SAS/OR® 12.3 User's Guide
Network Optimization Algorithms

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2013. *SAS/OR® 12.3 User's Guide: Network Optimization Algorithms*. Cary, NC: SAS Institute Inc.

SAS/OR® 12.3 User's Guide: Network Optimization Algorithms

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

July 2013

Contents

Credits

Documentation

Writing	Matthew Galati
Editing	Anne Baxter
Documentation Support	Tim Arnold, Melanie Gratton, Daniel Underwood
Technical Review	Manoj Chari, Charles B. Kelly, Michelle Opp, Bengt Pederson, Rob Pratt

Software

PROC OPTNET	Matthew Galati

Support Groups

Software Testing	Charles B. Kelly, Yu-Min Lin, Minghui Liu, Bengt Pederson
Technical Support	Tonya Chapman

Chapter 1
What's New in SAS/OR 12.1, 12.2, and 12.3

Contents

Overview

SAS/OR 12.1 delivers a broad range of new capabilities and enhanced features, encompassing optimization, constraint programming, and discrete-event simulation. SAS/OR 12.1 enhancements significantly improve performance and expand your tool set for building, analyzing, and solving operations research models.

In previous years, SAS/OR software was updated only with new releases of Base SAS software, but this is no longer the case. This means that SAS/OR software can be released to customers when enhancements are ready, and the goal is to update SAS/OR every 12 to 18 months. To mark this newfound independence, the release numbering scheme for SAS/OR changed starting with SAS/OR 12.1. This new numbering scheme will be maintained when new versions of Base SAS and SAS/OR are shipped at the same time.

SAS/OR 12.2 is a maintenance release that does not contain any new features. SAS/OR 12.3 is another maintenance release that includes two new features that are now production, as described in the next section.

Highlights of Enhancements in SAS/OR 12.3

In SAS/OR 12.3, two important distributed-computing features become production: the option tuner for the OPTMILP procedure and the nonlinear optimization multistart algorithm for the NLP solver. The option tuner helps determine the most productive combinations of option settings for the OPTMILP procedure, and the NLP multistart algorithm is instrumental in addressing nonconvex nonlinear optimization problems.

SAS/OR 12.3 also adds the OPTLSO procedure, which performs parallel hybrid derivative-free optimization for optimization problems in which any or all of the functions involved can be nonsmooth, discontinuous, or computationally expensive to evaluate directly. The OPTLSO procedure permits both continuous and integer decision variables, and can operate in single-machine mode or distributed mode.

NOTE: Distributed mode requires SAS High-Performance Optimization.

Highlights of Enhancements in SAS/OR 12.1

Highlights of the SAS/OR enhancements include the following:

- multithreading is used to improve performance in these three areas:
  - PROC OPTMODEL problem generation
  - multistart for nonlinear optimization
  - option tuning for mixed integer linear optimization
- concurrent solve capability (experimental) for linear programming (LP) and nonlinear programming (NLP)
- improvements to all simplex LP algorithms and mixed integer linear programming (MILP) solver
- new decomposition (DECOMP) algorithm for LP and MILP
- new option for controlling MILP cutting plane strategy
- new conflict search capability for MILP solver
- option tuning for PROC OPTMILP
- new procedure, PROC OPTNET, for network optimization and analysis
- new SUBMIT block for invoking SAS code within PROC OPTMODEL
- SAS Simulation Studio improvements:
  - one-click connection of remote blocks in large models
  - autoscrolling for navigating large models
  - new search capability for block types and label content
  - alternative Experiment window configuration for large experiments
  - selective animation capability
  - new submodel component (experimental)

The CLP Procedure

In SAS/OR 12.1, the CLP procedure adds two classes of constraints that expand its capabilities and can accelerate its solution process. The LEXICO statement imposes a lexicographic ordering between pairs of variable lists. Lexicographic order is essentially analogous to alphabetical order but expands the concept to include numeric values. One vector (list) of values is lexicographically less than another if the corresponding elements are equal up to a certain point and immediately after that point the next element of the first vector is numerically less than the second. Lexicographic ordering can be useful in eliminating certain types of symmetry that can arise among solutions to constraint satisfaction problems (CSPs). Imposing a lexicographic ordering eliminates many of the mutually symmetric solutions, reducing the number of permissible solutions to the problem and in turn shortening the solution process.

Another constraint class that is added to PROC CLP for SAS/OR 12.1 is the bin-packing constraint, imposed via the PACK statement. A bin-packing constraint directs that a specified number of items must be placed into a specified number of bins, subject to the capacities (expressed in numbers of items) of the bins. The PACK statement provides a compact way to express such constraints, which can often be useful components of larger CSPs or optimization problems.

The DTREE, GANTT, and NETDRAW Procedures

In SAS/OR 12.1 the DTREE, GANTT, and NETDRAW procedures each add procedure-specific graph styles that control fonts, line colors, bar and node fill colors, and background images.

Supporting Technologies for Optimization

The underlying improvements in optimization in SAS/OR 12.1 are chiefly related to multithreading, which denotes the use of multiple computational cores to enable computations to be executed in parallel rather than serially. Multithreading can provide dramatic performance improvements for optimization because these underlying computations are performed many times in the course of an optimization process.

The underlying linear algebra operations for the linear, quadratic, and nonlinear interior point optimization algorithms are now multithreaded. The LP, QP, and NLP solvers can be used by PROC OPTMODEL, PROC OPTLP, and PROC OPTQP in SAS/OR. For nonlinear optimization with PROC OPTMODEL, the evaluation of nonlinear functions is multithreaded for improved performance.

Finally, the process of creating an optimization model from PROC OPTMODEL statements has been multithreaded. PROC OPTMODEL contains powerful declarative and programming statements and is adept at enabling data-driven definition of optimization models, with the result that a rather small section of PROC OPTMODEL code can create a very large optimization model when it is executed. Multithreading can dramatically shorten the time that is needed to create an optimization model.

In SAS/OR 12.1 you can use the NTHREADS= option in the PERFORMANCE statement in PROC OPTMODEL and other SAS/OR optimization procedures to specify the number of cores to be used. Otherwise, SAS detects the number of cores available and uses them.

PROC OPTMODEL: Nonlinear Optimization

The nonlinear optimization solver that PROC OPTMODEL uses builds on the introduction of multithreading for its two most significant improvements in SAS/OR 12.1. First, in addition to the nonlinear solver options ALGORITHM=ACTIVESET and ALGORITHM=INTERIORPOINT, SAS/OR 12.1 introduces the ALGORITHM=CONCURRENT option (experimental), with which you can invoke both the active set and interior point algorithms for the specified problem, running in parallel on separate threads. The solution process terminates when either of the algorithms terminates. For repeated solves of a number of similarly structured problems or simply for problems for which the best algorithm isn't readily apparent, ALGORITHM=CONCURRENT should prove useful and illuminating.

Second, multithreading is central to the nonlinear optimization solver's enhanced multistart capability, which now takes advantage of multiple threads to execute optimizations from multiple starting points in parallel. The multistart capability is essential for problems that feature nonconvex nonlinear functions in either or both of the objective and the constraints because such problems might have multiple locally optimal points. Starting optimization from several different starting points helps to overcome this difficulty, and multithreading this process helps to ensure that the overall optimization process runs as fast as possible.

Linear Optimization with PROC OPTMODEL and PROC OPTLP

Extensive improvements to the primal and dual simplex linear optimization algorithms produce better performance and better integration with the crossover algorithm, which converts solutions that are found by the interior point algorithm into more usable basic optimal solutions. The crossover algorithm itself has undergone extensive enhancements that improve its speed and stability.

