hamiltonian cycle* (or *tour*) is an elementary cycle that visits every node. In solving the TSP, then, the goal is to find a Hamiltonian cycle of minimum total cost, where the total cost is the sum of the costs of the links in the tour. Associated with each link $(i, j) \in A$ are a binary variable x_{ij} , which indicates whether link x_{ij} is part of the tour, and a cost c_{ij} . Let $\delta(S) = \{(i, j) \in A \mid i \in S, j \notin S\}$. Then an integer linear programming formulation of the TSP (for an undirected graph G) is as follows:

$$\begin{aligned}
 & \text{minimize} && \sum_{(i,j) \in A} c_{ij} x_{ij} \\
 & \text{subject to} && \sum_{(i,j) \in \delta(i)} x_{i,j} = 2 \quad i \in N && (\text{two_match}) \\
 & && \sum_{(i,j) \in \delta(S)} x_{ij} \geq 2 \quad S \subset N, 2 \leq |S| \leq |N| - 1 && (\text{subtour_elim}) \\
 & && x_{ij} \in \{0, 1\} && (i, j) \in A
 \end{aligned}$$

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

For a directed graph, G , the same formulation and solution approach is used on an expanded graph G' , as described in Kumar and Li (1994). The network solver takes care of the construction of the expanded graph and returns the solution in terms of the original input graph.

In practical terms, you can think of the TSP in the context of a routing problem in which each node is a city and the links are roads that connect those cities. If you know the distance between each pair of cities, the goal is to find the shortest possible route that visits each city exactly once and returns to the starting city. The TSP has applications in planning, logistics, manufacturing, genomics, and many other areas.

In the network solver, you can invoke the traveling salesman problem solver by using the **TSP=** option.

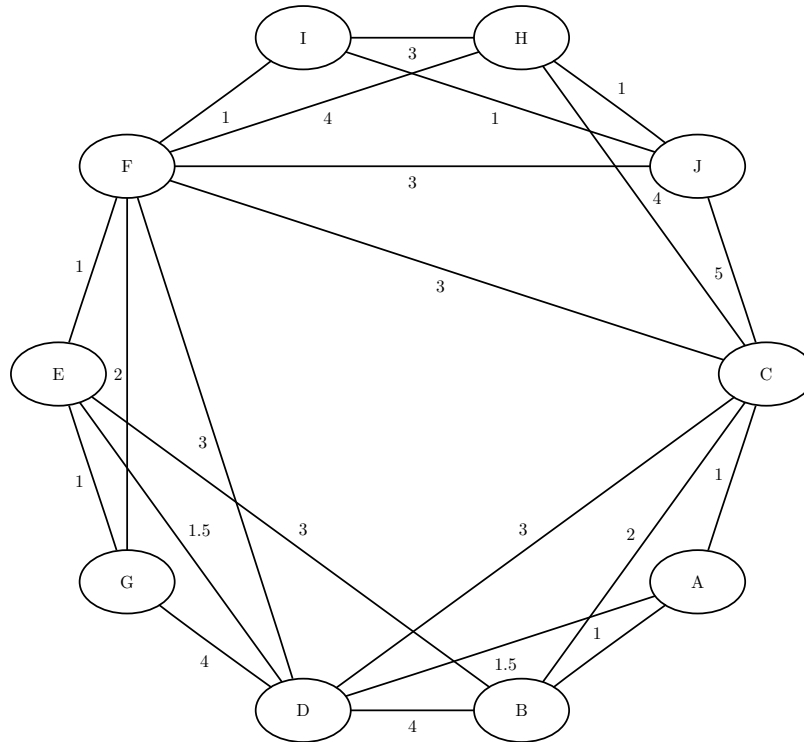
The algorithm that the network solver uses for solving a TSP is based on a variant of the branch-and-cut process described in Applegate et al. (2006).

The resulting tour is represented in two ways: in the numeric array that is specified in the **ORDER=** suboption in the **SOLVE WITH NETWORK** statement, the tour is specified as a sequence of nodes; in the set that is specified in the **TOUR=** suboption of the **TSP** option, the tour is specified as a list of links in the optimal tour.

Traveling Salesman Problem Applied to a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 9.68.

Figure 9.68 A Simple Undirected Graph



You can represent the links data set as follows:

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

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

```
proc optmodel;
  set<str,str> EDGES;
  set<str> NODES = union{<i,j> in EDGES} {i,j};
  num weight{EDGES};
  read data LinkSetIn into EDGES=[from to] weight;
  num tsp_order{NODES};
  set<str,str> TOUR;

  solve with NETWORK /
    loglevel = moderate
```

```

links      = (weight=weight)
tsp
out        = (order=tsp_order tour=TOUR)
;

put TOUR;
print {<i,j> in TOUR} weight;
print tsp_order;
create data NodeSetOut from [node]          tsp_order;
create data TSPTour      from [from to]=TOUR weight;
quit;

```

The progress of the procedure is shown in Figure 9.69.

Figure 9.69 Network Solver Log: Optimal Traveling Salesman Tour of a Simple Undirected Graph

```

NOTE: There were 22 observations read from the data set WORK.LINKSETIN.
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 22.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 16 using 0.00 (cpu:
      0.00) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Branch and Cut algorithm is used.

```

	Node	Active	Sols	BestInteger	BestBound	Gap	Time
	0	1	1	16.0000000	15.5005000	3.22%	0
	0	0	1	16.0000000	16.0000000	0.00%	0

```

NOTE: Optimal.
NOTE: Objective = 16.
NOTE: Processing the traveling salesman problem used 0.01 (cpu: 0.00) seconds.
{<'A', 'B'>, <'B', 'C'>, <'C', 'H'>, <'H', 'J'>, <'J', 'I'>, <'I', 'F'>, <'F', 'G'>, <'G', 'E'>,
<'E', 'D'>, <'D', 'A'>}
NOTE: The data set WORK.NODESETOUT has 10 observations and 2 variables.
NOTE: The data set WORK.TSPTOUR has 10 observations and 3 variables.

```

The data set NodeSetOut now contains a sequence of nodes in the optimal tour and is shown in Figure 9.70.

Figure 9.70 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 9.71](#).

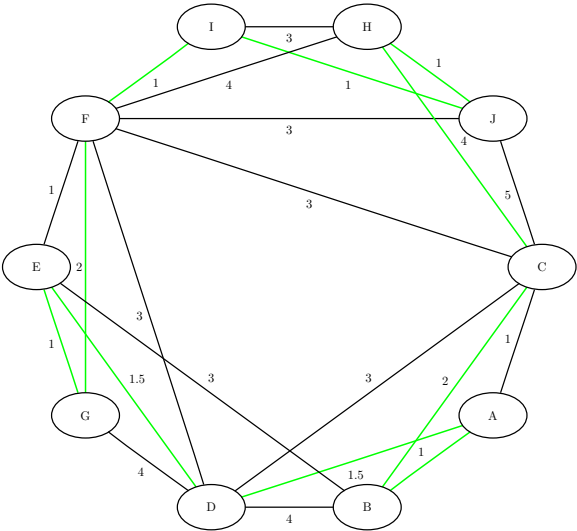
Figure 9.71 Links in the Optimal Traveling Salesman Tour

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 9.72](#).