Paralleling developments in nonlinear optimization, SAS/OR 12.1 linear optimization introduces a concurrent algorithm, invoked with the ALGORITHM=CONCURRENT option, in the SOLVE WITH LP statement for PROC OPTMODEL or in the PROC OPTLP statement. The concurrent LP algorithm runs a selection of linear optimization algorithms in parallel on different threads, with settings to suit the problem at hand. The optimization process terminates when the first algorithm identifies an optimal solution. As with nonlinear optimization, the concurrent LP algorithm has the potential to produce significant reductions in the time needed to solve challenging problems and to provide insights that are useful when you solve a large number of similarly structured problems.

Mixed Integer Linear Optimization with PROC OPTMODEL and PROC OPTMILP

Mixed integer linear optimization in SAS/OR 12.1 builds on and extends the advances in linear optimization. Overall, solver speed has increased by over 50% (on a library of test problems) compared to SAS/OR 9.3. The branch-and-bound algorithm has approximately doubled its ability to evaluate and solve component linear optimization problems (which are referred to as nodes in the branch-and-bound tree). These improvements have significantly reduced solution time for difficult problems.

The Decomposition Algorithm

The most fundamental change to both linear and mixed integer linear optimization in SAS/OR 12.1 is the addition of the decomposition (DECOMP) algorithm, which is invoked with a specialized set of options in the SOLVE WITH LP and SOLVE WITH MILP statements for PROC OPTMODEL or in the DECOMP statement for PROC OPTLP and PROC OPTMILP. For many linear and mixed integer linear optimization problems, most of the constraints apply only to a small set of decision variables. Typically there are many such sets of constraints, complemented by a small set of linking constraints that apply to all or most of the decision variables. Optimization problems with these characteristics are said to have a "block-angular" structure, because it is easy to arrange the rows of the constraint matrix so that the nonzero values, which correspond to the local sets of constraints, appear as blocks along the main diagonal.

The DECOMP algorithm exploits this structure, decomposing the overall optimization problem into a set of component problems that can be solved in parallel on separate computational threads. The algorithm repeatedly solves these component problems and then cycles back to the overall problem to update key information that is used the next time the component problems are solved. This process repeats until it produces a solution to the complete problem, with the linking constraints present. The combination of parallelized solving of the component problems and the iterative coordination with the solution of the overall problem can greatly reduce solution time for problems that were formerly regarded as too time-consuming to solve practically.

To use the DECOMP algorithm, you must either manually or automatically identify the blocks of the constraint matrix that correspond to component problems. The METHOD= option controls the means by which blocks are identified. METHOD=USER enables you to specify the blocks yourself, using the .block suffix to declare blocks. This is by far the most common method of defining blocks. If your problem has a significant or dominant network structure, you can use METHOD=NETWORK to identify the blocks in the problem automatically. Finally, if no linking constraints are present in your problem, then METHOD=AUTO identifies the blocks automatically.

The DECOMP algorithm uses a number of detailed options that specify how the solution processes for the component problems and the overall problem are configured and how they coordinate with each other. You can also specify the number of computational threads to make available for processing component problems and the level of detail in the information to appear in the SAS log. Options specific to the linear and mixed integer linear solvers that are used by the DECOMP algorithm are largely identical to those for the respective solvers.

Setting the Cutting Plane Strategy

Cutting planes are a major component of the mixed integer linear optimization solver, accelerating its progress by removing fractional (not integer feasible) solutions. SAS/OR 12.1 adds the CUTSTRATEGY= option in the PROC OPTMILP statement and in the SOLVE WITH MILP statement for PROC OPTMODEL, enabling you to determine the aggressiveness of your overall cutting plane strategy. This option complements the individual cut class controls (CUTCLQUE=, CUTGOMORY=, CUTMIR=, and so on), with which you can enable or disable certain cut types, and the ALLCUTS= option, which enables or disables all cutting planes. In contrast, the CUTSTRATEGY= option controls cuts at a higher level, creating a profile for cutting plane use. As the cut strategy becomes more aggressive, more effort is directed toward creating cutting planes and more cutting planes are applied. The available values of the CUTSTRATEGY= option are AUTOMATIC, BASIC, MODERATE, and AGGRESSIVE; the default is AUTOMATIC. The precise cutting plane strategy that corresponds to each of these settings can vary from problem to problem, because the strategy is also tuned to suit the problem at hand.

Conflict Search

Another means of accelerating the solution process for mixed integer linear optimization takes information from infeasible linear optimization problems that are encountered during an initial exploratory phase of the branch-and-bound process. This information is analyzed and ultimately is used to help the branch-and-bound process avoid combinations of decision variable values that are known to lead to infeasibility. This approach, known as conflict analysis or conflict search, influences presolve operations on branch-and-bound nodes, cutting planes, computation of decision variable bounds, and branching. Although the approach is complex, its application in SAS/OR 12.1 is straightforward. The CONFLICTSEARCH= option in the PROC OPTMILP statement or the SOLVE WITH MILP statement in PROC OPTMODEL enables you to specify the level of conflict search to be performed. The available values for the CONFLICTSEARCH= option are NONE, AUTOMATIC, MODERATE, and AGGRESSIVE. A more aggressive search strategy explores more branch-and-bound nodes initially before the branch-and-bound algorithm is restarted with information from infeasible nodes included. The default value is AUTOMATIC, which enables the solver to choose the search strategy.

PROC OPTMILP: Option Tuning

The final SAS/OR 12.1 improvement to the mixed integer linear optimization solver is option tuning, which helps you determine the best option settings for PROC OPTMILP. There are many options and settings available, including controls on the presolve process, branching, heuristics, and cutting planes. The TUNER statement enables you to investigate the effects of the many possible combinations of option settings on solver performance and determine which should perform best. The PROBLEMS= option enables you to submit several problems for tuning at once. The OPTIONMODE= option specifies the options to be tuned. OPTIONMODE=USER indicates that you will supply a set of options and initial values via the OPTIONVALUES= data set, OPTIONMODE=AUTO (the default) tunes a small set of predetermined options, and OPTIONMODE=FULL tunes a much more extensive option set.

Option tuning starts by using an initial set of option values to solve the problem. The problem is solved repeatedly with different option values, with a local search algorithm to guide the choices. When the tuning process terminates, the best option values are output to a data set specified by the SUMMARY= option. You can control the amount of time used by this process by specifying the MAXTIME= option. You can multithread this process by using the NTHREADS= option in the PERFORMANCE statement for PROC OPTMILP, permitting analyses of various settings to occur simultaneously.

PROC OPTMODEL: The SUBMIT Block

In SAS/OR 12.1, PROC OPTMODEL adds the ability to execute other SAS code nested inside PROC OPTMODEL syntax. This code is executed immediately after the preceding PROC OPTMODEL syntax and before the syntax that follows. Thus you can use the SUBMIT block to, for example, invoke other SAS procedures to perform analyses, to display results, or for other purposes, as an integral part of the process of creating and solving an optimization model with PROC OPTMODEL. This addition makes it even easier to integrate the operation of PROC OPTMODEL with other SAS capabilities.

To create a SUBMIT block, use a SUBMIT statement (which must appear on a line by itself) followed by the SAS code to be executed, and terminate the SUBMIT block with an ENDSUBMIT statement (which also must appear on a line by itself). The SUBMIT statement enables you to pass PROC OPTMODEL parameters, constants, and evaluated expressions to the SAS code as macro variables.

Network Optimization with PROC OPTNET

PROC OPTNET, new in SAS/OR 12.1, provides several algorithms for investigating the characteristics of networks and solving network-oriented optimization problems. A network, sometimes referred to as a graph, consists of a set of nodes that are connected by a set of arcs, edges, or links. There are many applications of network structures in real-world problems, including supply chain analysis, communications, transportation, and utilities problems. PROC OPTNET addresses the following classes of network problems:

- biconnected components
- maximal cliques
- connected components
- cycle detection
- weighted matching
- minimum-cost network flow
- minimum cut
- minimum spanning tree
- shortest path
- transitive closure
- traveling salesman

PROC OPTNET syntax provides a dedicated statement for each problem class in the preceding list.

The formats of PROC OPTNET input data sets are designed to fit network-structured data, easing the process of specifying network-oriented problems. The underlying algorithms are highly efficient and can successfully address problems of varying levels of detail and scale. PROC OPTNET is a logical destination for users who are migrating from some of the legacy optimization procedures in SAS/OR. Former users of PROC NETFLOW can turn to PROC OPTNET to solve shortest-path and minimum-cost network flow problems, and former users of PROC ASSIGN can instead use the LINEAR_ASSIGNMENT statement in PROC OPTNET to solve assignment problems.

SAS Simulation Studio 12.1

SAS Simulation Studio 12.1, a component of SAS/OR 12.1 for Windows environments, adds several features that improve your ability to build, explore, and work with large, complex discrete-event simulation models. Large models present a number of challenges to a graphical user interface such as that of SAS Simulation Studio. Connection of model components, navigation within a model, identification of objects or areas of interest, and management of different levels of modeling are all tasks that can become more difficult as the model size grows significantly beyond what can be displayed on one screen. An indirect effect of model growth is an increased number of factors and responses that are needed to parameterize and investigate the performance of the system being modeled.

Improvements in SAS Simulation Studio 12.1 address each of these issues. In SAS Simulation Studio, you connect blocks by dragging the cursor to create links between output and input ports on regular blocks and Connector blocks. SAS Simulation Studio 12.1 automatically scrolls the display of the Model window as you drag the link that is being created from its origin to its destination, thus enabling you to create a link between two blocks that are located far apart (additionally you can connect any two blocks by clicking on the OutEntity port of the first block and then clicking on the InEntity port of the second block). Automatic scrolling also enables you to navigate a large model more easily. To move to a new area in the Model window, you can simply hold down the left mouse button and drag the visible region of the model to the desired area. This works for simple navigation and for moving a block to a new, remote location in the model.

SAS Simulation Studio 12.1 also enables you to search among the blocks in a model and identify the blocks that have a specified type, a certain character string in their label, or both. From the listing of identified blocks, you can open the Properties dialog box for each identified block and edit its settings. Thus, if you can identify a set of blocks that need similar updates, then you can make these updates without manually searching through the model for qualifying blocks and editing them individually. For very large models, this capability not only makes the update process easier but also makes it more thorough because you can identify qualifying blocks centrally.

When you design experiments for large simulation models, you often need a large number of factors to parameterize the model and a large number of responses to track system performance in sufficient detail. This was a challenge prior to SAS Simulation Studio 12.1 because the Experiment window displayed factors and responses in the header row of a table, with design points and their replications' results displayed in the rows below. A very large number of factors and responses did not fit on one screen in this display scheme, and you had to scroll across the Experiment window to view all of them.

SAS Simulation Studio 12.1 provides you with two alternative configurations for the Experiment window. The Design Matrix tab presents the tabular layout described earlier. The Design Point tab presents each design point in its own display. Factors and responses (summarized over replications) are displayed in separate tables, each with the factor or response names appearing in one column and the respective values in a second column. This layout enables a large number of factors and responses to be displayed. Response values for each replication of the design point can be displayed in a separate window.

SAS Simulation Studio 12.1 enhances its multilevel model management features by introducing the submodel component (experimental). Like the compound block, the submodel encapsulates a group of SAS Simulation Studio blocks and their connections, but the submodel outpaces the compound block in some important ways. The submodel, when expanded, opens in its own window. This means a submodel in its collapsed form can be placed close to other blocks in the Model window without requiring space for its expanded form (as is needed for compound blocks). The most important property of the submodel is its ability to be copied and instantiated in several locations simultaneously, whether in the same model, in different models in the same project, or in different projects. Each such instance is a direct reference to the original submodel, not a disconnected copy. Thus you can edit the submodel by editing any of its instances; changes that are made to any instance are propagated to all current and future instances of the submodel. This feature enables you to maintain consistency across your models and projects.

Finally, SAS Simulation Studio 12.1 introduces powerful new animation controls that should prove highly useful in debugging simulation models. In the past, animation could be switched on or off and its speed controlled, but these choices were made for the entire model. If you needed to animate a particular segment of the model, perhaps during a specific time span for the simulation clock, you had to focus your attention on that area and pay special attention when the time period of interest arrived. In SAS Simulation Studio 12.1 you can select both the area of the model to animate (by selecting a block or a compound block) and the time period over which animation should occur (by specifying the start and end times for animation). You can also control simulation speed for each such selection. Multiple selections are supported so that you can choose to animate several areas of the model, each during its defined time period and at its chosen speed.

Chapter 2
The OPTNET Procedure

Contents

Overview: OPTNET Procedure

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

Table 2.1 Algorithm Classes in PROC OPTNET

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

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

Getting Started: OPTNET Procedure

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

This section shows an introductory example for getting started with the OPTNET procedure. For more detail about the input formats expected and the various algorithms available, see the sections "Details: OPTNET Procedure" on page 39 and "Examples: OPTNET Procedure" on page 104.

Road Network Shortest Path

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

In this road network (graph), the links are the roads and the nodes are intersections between roads. With each road, you assign a *link attribute* in the variable time_to_travel to describe the number of minutes that it takes to drive from one node to another. The following data were collected using Google Maps (Google 2011), which gives an approximate number of minutes to traverse between two points, based on the length of the road and the typical speed during normal traffic patterns:

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

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

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

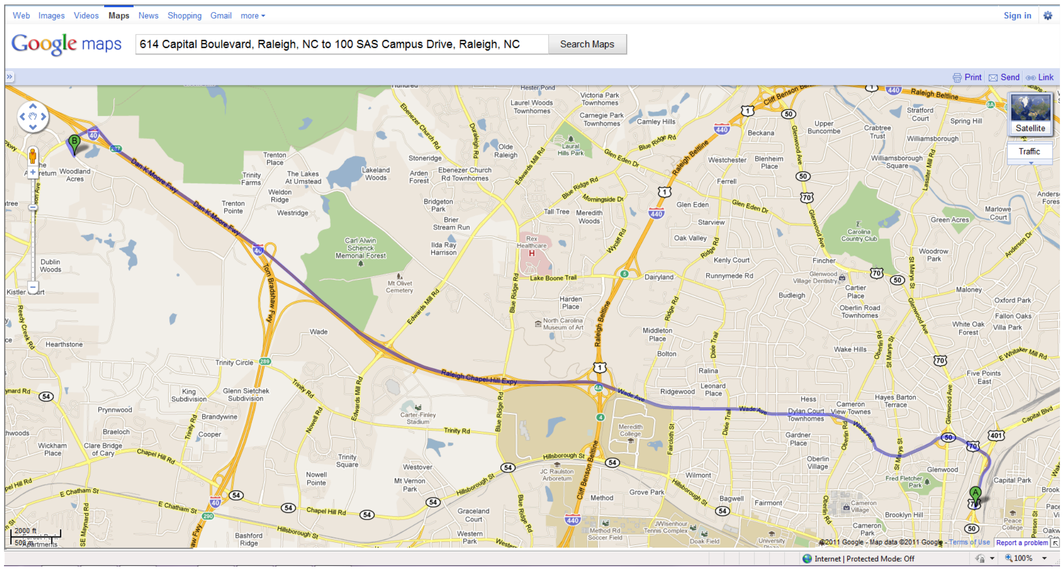
```
proc optnet
   data_links   = LinkSetInRoadNC10am;
   data_links_var
      from      = start_inter
      to        = end_inter
      weight    = time_to_travel;
   shortpath
      out_paths = ShortPath
      source    = "614CapitalBlvd"
      sink      = "SASCampusDrive";
run;
```

For more details about shortest path algorithms in PROC OPTNET, see the section "Shortest Path" on page 77. Figure 2.1 displays the output data set ShortPath, which gives the best route to take to minimize travel time at 10:00 a.m. This route is also shown in Google Maps in Figure 2.2.