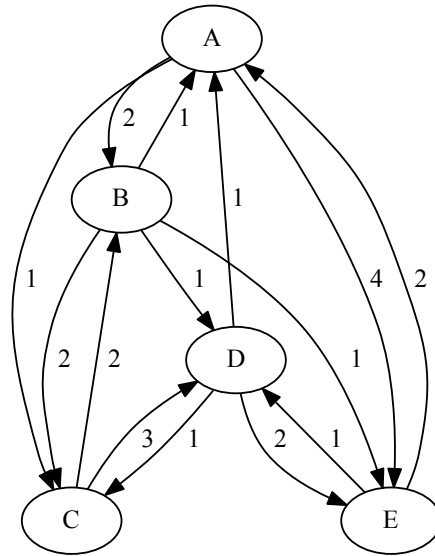
Figure 9.72 Optimal Traveling Salesman Tour



Traveling Salesman Problem Applied to a Simple Directed Graph

As another simple example, consider the weighted directed graph in Figure 9.73.

Figure 9.73 A Simple Directed Graph



You can represent the links data set as follows:

```

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

The following statements, which are identical to those in the undirected example above except for the SOLVE statement clause DIRECTION=DIRECTED, calculate an optimal traveling salesman tour (on a directed graph) and output the results in the data sets TSPTour and NodeSetOut:

```

proc optmodel;
  set<str,str> EDGES;
  set<str> NODES = union{<i,j> in EDGES} {i,j};
  num weight{EDGES};
  read data LinkSetIn into EDGES=[from to] weight;
  num tsp_order{NODES};
  set<str,str> TOUR;

  solve with NETWORK /
    loglevel = moderate
  
```

```

links      = (weight=weight)
direction  = directed
tsp
out        = (order=tsp_order tour=TOUR)
;

put TOUR;
print {<i,j> in TOUR} weight;
print tsp_order;
create data NodeSetOut from [node]      tsp_order;
create data TSPTour    from [from to]=TOUR weight;
quit;

```

The progress of the procedure is shown in [Figure 9.74](#).

Figure 9.74 Network Solver Log: Optimal Traveling Salesman Tour of a Simple Directed Graph

```

NOTE: There were 14 observations read from the data set WORK.LINKSETIN.
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 14.
NOTE: The TSP solver is starting using an augmented symmetric graph with 10
      nodes and 19 links.
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 6 using 0.00 (cpu:
      0.00) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Branch and Cut algorithm is used.

```

Node	Active	Sols	BestInteger	BestBound	Gap	Time
0	1	1	6.0000000	5.9001000	1.69%	0
0	0	1	6.0000000	6.0000000	0.00%	0

```

NOTE: Optimal.
NOTE: Objective = 6.
NOTE: Processing the traveling salesman problem used 0.01 (cpu: 0.00) seconds.
{<'A','C'>,<'C','B'>,<'B','E'>,<'E','D'>,<'D','A'>}
NOTE: The data set WORK.NODESETOUT has 5 observations and 2 variables.
NOTE: The data set WORK.TSPTOUR has 5 observations and 3 variables.

```

The data set NodeSetOut now contains a sequence of nodes in the optimal tour and is shown in [Figure 9.75](#).

Figure 9.75 Nodes in the Optimal Traveling Salesman Tour

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

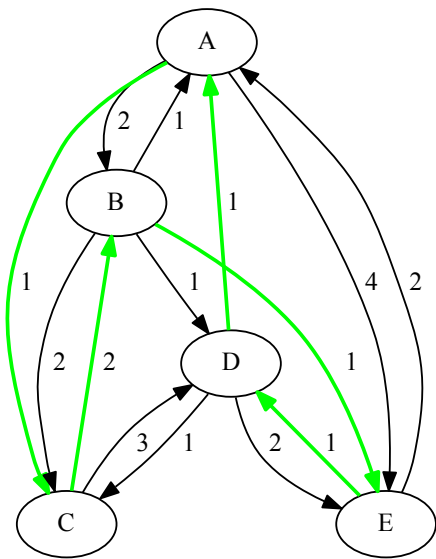
The data set TSPTour now contains the links in the optimal tour and is shown in Figure 9.76.

Figure 9.76 Links in the Optimal Traveling Salesman Tour

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

The minimum-cost links are shown in green in Figure 9.77.

Figure 9.77 Optimal Traveling Salesman Tour



Macro Variable `_OROPTMODEL_`

PROC OPTMODEL always creates and initializes a SAS macro variable called `_OROPTMODEL_`, which contains a character string. After each PROC OPTMODEL run, you can examine this macro variable by specifying `%put &_OROPTMODEL_;` and check the execution of the most recently invoked solver from the value of the macro variable.

Each keyword and value pair in `_OROPTMODEL_` also appears in two other places: the PROC OPTMODEL automatic arrays `_OROPTMODEL_NUM_` and `_OROPTMODEL_STR_`; and the ODS tables ProblemSummary and SolutionSummary, which appear after a SOLVE statement, unless you set the `PRINTLEVEL=` option to NONE. You can use these variables to obtain details about the solution even if you do not specify an output destination in the `OUT=` option.

After the solver is called, the various keywords in the variable are interpreted as follows:

STATUS

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

OK	The solver terminated normally.
SYNTAX_ERROR	The use of syntax is incorrect.
DATA_ERROR	The input data is inconsistent.
OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.
IO_ERROR	A problem in reading or writing of data has occurred.
SEMANTIC_ERROR	An evaluation error, such as an invalid operand type, has occurred.
ERROR	The status cannot be classified into any of the preceding categories.

SOLUTION_STATUS

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

ABORT_NOSOL	The solver was stopped by the user and did not find a solution.
ABORT_SOL	The solver was stopped by the user but still found a solution.
BAD_PROBLEM_TYPE	The problem type is not supported by the solver.
CONDITIONAL_OPTIMAL	The optimality of the solution cannot be proven.
ERROR	The algorithm encountered an error.
FAIL_NOSOL	The solver stopped due to errors and did not find a solution.
FAIL_SOL	The solver stopped due to errors but still found a solution.
FAILED	The solver failed to converge, possibly due to numerical issues.
HEURISTIC_NOSOL	The solver used only heuristics and did not find a solution.
HEURISTIC_SOL	The solver used only heuristics and found a solution.
INFEASIBLE	The problem is infeasible.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
INTERRUPTED	The solver was interrupted by the system or the user before completing its work.
ITERATION_LIMIT_REACHED	The solver reached the maximum number of iterations that is specified in the <code>MAXITER=</code> option.
NODE_LIM_NOSOL	The solver reached the maximum number of nodes specified in the <code>MAXNODES=</code> option and did not find a solution.
NODE_LIM_SOL	The solver reached the maximum number of nodes specified in the <code>MAXNODES=</code> option and found a solution.

NULL_GRAPH	The graph was null (it had 0 nodes) after PROC OPTMODEL processed the SUBGRAPH= option.
OK	The algorithm terminated normally.
OPTIMAL	The solution is optimal.
OPTIMAL_AGAP	The solution is optimal within the absolute gap that is specified in the ABSOBJGAP= option.
OPTIMAL_COND	The solution is optimal, but some infeasibilities (primal, bound, or integer) exceed tolerances because of scaling.
OPTIMAL_RGAP	The solution is optimal within the relative gap that is specified in the RELOBJGAP= option.
OUTMEM_NOSOL	The solver ran out of memory and either did not find a solution or failed to output the solution due to insufficient memory.
OUTMEM_SOL	The solver ran out of memory but still found a solution.
SOLUTION_LIM	The solver reached the maximum number of solutions specified in the MAXCLIQUES= , MAXCYCLES= , or MAXSOLS= option.
TARGET	The solution is not worse than the target that is specified in the TARGET= option.
TIME_LIM_NOSOL	The solver reached the execution time limit specified in the MAXTIME= option and did not find a solution.
TIME_LIM_SOL	The solver reached the execution time limit specified in the MAXTIME= option and found a solution.
TIME_LIMIT_REACHED	The solver reached its execution time limit.
UNBOUNDED	The problem is unbounded.

PROBLEM_TYPE

indicates the type of problem solved. It can take one of the following values:

BICONCOMP	Biconnected components
CLIQUE	Maximal cliques
CONCOMP	Connected components
CYCLE	Cycle detection
LAP	Linear assignment (matching)
MCF	Minimum-cost network flow
MINCUT	Minimum cut
MST	Minimum spanning tree
SHORTPATH	Shortest path
TRANSCL	Transitive closure
TSP	Traveling salesman
NONE	This value is used when you do not specify an algorithm to run.

OBJECTIVE

indicates the objective value that is obtained by the solver at termination. For problem classes that do not have an explicit objective, such as cycle, the value of this keyword within the _OROPTMODEL_ macro variable is missing (.).

RELATIVE_GAP

indicates the relative gap between the best integer objective (BestInteger) and the best bound on the objective function value (BestBound) upon termination of the MILP solver. The relative gap is equal to

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

ABSOLUTE_GAP

indicates the absolute gap between the best integer objective (BestInteger) and the best bound on the objective function value (BestBound) upon termination of the MILP solver. The absolute gap is equal to $| \text{BestInteger} - \text{BestBound} |$.

PRIMAL_INFEASIBILITY

indicates the maximum (absolute) violation of the primal constraints by the solution.

BOUND_INFEASIBILITY

indicates the maximum (absolute) violation by the solution of the lower or upper bounds (or both).

INTEGER_INFEASIBILITY

indicates the maximum (absolute) violation of the integrality of integer variables returned by the MILP solver.

BEST_BOUND

indicates the best bound on the objective function value at termination. A missing value indicates that the MILP solver was not able to obtain such a bound.

NODES

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

ITERATIONS

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

PRESOLVE_TIME

indicates the time (in seconds) used in preprocessing.

SOLUTION_TIME

indicates the time (in seconds) taken to solve the problem, including preprocessing time.

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

When SOLUTION_STATUS has a value of OPTIMAL, CONDITIONAL_OPTIMAL, ITERATION_LIMIT_REACHED, or TIME_LIMIT_REACHED, all terms of the _OROPTMODEL_ macro variable are present; for other values of SOLUTION_STATUS, some terms do not appear.

The following keywords within the _OROPTMODEL_ macro variable appear only with certain algorithms. The keywords convey information about the number of solutions each algorithm found:

NUM_ARTICULATION_POINTS

indicates the number of articulation points found. This term appears only for biconnected components.

NUM_CLIQUES

indicates the number of cliques found. This term appears only for clique.

NUM_COMPONENTS

indicates the number of components that match the definitions of the corresponding problem class. This term appears only for connected components and biconnected components.

NUM_CYCLES

indicates the number of cycles found that satisfy the criteria you provide. This term appears only for cycles.

Examples: Network Solver

Example 9.1: Articulation Points in a Terrorist Network

This example considers the terrorist communications network from the attacks on the United States on September 11, 2001, described in Krebs 2002. [Figure 9.78](#) shows this network, which was constructed after the attacks, based on collected intelligence information.

Figure 9.78 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 option in this context:

```
data LinkSetInTerror911;
  input from & $32. to & $32.;
  datalines;
Abu Zubeida          Djamal Beghal
Jean-Marc Grandvisir Djamal Beghal
Nizar Trabelsi       Djamal Beghal
Abu Walid             Djamal Beghal
Abu Qatada            Djamal Beghal
Zacarias Moussaoui   Djamal Beghal
Jerome Courtaillier  Djamal Beghal
Kamel Daoudi         Djamal Beghal
Abu Walid            Kamel Daoudi
Abu Walid            Abu Qatada
Kamel Daoudi         Zacarias Moussaoui
Kamel Daoudi         Jerome Courtaillier
```

```

... more lines ...

Nawaf Alhazmi           Khalid Al-Mihdhar
Osama Awadallah          Khalid Al-Mihdhar
Abdussattar Shaikh       Khalid Al-Mihdhar
Abdussattar Shaikh       Osama Awadallah
;

```

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 optmodel;
  set<str,str> LINKS;
  read data LinkSetInTerror911 into LINKS=[from to];
  set NODES = union{<i,j> in LINKS} {i,j};
  set<str> ARTPOINTS;

  solve with NETWORK /
    links      = (include=LINKS)
    biconcomp
    out        = (artpoints=ARTPOINTS)
  ;

  put ARTPOINTS;
  create data ArtPoints from [node]=ARTPOINTS artpoint=1;
quit;

```

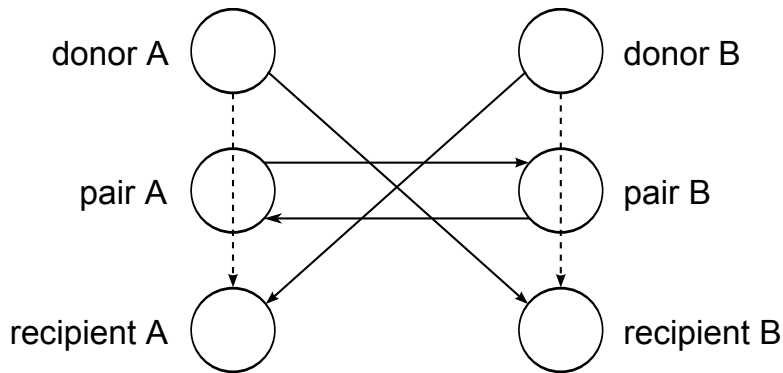
The data set ArtPoints contains members of the network who are articulation points. Focusing investigations on cutting off these particular members could have caused a great deal of disruption in the terrorists' ability to communicate when formulating the attack.

Output 9.1.1 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 9.2: Cycle Detection for Kidney Donor Exchange

This example looks at an application of cycle detection to help create a kidney donor exchange. Suppose someone needs a kidney transplant and a family member is willing to donate one. If the donor and recipient are incompatible (because of blood types, tissue mismatch, and so on), the transplant cannot happen. Now suppose two donor-recipient pairs A and B are in this situation, but donor A is compatible with recipient B and donor B is compatible with recipient A. Then two transplants can take place in a two-way swap, shown graphically in Figure 9.79. More generally, an n -way swap can be performed involving n donors and n recipients (Willingham 2009).

Figure 9.79 Kidney Donor Exchange Two-Way Swap

To model this problem, define a directed graph as follows. Each node is an incompatible donor-recipient pair. Link (i, j) exists if the donor from node i is compatible with the recipient from node j . The link weight is a measure of the quality of the match. By introducing dummy links whose weight is 0, you can also include altruistic donors who have no recipients or recipients who have no donors. The idea is to find a maximum-weight node-disjoint union of directed cycles. You want the union to be node-disjoint so that no kidney is donated more than once, and you want cycles so that the donor from node i gives up a kidney if and only if the recipient from node i receives a kidney.

Without any other constraints, the problem could be solved as a linear assignment problem, as described in the section “[Linear Assignment \(Matching\)](#)” on page 423. But doing so would allow arbitrarily long cycles in the solution. Because of practical considerations (such as travel) and to mitigate risk, each cycle must have no more than L links. The kidney exchange problem is to find a maximum-weight node-disjoint union of short directed cycles.