Figure 2.1 Shortest Path for Road Network at 10:00 A.M.

```
                                                             time_to_
 order        start_inter              end_inter              travel

   1       614CapitalBlvd         Capital/WadeAve             1.4400
   2       Capital/WadeAve        WadeAve/RaleighExpy         4.5000
   3       WadeAve/RaleighExpy    RaleighExpy/US40W           3.0000
   4       RaleighExpy/US40W      US40W/HarrisonAve           1.4182
   5       US40W/HarrisonAve      SASCampusDrive              1.2000
                                                            ========
                                                             11.5582
```

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



614 Capital Boulevard, Raleigh, NC to 100 SAS Campus Drive, Raleigh, NC

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

```
data LinkSetInRoadNC5pm;
   input start_inter $1-20 end_inter $20-40 miles miles_per_hour;
   datalines;
614CapitalBlvd      Capital/WadeAve      0.6  25
614CapitalBlvd      Capital/US70W        0.6  25
614CapitalBlvd      Capital/US440W       3.0  45
Capital/WadeAve     WadeAve/RaleighExpy  3.0  25 /*high traffic*/
Capital/US70W       US70W/US440W         3.2  60
US70W/US440W        US440W/RaleighExpy   2.7  60
```

```
Capital/US440W      US440W/RaleighExpy   6.7  60
US440W/RaleighExpy  RaleighExpy/US40W    3.0  60
WadeAve/RaleighExpy RaleighExpy/US40W    3.0  60
RaleighExpy/US40W   US40W/HarrisonAve    1.3  55
US40W/HarrisonAve   SASCampusDrive       0.5  25
;

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

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

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

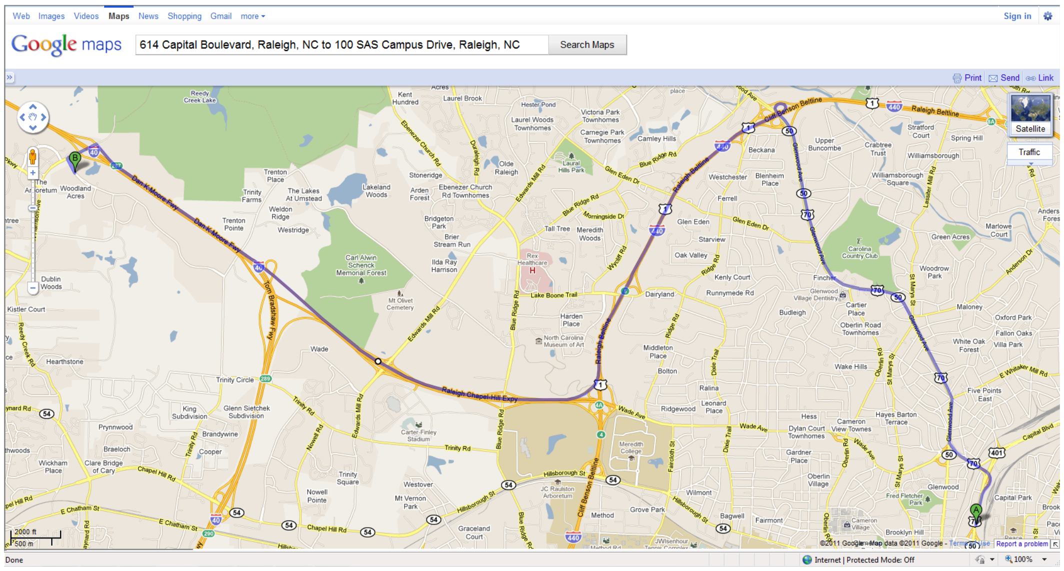
Now, the output data set ShortPath, shown in Figure 2.3, shows the best route for going home. Since the traffic on Wade Avenue is typically heavy at this time of day, the route home is different from the route to work.

Figure 2.3 Shortest Path for Road Network at 5:00 P.M.

```
                                                   time_to_
 order    start_inter          end_inter             travel

   1      SASCampusDrive       US40W/HarrisonAve     1.2000
   2      US40W/HarrisonAve    RaleighExpy/US40W     1.4182
   3      RaleighExpy/US40W    US440W/RaleighExpy    3.0000
   4      US440W/RaleighExpy   US70W/US440W          2.7000
   5      US70W/US440W         Capital/US70W         3.2000
   6      Capital/US70W        614CapitalBlvd        1.4400
                                                   ========
                                                    12.9582
```

This new route is shown in Google Maps in Figure 2.4.

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



614 Capital Boulevard, Raleigh, NC to 100 SAS Campus Drive, Raleigh, NC

Syntax: OPTNET Procedure

PROC OPTNET *options* **;**

Data Input Statements:

DATA_ADJ_MATRIX_VAR *column1 <,column2,...>* **;**
DATA_LINKS_VAR *< options >* **;**
DATA_MATRIX_VAR *column1 <,column2,...>* **;**
DATA_NODES_VAR *< options >* **;**

Algorithm Statements:

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

PROC OPTNET statements are divided into three main categories: the PROC statement, the data input statements, and the algorithm statements. The PROC statement invokes the procedure and sets option values that are used across multiple algorithms. The data input statements control the names of the variables that

PROC OPTNET expects in the data input. The algorithm statements determine which algorithms are run and set options for each individual algorithm.

The section "Functional Summary" on page 17 provides a quick reference for each of the options for each statement. Each statement is then described in more detail in its own section; the PROC OPTNET statement is described first, and sections that describe all other statements are presented in alphabetical order.

Functional Summary

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

Table 2.2 Functional Summary

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

Table 2.2 (continued)

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

Table 2.2 (continued)

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

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

Table 2.3 Supported Input Formats and Graph Types by Statement

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

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

Table 2.4 Output Options by Statement

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

PROC OPTNET Statement

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

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

DATA_ADJ_MATRIX=*SAS-data-set*

ADJ_MATRIX=*SAS-data-set*

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

See the section "Adjacency Matrix Input Data" on page 43 for more information.

DATA_LINKS=*SAS-data-set*

LINKS=*SAS-data-set*

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

See the section "Link Input Data" on page 39 for more information.

DATA_MATRIX=*SAS-data-set*

MATRIX=*SAS-data-set*

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

See the section "Matrix Input Data" on page 46 for more information.

DATA_NODES=*SAS-data-set*

NODES=*SAS-data-set*

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

See the section "Node Input Data" on page 44 for more information.

DATA_NODES_SUB=*SAS-data-set*

NODES_SUB=*SAS-data-set*

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

See the section "Node Subset Input Data" on page 45 for more information.

GRAPH_DIRECTION=DIRECTED | UNDIRECTED

DIRECTION=DIRECTED | UNDIRECTED

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

Table 2.5 Values for the GRAPH_DIRECTION= Option

Option Value	Description
DIRECTED	Specifies the graph as directed. In a directed graph, each link (i, j) has a direction that defines how something (for example, information) might flow over that link. In link (i, j), information flows from node i to node j $(i \rightarrow j)$. The node i is called the *source* (or *tail*) node, and j is called the *sink* (or *head*) node.

Table 2.5 (continued)

Option Value	**Description**
UNDIRECTED	Specifies the graph as undirected. In an undirected graph, each link $\{i, j\}$ has no direction and information can flow in either direction. That is, $\{i, j\} = \{j, i\}$. This is the default.

See the section "Graph Input Data" on page 39 for more information.

GRAPH_INTERNAL_FORMAT=THIN | FULL

INTERNAL_FORMAT=THIN | FULL

requests the internal graph format for the algorithms to use.

Table 2.6 Values for the GRAPH_INTERNAL_FORMAT= Option

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

See the section "Graph Input Data" on page 39 for more information.

INCLUDE_SELFLINK

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

LOGLEVEL=*number* | *string*

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

Table 2.7 Values for LOGLEVEL= Option

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

The default is BASIC.

OUT_LINKS=*SAS-data-set*

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

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

OUT_NODES=*SAS-data-set*

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

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

TIMETYPE=*number | string*

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

Table 2.8 Values for TIMETYPE= Option

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

The default is CPU.

BICONCOMP Statement

BICONCOMP < *option* > **;**

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

See the section "Biconnected Components and Articulation Points" on page 47 for more information.

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

LOGLEVEL=*number* | *string*

controls the amount of information that is displayed in the SAS log. Table 2.9 describes the valid values for this option.

Table 2.9 Values for LOGLEVEL= Option

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

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

CLIQUE Statement

CLIQUE < *options* > **;**

The CLIQUE statement invokes an algorithm that finds maximal cliques on the input graph. Maximal cliques are described in the section "Clique" on page 51.

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

LOGLEVEL=*number* | *string*

controls the amount of information that is displayed in the SAS log. Table 2.10 describes the valid values for this option.

Table 2.10 Values for LOGLEVEL= Option

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

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

MAXCLIQUES=*number*

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

MAXTIME=*number*

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

OUT=*SAS-data-set*

specifies the output data set to contain the maximal cliques.

CONCOMP Statement

CONCOMP < *options* > **;**

The CONCOMP statement invokes an algorithm that finds the connected components of the input graph. Connected components are described in the section "Connected Components" on page 54.

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

ALGORITHM=DFS | UNION_FIND

specifies the algorithm to use for calculating connected components.

Table 2.11 Values for the ALGORITHM= Option

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

The default is DFS.

LOGLEVEL=*number* **|** *string*

controls the amount of information that is displayed in the SAS log. Table 2.12 describes the valid values for this option.

Table 2.12 Values for LOGLEVEL= Option

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

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

CYCLE Statement

CYCLE < *options* > **;**

The CYCLE statement invokes an algorithm that finds the cycles (or the existence of a cycle) in the input graph. Cycles are described in the section “Cycle” on page 59.

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

LOGLEVEL=*number* **|** *string*

controls the amount of information that is displayed in the SAS log. Table 2.13 describes the valid values for this option.

Table 2.13 Values for LOGLEVEL= Option

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

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

MAXCYCLES=*number*

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

MAXLENGTH=*number*

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

MAXLINKWEIGHT=*number*

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

MAXNODEWEIGHT=*number*

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

MAXTIME=*number*

specifies the maximum amount of time to spend finding cycles. The type of time (either CPU time or real time) is determined by the value of the TIMETYPE= option. The value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment. This option works only when you also specify MODE=ALL_CYCLES.

MINLENGTH=*number*

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

MINLINKWEIGHT=*number*

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

MINNODEWEIGHT=*number*

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

MODE=*option*

specifies the mode for processing cycles.

Table 2.14 Values for the MODE= Option

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

The default is FIRST_CYCLE.

OUT=*SAS-data-set*

specifies the output data set to contain the cycles found.

DATA_ADJ_MATRIX_VAR Statement

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

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

The DATA_ADJ_MATRIX_VAR statement enables you to explicitly define the data set variable names for PROC OPTNET to use when it reads the data set that is specified in the DATA_ADJ_MATRIX= option in the PROC OPTNET statement. The format of the adjacency matrix input data set is defined in the section "Adjacency Matrix Input Data" on page 43. The value of each *column* variable must be numeric.

DATA_LINKS_VAR Statement

DATA_LINKS_VAR *< options >* **;**

LINKS_VAR *< options >* **;**

The DATA_LINKS_VAR statement enables you to explicitly define the data set variable names for PROC OPTNET to use when it reads the data set that is specified in the DATA_LINKS= option in the PROC OPTNET statement. The format of the links input data set is defined in the section "Link Input Data" on page 39.

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

FROM=*column*

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

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

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

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

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

DATA_MATRIX_VAR Statement

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

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

The DATA_MATRIX_VAR statement enables you to explicitly define the data set variable names for PROC OPTNET to use when it reads the data set that is specified in the DATA_MATRIX= option in the PROC OPTNET statement. The format of the matrix input data set is defined in the section "Matrix Input Data" on page 46. The value of each *column* variable must be numeric.

DATA_NODES_VAR Statement

DATA_NODES_VAR *< options >* **;**

NODES_VAR *< options >* **;**

The DATA_NODES_VAR statement enables you to explicitly define the data set variable names for PROC OPTNET to use when it reads the data set that is specified in the DATA_NODES= option in the PROC OPTNET statement. The format of the node input data set is defined in the section "Node Input Data" on page 44.

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

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

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

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

LINEAR_ASSIGNMENT Statement

LINEAR_ASSIGNMENT < *options* > **;**

LAP < *options* > **;**