One way to solve this problem is to explicitly generate all cycles whose length is at most L and then solve a set packing problem. You can use PROC OPTMODEL to generate the cycles, formulate the set packing problem, call the mixed integer linear programming solver, and output the optimal solution.

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

```
/* 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 declare parameters and then read the input data:

```
%let max_length = 10;
proc optmodel;
  /* declare index sets and parameters, and read data */
  set <num,num> ARCS;
  num weight {ARCS};
  read data LinkSetIn into ARCS=[from to] weight;
  set<num,num,num> ID_ORDER_NODE;
```

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

```
/* generate all cycles with 2 <= length <= max_length */
solve with NETWORK /
  loglevel          = moderate
  graph_direction   = directed
  links             = (include=ARCS)
  cycle             = (mode=all_cycles minlength=2 maxlength=&max_length)
  out               = (cycles=ID_ORDER_NODE)
;
```

The network solver finds 224 cycles of the appropriate length, as shown in [Output 9.2.1](#).

Output 9.2.1 Cycles for Kidney Donor Exchange Network Solver Log

```
NOTE: There were 194 observations read from the data set WORK.LINKSETIN.
NOTE: The number of nodes in the input graph is 97.
NOTE: The number of links in the input graph is 194.
NOTE: Processing cycle detection.
NOTE: The graph has 224 cycles.
NOTE: Processing cycle detection used 0.31 (cpu: 0.31) seconds.
```

From the resulting set ID_ORDER_NODE, use the following statements to convert to one tuple per cycle-arc combination:

```
/* extract <cid,from,to> triples from <cid,order,node> triples */
set <num,num,num> ID_FROM_TO init {};
num last init ., from, to;
for {<cid,order,node> in ID_ORDER_NODE} do;
  from = last;
  to   = node;
  last = to;
  if order ne 1 then ID_FROM_TO = ID_FROM_TO union {<cid,from,to>};
end;
```

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

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

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 */
set CYCLES = setof {<c,i,j> in ID_FROM_TO} c;
set ARCS_c {c in CYCLES} = setof {<(c),i,j> in ID_FROM_TO} <i,j>;
set NODES_c {c in CYCLES} = union {<i,j> in ARCS_c[c]} {i,j};
set NODES = union {c in CYCLES} NODES_c[c];
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 9.2.2](#), it was able to find a total weight (quality level) of 26.02.

Output 9.2.2 Cycles for Kidney Donor Exchange MILP Solver Log

```

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 46 variables and 35 constraints.
NOTE: The MILP presolver removed 901 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 178 variables, 28 constraints, and 999
      constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.
      Node  Active    Sols  BestInteger  BestBound    Gap    Time
          0         1      3    22.7780692   868.9019355  97.38%    0
          0         1      3    22.7780692   26.7803921  14.94%    0
          0         1      4    23.2747070   26.0966646  10.81%    0
          0         1      4    23.2747070   26.0202906  10.55%    0
          0         1      5    26.0202875   26.0202875   0.00%    0
NOTE: The MILP solver added 8 cuts with 501 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 26.020287528.
NOTE: The data set WORK.SOLUTION has 6 observations and 2 variables.
STATUS=OK ALGORITHM=BAC SOLUTION_STATUS=OPTIMAL OBJECTIVE=26.020287528
RELATIVE_GAP=0 ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=1.110223E-15
BOUND_INFEASIBILITY=6E-7 INTEGER_INFEASIBILITY=2.8E-6 BEST_BOUND=26.020287528
NODES=1 ITERATIONS=142 PRESOLVE_TIME=0.00 SOLUTION_TIME=0.01

```

The data set Solution, shown in [Output 9.2.3](#), now contains the cycles that define the best exchange and their associated weight (quality).

Output 9.2.3 Maximum Quality Solution for Kidney Donor Exchange

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

Example 9.3: Linear Assignment Problem for Minimizing Swim Times

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

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


```

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

```

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

```

proc contents data=RelayTimes
  out=stroke_data(rename=(name=stroke) where=(type=1));
run;

proc optmodel;
  set <str> STROKES;
  read data stroke_data into STROKES=[stroke];
  set <str> SWIMMERS;
  str sex {SWIMMERS};
  num time {SWIMMERS, STROKES};
  read data RelayTimes into SWIMMERS=[name] sex
    {stroke in STROKES} <time[name,stroke]=col(stroke)>;
  set FEMALES = {i in SWIMMERS: sex[i] = 'F'};
  set FNODES = FEMALES union STROKES;
  set MALES = {i in SWIMMERS: sex[i] = 'M'};
  set MNODES = MALES union STROKES;
  set <str,str> PAIRS;

  solve with NETWORK /
    graph_direction = directed
    links           = (weight=time)
    subgraph        = (nodes=FNODES)
    lap
    out              = (assignments=PAIRS)
  ;
  put PAIRS;
  create data LinearAssignF from [name assign]=PAIRS sex[name] cost=time;

  solve with NETWORK /
    graph_direction = directed
    links           = (weight=time)
    subgraph        = (nodes=MNODES)
    lap
    out              = (assignments=PAIRS)
  ;
  put PAIRS;
  create data LinearAssignM from [name assign]=PAIRS sex[name] cost=time;
quit;

```

The progress of the two SOLVE WITH NETWORK calls is shown in [Output 9.3.1](#).

Output 9.3.1 Network Solver Log: Linear Assignment for Swim Times

```
NOTE: The data set WORK.STROKE_DATA has 4 observations and 41 variables.
NOTE: There were 4 observations read from the data set WORK.STROKE_DATA.
NOTE: There were 10 observations read from the data set WORK.RELAYTIMES.
NOTE: The SUBGRAPH= option filtered 16 elements from 'time.'
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 24.
NOTE: Processing the linear assignment problem.
NOTE: Objective = 111.5.
NOTE: Processing the linear assignment problem used 0.00 (cpu: 0.00) seconds.
{<'Karen','breast'>,<'Jan','fly'>,<'Carol','free'>,<'Ellen','back'>}
NOTE: The data set WORK.LINEARASSIGNF has 4 observations and 4 variables.
NOTE: The SUBGRAPH= option filtered 24 elements from 'time.'
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 16.
NOTE: Processing the linear assignment problem.
NOTE: Objective = 96.6.
NOTE: Processing the linear assignment problem used 0.00 (cpu: 0.00) seconds.
{<'Jim','free'>,<'Mike','breast'>,<'Sam','back'>,<'Clayton','fly'>}
NOTE: The data set WORK.LINEARASSIGNM has 4 observations and 4 variables.
```

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

Output 9.3.2 Optimal Assignments for Best Female Swim Times

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

Output 9.3.3 Optimal Assignments for Best Male Swim Times

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

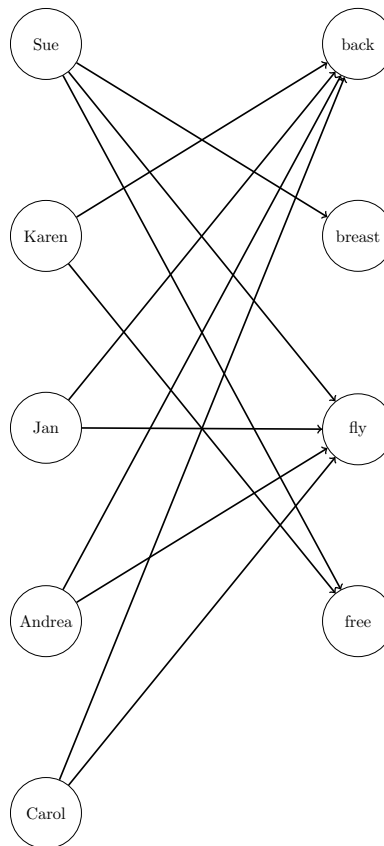
Example 9.4: Linear Assignment Problem, Sparse Format versus Dense Format

This example looks at the problem of assigning swimmers to strokes based on their best times. However, in this case certain swimmers are not eligible to perform certain strokes. A missing (.) value in the data matrix identifies an ineligible assignment. For example:

```
data RelayTimesMatrix;
  input name $ sex $ back breast fly free;
  datalines;
Sue      F      .  36.7 28.3 36.1
Karen    F 34.6      .      . 26.2
Jan      F 31.3      . 27.1      .
Andrea   F 28.6      . 29.1      .
Carol    F 32.9      . 26.6      .
;
```

Recall that the linear assignment problem can also be interpreted as the minimum-weight matching in a bipartite directed graph. The eligible assignments define links between the rows (swimmers) and the columns (strokes), as in Figure 9.80.

Figure 9.80 Bipartite Graph for Linear Assignment Problem



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

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

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

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

```
proc contents data=RelayTimesMatrix
  out=stroke_data(rename=(name=stroke) where=(type=1));
run;

proc optmodel;
  set <str> STROKES;
  read data stroke_data into STROKES=[stroke];
  set <str> SWIMMERS;
  str sex {SWIMMERS};
  num time {SWIMMERS, STROKES};
  read data RelayTimesMatrix into SWIMMERS=[name]
    sex
    {stroke in STROKES} <time[name,stroke]=col(stroke)>;
  set SWIMMERS_STROKES =
    {name in SWIMMERS, stroke in STROKES: time[name,stroke] ne .};
  set <str,str> PAIRS;

  solve with NETWORK /
    graph_direction = directed
    links           = (weight=time)
    subgraph        = (links=SWIMMERS_STROKES)
    lap
    out             = (assignments=PAIRS)
  ;

  put PAIRS;
  create data LinearAssignMatrix from [name assign]=PAIRS
    sex[name] cost=time;
quit;

proc sql;
  create table stroke_data as
  select distinct attr as stroke
```

```

    from RelayTimesLinks;
quit;

proc optmodel;
    set <str> STROKES;
    read data stroke_data into STROKES=[stroke];
    set <str> SWIMMERS;
    str sex {SWIMMERS};
    set <str,str> SWIMMERS_STROKES;
    num time {SWIMMERS_STROKES};
    read data RelayTimesLinks into SWIMMERS_STROKES=[name attr] time=cost;
    set <str,str> PAIRS;

    solve with NETWORK /
        graph_direction = directed
        links            = (weight=time)
        lap
        out              = (assignments=PAIRS)
    ;

    put PAIRS;
    create data LinearAssignLinks from [name attr]=PAIRS cost=time;
quit;

```

The data sets LinearAssignMatrix and LinearAssignLinks now contain the optimal assignments, as shown in [Output 9.4.1](#) and [Output 9.4.2](#).

Output 9.4.1 Optimal Assignments for Swim Times (Dense Input)

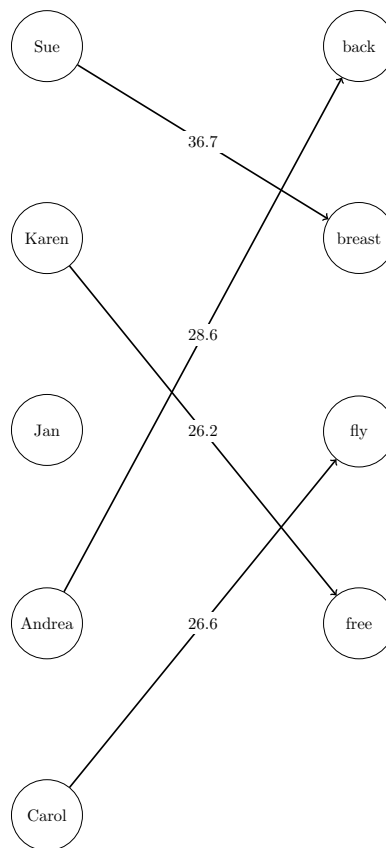
name	assign	sex	cost
Sue	breast	F	36.7
Karen	free	F	26.2
Andrea	back	F	28.6
Carol	fly	F	26.6
			118.1

Output 9.4.2 Optimal Assignments for Swim Times (Sparse Input)

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

The optimal assignments are shown graphically in Figure 9.81.

Figure 9.81 Optimal Assignments for Swim Times

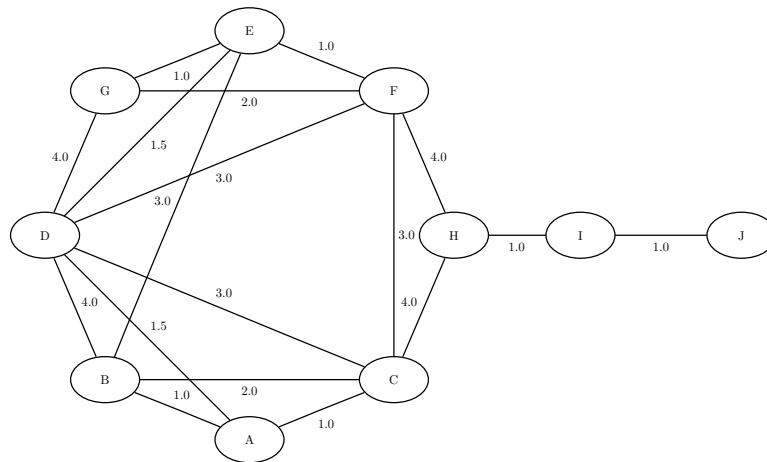


For large problems where a number of links are forbidden, the sparse format can be faster and can save a great deal of memory. Consider an example that uses the dense format with 15,000 columns ($|S| = 15,000$) and 4,000 rows ($|T| = 4,000$). To store the dense matrix in memory, the network solver needs to allocate approximately $|S| \cdot |T| \cdot 8/1024/1024 = 457$ MB. If the data have mostly ineligible links, then the sparse (graph) format is much more efficient with respect to memory. For example, if the data have only 5% of the eligible links ($15,000 \cdot 4,000 \cdot 0.05 = 3,000,000$), 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 is more efficient.

Example 9.5: Minimum Spanning Tree for Computer Network Topology

Consider the problem of designing a small network of computers in an office. In designing the network, the goal is to make sure that each machine in the office can reach every other machine. To accomplish this goal, Ethernet lines must be constructed and run between the machines. The construction costs for each possible link are based approximately on distance and are shown in Figure 9.82. Besides distance, the costs also reflect some restrictions due to physical boundaries. To connect all the machines in the office at minimal cost, you need to find a minimum spanning tree on the network of possible links.

Figure 9.82 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:

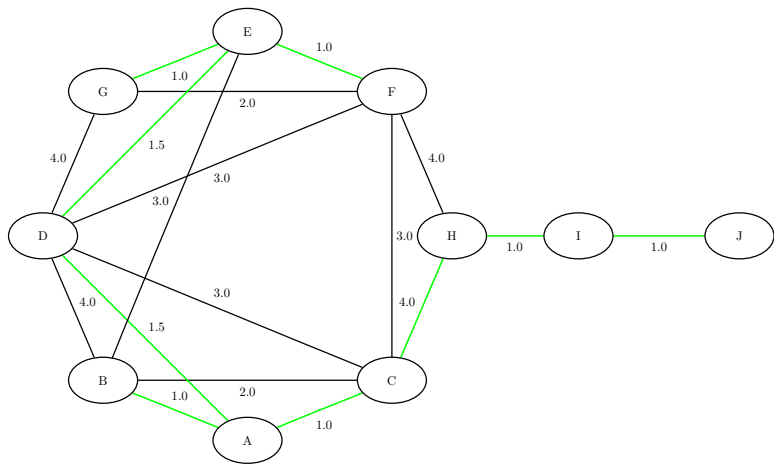
```
proc optmodel;
  set<str,str> LINKS;
  num weight{LINKS};
  read data LinkSetInCompNet into LINKS=[from to] weight;
  set<str,str> FOREST;