The LINEAR_ASSIGNMENT statement invokes an algorithm that solves the minimal-cost linear assignment problem. In graph terms, this problem is also known as the minimum link-weighted matching problem on a bipartite graph. The input data (the cost matrix) is typically defined in the input data set that is specified in the DATA_MATRIX= option in the PROC OPTNET statement. The data can also be defined as a directed graph by specifying the DATA_LINKS= option in the PROC OPTNET statement, where the costs are defined as link weights. Internally, the graph is treated as a bipartite graph in which the *from* nodes define one part and the *to* nodes define the other part.

The linear assignment problem is described in the section "Linear Assignment (Matching)" on page 65.

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

ID=(*column1 <,column2,...>***)**

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

LOGLEVEL=*number* | *string*

controls the amount of information that is displayed in the SAS log. Table 2.15 describes the valid values for this option.

Table 2.15 Values for LOGLEVEL= Option

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

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

OUT=*SAS-data-set*

specifies the output data set to contain the solution to the linear assignment problem.

WEIGHT=(*column1 <,column2,...>***)**

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

MINCOSTFLOW Statement

MINCOSTFLOW < *options* > **;**

MCF < *options* > **;**

The MINCOSTFLOW statement invokes an algorithm that solves the minimum-cost network flow problem on an input graph.

The minimum-cost network flow problem is described in the section "Minimum-Cost Network Flow" on page 73.

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

LOGFREQ=*number*

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

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

LOGLEVEL=*number* | *string*

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

Table 2.16 Values for LOGLEVEL= Option

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

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

MAXTIME=*option*

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

MINCUT Statement (Experimental)

MINCUT < *options* > ;

The MINCUT statement invokes an algorithm that finds the minimum link-weighted cut of an input graph.

The minimum cut problem is described in the section "Minimum Cut" on page 66.

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

LOGLEVEL=*number* | *string*

controls the amount of information that is displayed in the SAS log. Table 2.17 describes the valid values for this option.

Table 2.17 Values for LOGLEVEL= Option

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

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

MAXNUMCUTS=*number*

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

MAXWEIGHT=*number*

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

OUT=*SAS-data-set*

specifies the output data set to contain the solution to the minimum cut problem.

MINSPANTREE Statement

MINSPANTREE < *options* > ;

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

The minimum spanning tree problem is described in the section "Minimum Spanning Tree" on page 70.

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

LOGLEVEL=*number | string*

controls the amount of information that is displayed in the SAS log. Table 2.18 describes the valid values for this option.

Table 2.18 Values for LOGLEVEL= Option

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

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

OUT=*SAS-data-set*

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

SHORTPATH Statement

SHORTPATH < *options* > **;**

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

The shortest path algorithm is described in the section "Shortest Path" on page 77.

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

LOGFREQ=*number*

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

LOGLEVEL=*number*

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

Table 2.19 Values for LOGLEVEL= Option

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

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

OUT_PATHS=*SAS-data-set*

OUT=*SAS-data-set*

specifies the output data set to contain the shortest paths.

OUT_WEIGHTS=*SAS-data-set*

specifies the output data set to contain the shortest path summaries.

PATHS=ALL | SHORTEST | LONGEST

specifies the type of output to produce in the output data set that is specified in the OUT_PATHS= option.

Table 2.20 Values for the PATHS= Option

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

SINK=*sink-node*

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

SOURCE=*source-node*

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

USEWEIGHT=YES | NO

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

Table 2.21 Values for the WEIGHT= Option

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

WEIGHT2=*column*

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

TRANSITIVE_CLOSURE Statement

TRANSITIVE_CLOSURE < *options* > **;**

TRANSC < *options* > **;**

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

Transitive closure is described in the section "Transitive Closure" on page 89.

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

LOGLEVEL=*number*

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

Table 2.22 Values for LOGLEVEL= Option

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

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

OUT=*SAS-data-set*

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

TSP Statement

TSP < *options* > **;**

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

The traveling salesman problem is described in the section "Traveling Salesman Problem" on page 91. The algorithm that is used to solve this problem is built around the same method as is used in PROC OPTMILP: a branch-and-cut algorithm. Many of the options below are the same as those described for PROC OPTMILP in the *SAS/OR User's Guide: Mathematical Programming*.

You can specify the following *options*:

ABSOBJGAP=*number*

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

CUTOFF=*number*

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

CUTSTRATEGY=*option*

specifies the level of cuts to be generated by PROC OPTNET. Table 2.23 lists the valid values for this option.

Table 2.23 Values for CUTSTRATEGY= Option

number	*string*	**Description**
–1	AUTOMATIC	Disables most of the generic mixed-integer programming cuts and focuses on the generation of TSP-specific cuts
0	NONE	Disables generation of cutting planes
1	MODERATE	Uses a moderate cut strategy
2	AGGRESSIVE	Uses an aggressive cut strategy

The default is AUTOMATIC.

EMPHASIS=*number* **|** *string*

specifies a search emphasis *option* or its corresponding value *number* as listed in Table 2.24.

Table 2.24 Values for EMPHASIS= Option

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

The default is BALANCE.

HEURISTICS=*number* **|** *string*

controls the level of initial and primal heuristics that are applied by PROC OPTNET. This level determines how frequently primal heuristics are applied during the branch-and-bound tree search. It also affects the maximum number of iterations that are allowed in iterative heuristics. Some computationally expensive heuristics might be disabled by the solver at less aggressive levels. Table 2.25 lists the valid values for this option.

Table 2.25 Values for HEURISTICS= Option

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

The default is AUTOMATIC.

INTTOL=*number*

specifies the amount by which an integer variable value can differ from an integer and still be considered integer feasible. The value of *number* can be any number between 0.0 and 1.0; the default value is 1E–5. PROC OPTNET attempts to find an optimal solution with integer infeasibility less than *number*. If you assign a value that is less than 1E–10 to *number* and the best solution found by PROC OPTNET has integer infeasibility between *number* and 1E–10, then PROC OPTNET ends with a solution status of OPTIMAL_COND (see the section "TSP" on page 98).

LOGFREQ=*number*

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

LOGLEVEL=*number* **|** *string*

controls the amount of information displayed in the SAS log by the solver, from a short description of presolve information and summary to details at each branch-and-bound node. Table 2.26 describes the valid values for this option.

Table 2.26 Values for LOGLEVEL= Option

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

The default value is MODERATE.

MAXNODES=*number*

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

MAXSOLS=*number*

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

MAXTIME=*number*

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

MILP=*number* | *string*

specifies whether to use a mixed-integer linear programming (MILP) solver for solving the traveling salesman problem. The MILP solver attempts to find the overall best TSP tour by using a branch-and-bound based algorithm. This algorithm can be expensive for large-scale problems. If MILP=OFF, then PROC OPTNET uses its initial heuristics to find a feasible, but not necessarily optimal, tour as quickly as possible. Table 2.27 describes the valid values for this option.

Table 2.27 Values for MILP= Option

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

NODESEL=*number* | *string*

specifies the branch-and-bound node selection strategy *option* or its corresponding value *number*, as listed in Table 2.28.

Table 2.28 Values for NODESEL= Option

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

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

OUT=*SAS-data-set*

specifies the output data set to contain the solution to the traveling salesman problem.

PROBE=*number* | *string*

specifies a probing *option* or its corresponding value *number*, as listed in Table 2.29:

Table 2.29 Values for PROBE= Option

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

The default value is NONE.

RELOBJGAP=*number*

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

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

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

STRONGITER=*number*

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

STRONGLEN=*number*

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

TARGET=*number*

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

VARSEL=*number* **|** *string*

specifies the rule for selecting the branching variable. Table 2.30 lists the valid values for this option.

Table 2.30 Values for VARSEL= Option

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

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

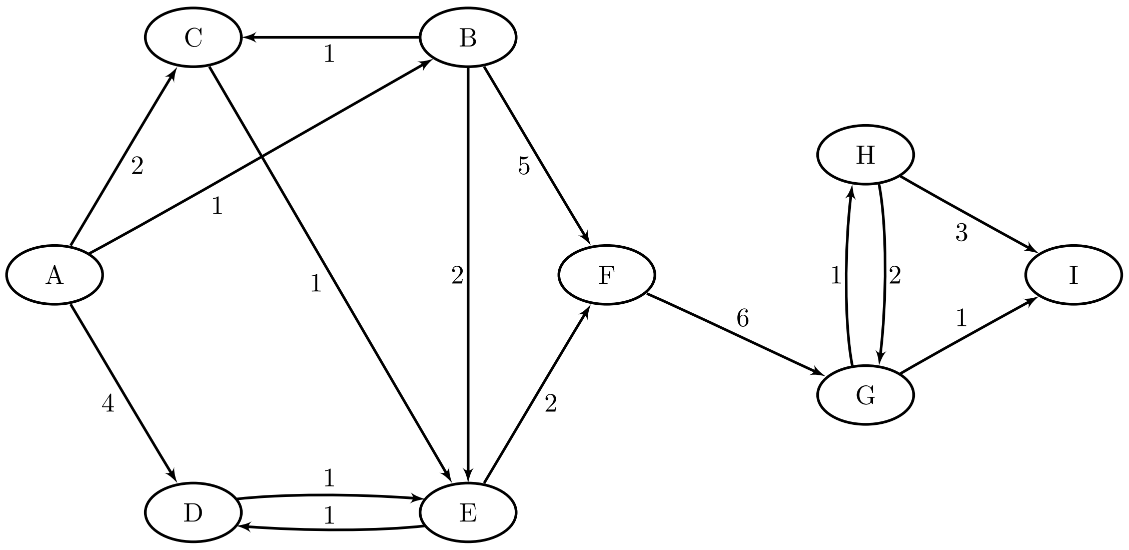
Details: OPTNET Procedure

Graph Input Data

This section describes how to input a graph for analysis by PROC OPTNET. Let $G = (N, A)$ define a graph with a set N of nodes and a set A of links. There are two main methods for defining the set of links A as a SAS data set. The first is to use a list of links as described in the section "Link Input Data" on page 39. The second is to use an adjacency matrix as described in the section "Adjacency Matrix Input Data" on page 43.

To illustrate the different methods for input of a graph, consider the directed graph shown in Figure 2.5.

Figure 2.5 A Simple Directed Graph



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

Link Input Data

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

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

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

The data set variable names can have any values that you want. If you use nonstandard names, you must identify the variables by using the DATA_LINKS_VAR statement, as described in the section "DATA_LINKS_VAR Statement" on page 27.

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

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

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

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

```
proc optnet
   data_links = LinkSetInA;
run;

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

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

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

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

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

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

Figure 2.6 Node Data Set of a Simple Directed Graph

```
node

 A
 B
 C
 D
 E
 F
 G
 H
 I
```

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

Figure 2.7 Link Data Set of a Simple Directed Graph

```
Obs    from    to    weight

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

If you define this graph as undirected, then reciprocal links (for example, $D \leftrightarrow E$) are treated as the same link and duplicates are removed. PROC OPTNET takes the first occurrence of the link and ignores the others. The default for the GRAPH_DIRECTION= option is UNDIRECTED, so you can just remove this option to declare the graph as undirected.

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

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

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

```
NOTE: ---------------------------------------------------------------------------
NOTE: Running OPTNET version 12.3.
NOTE: ---------------------------------------------------------------------------
NOTE: Data input used 0.01 (cpu: 0.02) seconds.
WARNING: Link (E,D) in observation 9 of the DATA_LINKS data set is a duplicate
         and is ignored.
WARNING: Link (H,G) in observation 14 of the DATA_LINKS data set is a duplicate
         and is ignored.
NOTE: The number of nodes in the input graph is 9.
NOTE: The number of links in the input graph is 13.
NOTE: ---------------------------------------------------------------------------
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------
NOTE: The data set WORK.NODESETOUT has 9 observations and 1 variables.
NOTE: The data set WORK.LINKSETOUT has 13 observations and 3 variables.
```

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

Figure 2.9 Link Data Set of a Simple Undirected Graph

```
Obs    from    to    weight

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

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

Adjacency Matrix Input Data

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

You can specify any values that you want for the data set variable names (the columns) by using the DATA_ADJ_MATRIX_VAR statement, as described in the section "DATA_ADJ_MATRIX_VAR Statement" on page 27. If no names are given, then PROC OPTNET assumes that all numeric variables in the data set are to be used in defining nodes and links.

The directed graph G shown in Figure 2.5 can be represented structurally by using the adjacency matrix data set AdjMatSetIn as follows:

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

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

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

To represent the weights, you can simply use the weights from Figure 2.5 in the input matrix as follows:

```
data AdjMatWtSetIn;
   input var1-var9;
   datalines;
. 1 2 4 . . . . .
. . 1 . 2 5 . . .
. . . . 1 . . . .
. . . . 1 . . . .
. . . 1 . 2 . . .
. . . . . . 6 . .
. . . . . . . 1 1
. . . . . . 2 . 3
. . . . . . . . .
;
```

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

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

So the following two procedure calls are equivalent:

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

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

The first set of statements uses the DATA_LINKS= option, which represents the graph in sparse format, as described in the section "Link Input Data" on page 39. The second set of statements uses the DATA_ADJ_MATRIX= option, which represents the graph as an adjacency matrix (a dense format). The dense format is not appropriate for large graphs because the memory requirements grow quadratically with the number of nodes.

Node Input Data

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

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

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

You can specify any values that you want for the data set variable names. If you use nonstandard names, you must identify the variables by using the DATA_NODES_VAR statement, as described in the section "DATA_NODES_VAR Statement" on page 28.

The data set that is specified in the DATA_LINKS= option defines the set of nodes that are incident to some link. If the graph contains a node that has no links (called a *singleton node*), then this node must be defined in the DATA_NODES data set. The following is an example of a graph with three links but four nodes, including a singleton node D:

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

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

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

Node Subset Input Data

For some algorithms you might want to process only a subset of the nodes in the input graph. You can accomplish this by using the DATA_NODES_SUB= option in the PROC OPTNET statement. You can use the node subset data set in conjunction with the SHORTPATH statement (see the section "Shortest Path" on page 77. The node subset data set is expected to contain some combination of the following variables:

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

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

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

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

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

Matrix Input Data

This section describes the matrix input format that you can use with some of the algorithms in PROC OPTNET. The DATA_MATRIX= option in the PROC OPTNET statement defines the data set that contains the matrix values. You can specify any values that you want for the data set variable names (the columns) by using the DATA_MATRIX_VAR statement, as described in the section "DATA_MATRIX_VAR Statement" on page 28. If you do not specify any names, then PROC OPTNET assumes that all numeric variables in the data set are to be used in defining the matrix.

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

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

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

Biconnected Components and Articulation Points

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

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

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

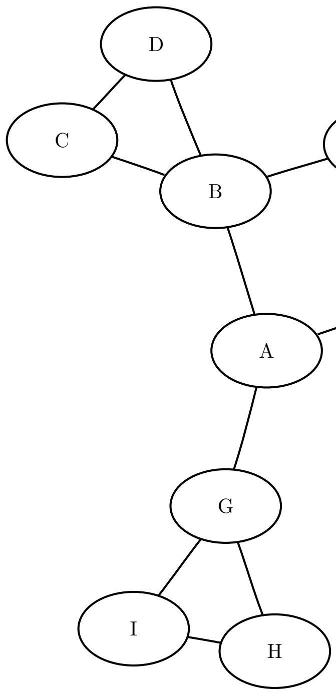
The biconnected components algorithm reports status information in a macro variable called _OROPTNET_BICONCOMP_. See the section "Macro Variable _OROPTNET_BICONCOMP_" on page 99 for more information about this macro variable.

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

Biconnected Components of a Simple Undirected Graph

This section illustrates the use of the biconnected components algorithm on the simple undirected graph G shown in Figure 2.10.

Figure 2.10 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 optnet
   data_links = LinkSetInBiCC
   out_links  = LinkSetOut
   out_nodes  = NodeSetOut;
   biconcomp;
run;
```

The data set LinkSetOut now contains the biconnected components of the input graph, as shown in Figure 2.11.

Figure 2.11 Biconnected Components of a Simple Undirected Graph

```
from    to    biconcomp

 A      B         2
 A      F         2
 A      G         4
 B      C         1
 B      D         1
 B      E         2
 C      D         1
 E      F         2
 G      I         3
 G      H         3
 H      I         3
```

In addition, the data set NodeSetOut contains the articulation points of the input graph, as shown in Figure 2.12.

Figure 2.12 Articulation Points of a Simple Undirected Graph