  solve with NETWORK /
    links      = (weight=weight)
    minspantree
    out        = (forest=FOREST)
  ;

  put FOREST;
  put (sum {<i,j> in FOREST} weight[i,j]);
  create data MinSpanTree from [from to]=FOREST weight;
quit;
```

Output 9.5.1 shows the resulting data set MinSpanTree, which is displayed graphically in Figure 9.83 with the minimal cost links shown in green.

Figure 9.83 Minimum Spanning Tree for Office Computer Network



Output 9.5.1 Minimum Spanning Tree of a Computer Network Topology

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

Example 9.6: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System

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

The bug tracking system needs to check for two situations when users declare a bug to be a DUP. The first situation is called a *circular dependence*. Consider bugs A, B, C, and D in the tracking system. The first user declares that A is a DUP of B and that C is a DUP of D. Then, a second user declares that B is a DUP of C, and a third user declares that D is a DUP of A. You now have a circular dependence, and no primary bug is defined on which the development team should focus. You can easily see this circular dependence in the graph representation, because $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$. Finding such circular dependencies can be done using cycle detection, which is described in the section “Cycle” on page 418. However, the second situation

that needs to be checked is more general. If a user declares that A is a DUP of B and another user declares that B is a DUP of C, this chain of duplicates is already an issue. The bug tracking system needs to provide one primary bug to which the rest of the bugs are duplicated. The existence of these chains can be identified by calculating the transitive closure of the directed graph that is defined by the DUP links.

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

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

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

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

```
proc optmodel;
  set<str,str> LINKS;
  read data DefectLinks into LINKS=[defectId linkedDefect];
  set<num,num,str> CYCLES;
  set<str,str> CLOSURE;

  solve with NETWORK /
    loglevel          = moderate
    graph_direction   = directed
    links              = (include=LINKS)
    cycle              = (mode=first_cycle)
    out                = (cycles=CYCLES)
  ;

  put CYCLES;
  create data Cycles from [cycle order node]=CYCLES;
```

```

solve with NETWORK /
  loglevel          = moderate
  graph_direction   = directed
  links             = (include=LINKS)
  transitive_closure
  out               = (closure=CLOSURE)
;

put CLOSURE;
create data TransClosure from [defectId linkedDefect]=CLOSURE;
quit;

proc sql;
  create table Chains as
  select defectId, linkedDefect from TransClosure
  except
  select defectId, linkedDefect from DefectLinks;
quit;

```

The progress of the procedure is shown in [Output 9.6.1](#).

Output 9.6.1 Network Solver Log: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System

```

NOTE: There were 12 observations read from the data set WORK.DEFECTLINKS.
NOTE: The number of nodes in the input graph is 16.
NOTE: The number of links in the input graph is 12.
NOTE: Processing cycle detection.
NOTE: The graph does have a cycle.
NOTE: Processing cycle detection used 0.00 (cpu: 0.00) seconds.
{<1,1,'S0152674'>,<1,2,'S0153280'>,<1,3,'S0153307'>,<1,4,'S0152674'>}
NOTE: The data set WORK.CYCLES has 4 observations and 3 variables.
NOTE: The number of nodes in the input graph is 16.
NOTE: The number of links in the input graph is 12.
NOTE: Processing the transitive closure.
NOTE: Processing the transitive closure used 0.00 (cpu: 0.00) seconds.
{<'D0096978','S0711218'>,<'S0152674','S0153280'>,<'S0153280','S0153307'>,<
'S0153307','S0152674'>,<'S0162973','S0162978'>,<'S0162978','S0165405'>,<
'S0325026','S0575748'>,<'S0347945','S0346582'>,<'S0350596','S0346582'>,<
'S0539744','S0643230'>,<'S0575748','S0643230'>,<'S0629984','S0643230'>,<
'S0153307','S0153280'>,<'S0152674','S0153307'>,<'S0153307','S0153307'>,<
'S0152674','S0152674'>,<'S0153280','S0152674'>,<'S0153280','S0153280'>,<
'S0162973','S0165405'>,<'S0325026','S0643230'>}
NOTE: The data set WORK.TRANSCLCLOSURE has 20 observations and 2 variables.
NOTE: Table WORK.CHAINS created, with 8 rows and 2 columns.

```

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

Output 9.6.2 Cycle in Bug Tracking System

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

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

Output 9.6.3 Chains in Bug Tracking System

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

Example 9.7: Traveling Salesman Tour through US Capital Cities

Consider a cross-country trip where you want to travel the fewest miles to visit all of the capital cities in all US states except Alaska and Hawaii. Finding the optimal route is an instance of the traveling salesman problem, which is described in section “[Traveling Salesman Problem](#)” on page 452.

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 OPTMODEL statements find the optimal tour through each of the capital cities:

```
/* Find optimal tour by using the network solver */
proc optmodel;
  set<str,str> CAPPAIRS;
  set<str> CAPITALS = union {<i,j> in CAPPAIRS} {i,j};
  num distance{i in CAPITALS, j in CAPITALS: i < j};
  read data CitiesDist into CAPPAIRS=[city1 city2] distance;
  set<str,str> TOUR;
  num order{CAPITALS};

  solve with NETWORK /
    loglevel = moderate
    links = (weight=distance)
    tsp
    out = (order=order tour=TOUR)
  ;

  put (sum{<i,j> in TOUR} distance[i,j]);
  /* Create tour-ordered pairs (rather than input-ordered pairs) */
  str CAPbyOrder{1..card(CAPITALS)};
  for {i in CAPITALS} CAPbyOrder[order[i]] = i;
  set TSPEDGES init
    setof{i in 2..card(CAPITALS)} <CAPbyOrder[i-1],CAPbyOrder[i]>
    union {<CAPbyOrder[card(CAPITALS)],CAPbyOrder[1]>};
  num distance2{<i,j> in TSPEDGES} =
    if i < j then distance[i,j] else distance[j,i];
  create data TSPTourNodes from [node] tsp_order=order;
  create data TSPTourLinks from [city1 city2]=TSPEDGES distance=distance2;
quit;
```

The progress of the procedure is shown in [Output 9.7.1](#). The total mileage needed to optimally traverse the capital cities is 10,627.75 miles.

Output 9.7.1 Network Solver Log: Traveling Salesman Tour through US Capital Cities

NOTE: There were 1176 observations read from the data set WORK.CITIESDIST.

NOTE: The number of nodes in the input graph is 49.

NOTE: The number of links in the input graph is 1176.

NOTE: Processing the traveling salesman problem.

NOTE: The initial TSP heuristics found a tour with cost 10645.918753 using 0.09 (cpu: 0.06) seconds.

NOTE: The MILP presolver value NONE is applied.

NOTE: The MILP solver is called.

NOTE: The Branch and Cut algorithm is used.

Node	Active	Sols	BestInteger	BestBound	Gap	Time
0	1	1	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	10350.0790852	2.86%	0
0	1	1	10645.9187534	10381.4538838	2.55%	0
0	1	1	10645.9187534	10470.6122886	1.67%	0
0	1	1	10645.9187534	10492.6949171	1.46%	0
0	1	1	10645.9187534	10519.2912679	1.20%	0
0	1	1	10645.9187534	10576.0823291	0.66%	0
0	1	1	10645.9187534	10590.9748294	0.52%	0
0	1	1	10645.9187534	10607.8528157	0.36%	0
0	1	1	10645.9187534	10607.8528157	0.36%	0

NOTE: The MILP solver added 12 cuts with 3277 cut coefficients at the root.

NOTE: Objective of the best integer solution found = 10645.918753.

NOTE: Processing the traveling salesman problem used 0.10 (cpu: 0.08) seconds.

NOTE: The TSP solver is restarting using a reduced graph with 49 nodes and 107 links.

NOTE: Processing the traveling salesman problem.

NOTE: The initial TSP heuristics found a tour with cost 10645.918753 using 0.00 (cpu: 0.00) seconds.

NOTE: The MILP presolver value NONE is applied.

NOTE: The MILP solver is called.

NOTE: The Branch and Cut algorithm is used.

Node	Active	Sols	BestInteger	BestBound	Gap	Time
0	1	1	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	10350.0790852	2.86%	0
0	1	1	10645.9187534	10381.4538838	2.55%	0
0	1	1	10645.9187534	10470.6122886	1.67%	0
0	1	1	10645.9187534	10497.1240512	1.42%	0
0	1	1	10645.9187534	10519.2912679	1.20%	0
0	1	1	10645.9187534	10576.0823291	0.66%	0
0	1	1	10645.9187534	10590.9748294	0.52%	0
0	1	1	10645.9187534	10607.8528157	0.36%	0
0	1	3	10645.9187534	10607.8528157	0.36%	0

NOTE: The MILP solver added 10 cuts with 73 cut coefficients at the root.

2	1	7	10627.7543183	10623.5771454	0.04%	0
4	0	7	10627.7543183	10627.7543183	0.00%	0

Output 9.7.1 *continued*

```
NOTE: Optimal.
NOTE: Objective = 10627.754318.
NOTE: Processing the traveling salesman problem used 0.01 (cpu: 0.00) seconds.
10627.754318
NOTE: The data set WORK.TSPTOURNODES has 49 observations and 2 variables.
NOTE: The data set WORK.TSPTOURLINKS has 49 observations and 3 variables.
```

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;

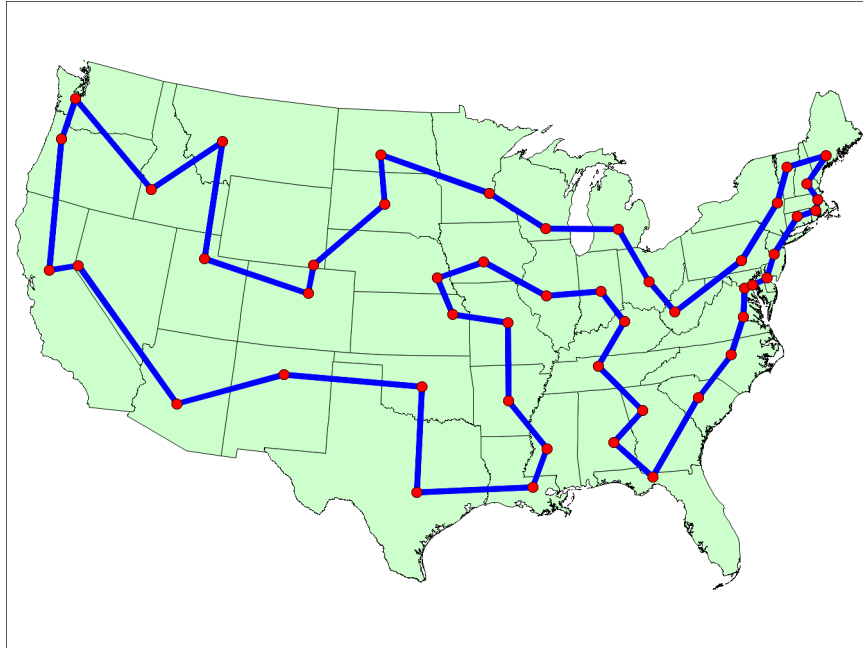
/* Create the final annotate data set */
data anno_dots;
    set anno_dots;
    length function color $8;
    xsys='2'; ysys='2'; when='a'; hsys='3';
    function='pie';
    rotate=360; size=0.8; style='psolid'; color="red";
    output;
    style='pempty'; color="black";
    output;
run;

/* Generate the map with GMAP */
pattern1 v=s c=cxcccffcc 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 [Output 9.7.2](#).

Output 9.7.2 Optimal Traveling Salesman Tour through US Capital Cities



The data set `TSPTourLinks` contains the links in the optimal tour. To display the links in the order they are to be visited, you can use the following DATA step:

```
/* Create the directed optimal tour */
data TSPTourLinksDirected(drop=next);
  set TSPTourLinks;
  retain next;
  if _N_ ne 1 and city1 ne next then do;
    city2 = city1;
    city1 = next;
  end;
  next = city2;
run;
```


The data set TSPTourLinksDirected is shown in Figure 9.84.

Figure 9.84 Links in the Optimal Traveling Salesman Tour

city1	city2	distance	city1	city2	distance
Montgomery	Tallahassee	177.14	Denver	Salt Lake City	373.05
Tallahassee	Columbia	311.23	Salt Lake City	Helena	403.40
Columbia	Raleigh	182.99	Helena	Boise City	291.20
Raleigh	Richmond	135.58	Boise City	Olympia	401.31
Richmond	Washington	97.96	Olympia	Salem	146.00
Washington	Annapolis	27.89	Salem	Sacramento	447.40
Annapolis	Dover	54.01	Sacramento	Carson City	101.51
Dover	Trenton	83.88	Carson City	Phoenix	577.84
Trenton	Hartford	151.65	Phoenix	Santa Fe	378.27
Hartford	Providence	65.56	Santa Fe	Oklahoma City	474.92
Providence	Boston	38.41	Oklahoma City	Austin	357.38
Boston	Concord	66.30	Austin	Baton Rouge	394.78
Concord	Augusta	117.36	Baton Rouge	Jackson	139.75
Augusta	Montpelier	139.32	Jackson	Little Rock	206.87
Montpelier	Albany	126.19	Little Rock	Jefferson City	264.75
Albany	Harrisburg	230.24	Jefferson City	Topeka	191.67
Harrisburg	Charleston	287.34	Topeka	Lincoln	132.94
Charleston	Columbus	134.64	Lincoln	Des Moines	168.10
Columbus	Lansing	205.08	Des Moines	Springfield	243.02
Lansing	Madison	246.88	Springfield	Indianapolis	186.46
Madison	Saint Paul	226.25	Indianapolis	Frankfort	129.90
Saint Paul	Bismarck	391.25	Frankfort	Nashville-Davidson	175.58
Bismarck	Pierre	170.27	Nashville-Davidson	Atlanta	212.61
Pierre	Cheyenne	317.90	Atlanta	Montgomery	145.39
Cheyenne	Denver	98.33			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, NJ: Princeton University Press.
- Bron, C., and Kerbosch, J. (1973). "Algorithm 457: Finding All Cliques of an Undirected Graph." *Communications of the ACM* 16:48–50.
- Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms*. Cambridge, MA, and New York: MIT Press and McGraw-Hill.
- Google (2011). "Google Maps." Accessed March 16, 2011. <http://maps.google.com>.
- Harley, E. R. (2003). *Graph Algorithms for Assembling Integrated Genome Maps*. Ph.D. diss., University of Toronto.

- Johnson, D. B. (1975). "Finding All the Elementary Circuits of a Directed Graph." *SIAM Journal on Computing* 4:77–84.
- Jonker, R., and Volgenant, A. (1987). "A Shortest Augmenting Path Algorithm for Dense and Sparse Linear Assignment Problems." *Computing* 38:325–340.
- Krebs, V. (2002). "Uncloaking Terrorist Networks." *First Monday* 7. http://www.firstmonday.org/issues/issue7_4/krebs/.
- Kruskal, J. B. (1956). "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem." *Proceedings of the American Mathematical Society* 7:48–50.
- Kumar, R., and Li, H. (1994). "On Asymmetric TSP: Transformation to Symmetric TSP and Performance Bound." <http://home.eng.iastate.edu/~rkumar/PUBS/atsp.pdf>.
- Stoer, M., and Wagner, F. (1997). "A Simple Min-Cut Algorithm." *Journal of the Association for Computing Machinery* 44:585–591.
- Tarjan, R. E. (1972). "Depth-First Search and Linear Graph Algorithms." *SIAM Journal on Computing* 1:146–160.
- Willingham, V. (2009). "Massive Transplant Effort Pairs 13 Kidneys to 13 Patients." CNN Health. Accessed March 16, 2011. <http://www.cnn.com/2009/HEALTH/12/14/kidney.transplant/index.html>.

Subject Index

graph theory and network analysis, [376](#)

macro variable

`_OROPTMODEL_`, [458](#)

network solver

 overview, [376](#)

OPTMODEL procedure, network solver

 macro variable `_OROPTMODEL_`, [458](#)

`_OROPTMODEL_` macro variable, [458](#)

overview

 network solver, [376](#)

Syntax Index

- ABSOBJGAP= suboption
 - TSP= option, [394](#)
- ALGORITHM= suboption
 - CONCOMP= option, [391](#)
- ARTPOINTS= suboption
 - OUT= option, [388](#)
- ASSIGNMENTS= suboption
 - OUT= option, [389](#)
- BICONCOMP option
 - algorithm options, [390](#)
- BICONCOMP= suboption
 - OUT= option, [389](#)
- CLIQUE= option
 - algorithm options, [391](#)
- CLIQUE= suboption
 - OUT= option, [389](#)
- CONCOMP= option
 - algorithm options, [391](#)
- CONCOMP= suboption
 - OUT= option, [389](#)
- CONFLICTSEARCH= suboption
 - TSP= option, [394](#)
- CUTOFF= suboption
 - TSP= option, [395](#)
- CUTSETS= suboption
 - OUT= option, [389](#)
- CUTSTRATEGY= suboption
 - TSP= option, [395](#)
- CYCLE= option
 - algorithm options, [391](#)
- CYCLES= suboption
 - OUT= option, [389](#)
- EMPHASIS= suboption
 - TSP= option, [395](#)
- FLOW= suboption
 - OUT= option, [389](#)
- FOREST= suboption
 - OUT= option, [389](#)
- GRAPH_DIRECTION= option
- SOLVE WITH NETWORK statement, [386](#)
- HEURISTICS= suboption
 - TSP= option, [396](#)
- INCLUDE= suboption
 - LINKS= option, [387](#)
 - NODES= option, [388](#)
- INCLUDE_SELFLINK option
 - SOLVE WITH NETWORK statement, [386](#)
- LINEAR_ASSIGNMENT option
 - algorithm options, [392](#)
- LINKS= option, [387](#)
- LINKS= suboption
 - OUT= option, [389](#)
 - SUBGRAPH= option, [390](#)
- LOGFREQ= option
 - SOLVE WITH NETWORK statement, [386](#)
- LOGLEVEL= option
 - SOLVE WITH NETWORK statement, [386](#)
- LOWER= suboption
 - LINKS= option, [388](#)
- MAXCLIQUE= suboption
 - CLIQUE= option, [391](#)
- MAXCYCLES= suboption
 - CYCLE= option, [391](#)
- MAXLENGTH= suboption
 - CYCLE= option, [391](#)
- MAXLINKWEIGHT= suboption
 - CYCLE= option, [392](#)
- MAXNODES= suboption
 - TSP= option, [396](#)
- MAXNODEWEIGHT= suboption
 - CYCLE= option, [392](#)
- MAXNUMCUTS= suboption
 - MINCUT= option, [393](#)
- MAXSOLS= suboption
 - TSP= option, [396](#)
- MAXTIME= option
 - SOLVE WITH NETWORK statement, [387](#)
- MAXWEIGHT= suboption
 - MINCUT= option, [393](#)

MILP= suboption
 TSP= option, 396
 MINCOSTFLOW option
 algorithm options, 393
 MINCUT= option
 algorithm options, 393
 MINLENGTH= suboption
 CYCLE= option, 392
 MINLINKWEIGHT= suboption
 CYCLE= option, 392
 MINNODEWEIGHT= suboption
 CYCLE= option, 392
 MINSPANTREE= option
 algorithm options, 393
 MODE= suboption
 CYCLE= option, 392

 network solver, 381
 NODES= option
 SOLVE WITH NETWORK statement, 388
 NODES= suboption
 OUT= option, 389
 SUBGRAPH= option, 390
 NODESEL= suboption
 TSP= option, 397

 ORDER= suboption
 OUT= option, 390
 OUT= option
 SOLVE WITH NETWORK statement, 388

 PARTITIONS= suboption
 OUT= option, 390
 PATHS= suboption
 SHORTPATH=option, 393
 PROBE= suboption
 TSP= option, 397

 RELOBJGAP= suboption
 TSP= option, 397

 SHORTPATH= option
 algorithm options, 393
 SINK= suboption
 SHORTPATH= option, 394
 SOLVE WITH NETWORK statement
 statement options, 386
 SOURCE= suboption
 SHORTPATH= option, 394
 SPPATHS= suboption
 OUT= option, 390
 SPWEIGHTS= suboption
 OUT= option, 390
 STRONGITER= suboption
 TSP= option, 397

 STRONGLEN= suboption
 TSP= option, 398
 SUBGRAPH= option
 SOLVE WITH NETWORK statement, 390

 TARGET= suboption
 TSP= option, 398
 TIMETYPE= option
 SOLVE WITH NETWORK statement, 387
 TOUR= suboption
 OUT= option, 390
 TRANSCL= suboption
 OUT= option, 390
 TRANSITIVE_CLOSURE option
 algorithm options, 394
 TSP option
 algorithm options, 394

 UPPER= suboption
 LINKS= option, 388
 USEWEIGHT= suboption
 SHORTPATH= option, 394

 VARSEL= suboption
 TSP= option, 398

 WEIGHT2= suboption
 NODES= option, 388
 WEIGHT= suboption
 LINKS= option, 388
 NODES= option, 388