```
node    artpoint

 A          1
 B          1
 F          0
 G          1
 C          0
 D          0
 E          0
 I          0
 H          0
```

The biconnected components are shown graphically in Figure 2.13 and Figure 2.14.

Figure 2.13 Biconnected Components C^1 and C^2

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

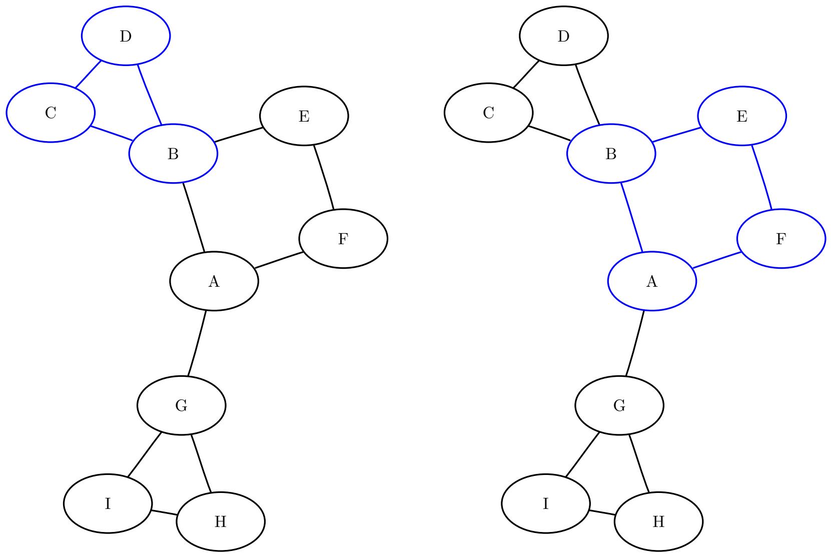
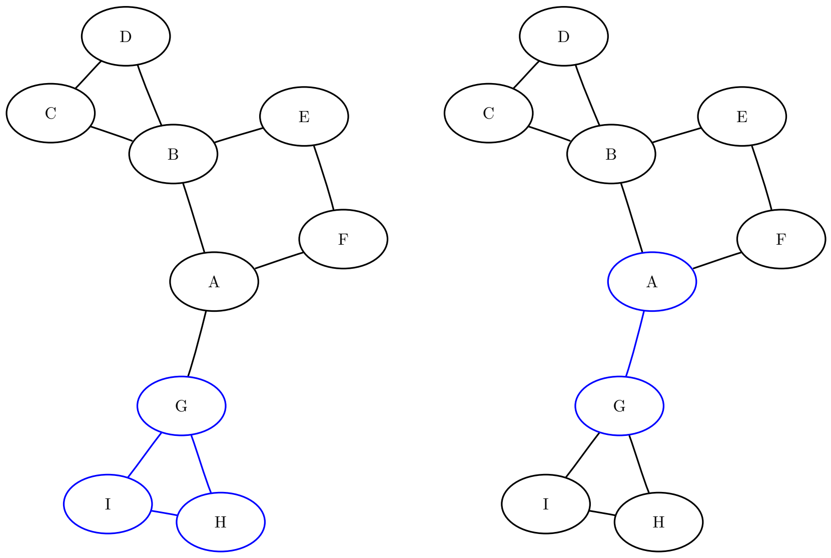


Figure 2.14 Biconnected Components C^3 and C^4

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



For a more detailed example, see "Example 2.1: Articulation Points in a Terrorist Network" on page 104.

Clique

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

You can find the maximal cliques of an input graph by invoking the CLIQUE statement. The options for this statement are described in the section "CLIQUE Statement" on page 24. This algorithm works only with undirected graphs.

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

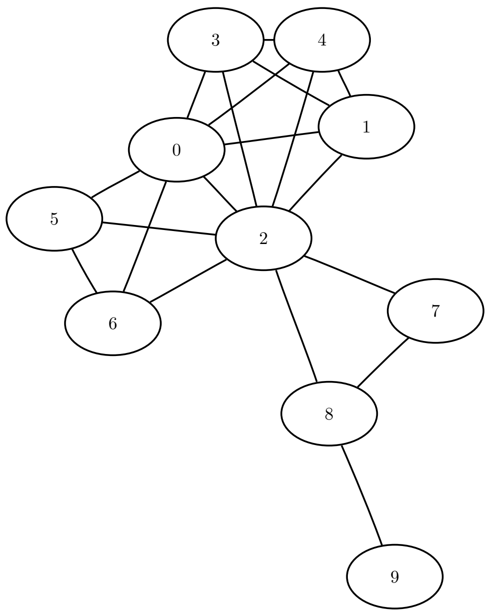
The clique algorithm reports status information in a macro variable called _OROPTNET_CLIQUE_. See the section "Macro Variable _OROPTNET_CLIQUE_" on page 100 for more information about this macro variable.

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

Maximal Cliques of a Simple Undirected Graph

This section illustrates the use of the clique algorithm on the simple undirected graph G shown in Figure 2.15.

Figure 2.15 A Simple Undirected Graph G



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

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

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

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

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

The data set Cliques now contains the maximal cliques of the input graph; it is shown in Figure 2.16.

Figure 2.16 Maximal Cliques of a Simple Undirected Graph

```
clique    node

   1        0
   1        2
   1        1
   1        3
   1        4
   2        0
   2        2
   2        5
   2        6
   3        2
   3        8
   3        7
   4        8
   4        9
```

In addition, the data set CliqueSizes contains the number of nodes in each clique; it is shown in Figure 2.17.

Figure 2.17 Sizes of Maximal Cliques of a Simple Undirected Graph

```
clique    size

   1        5
   2        4
   3        3
   4        2
```

The maximal cliques are shown graphically in Figure 2.18 and Figure 2.19.

Figure 2.18 Maximal Cliques C^1 and C^2

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

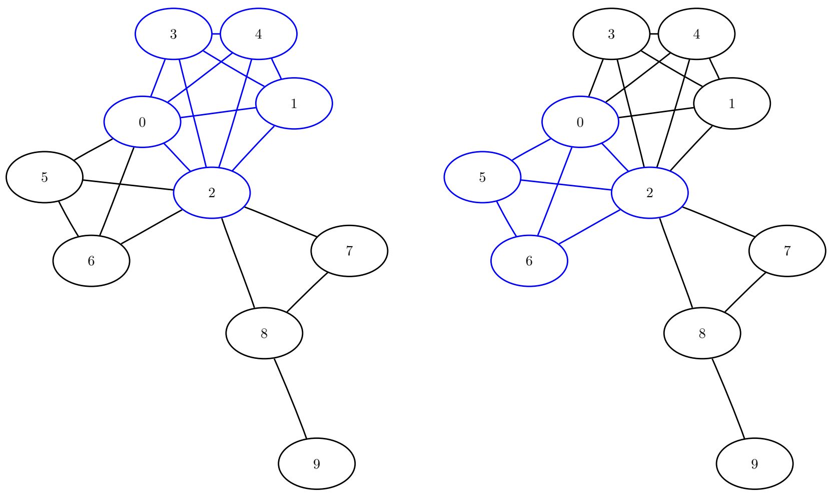
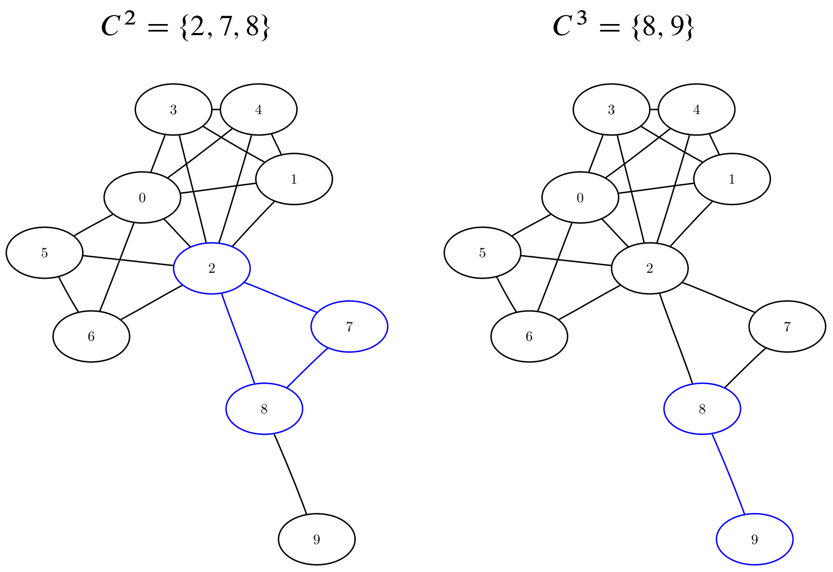


Figure 2.19 Maximal Cliques C^2 and C^3



Connected Components

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

In PROC OPTNET, connected components can be invoked by using the CONCOMP statement. The options for this statement are described in the section "CONCOMP Statement" on page 24.

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

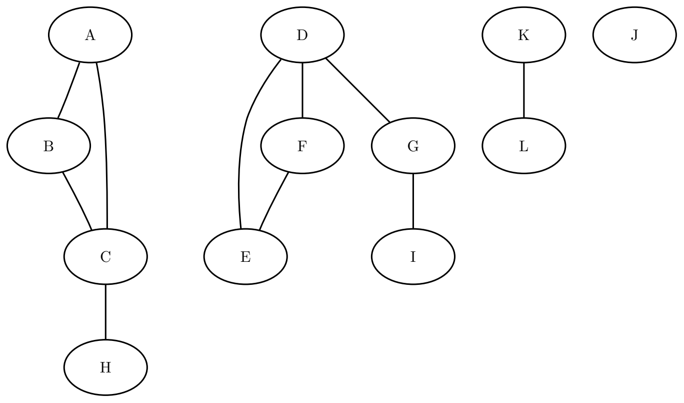
The results for the connected components algorithm are written to the output node data set that is specified in the OUT_NODES= option in the PROC OPTNET statement. For each node in the node data set, the variable concomp identifies its component. The component identifiers are numbered sequentially starting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

The connected components algorithm reports status information in a macro variable called _OROPT-NET_CONCOMP_. See the section "Macro Variable _OROPTNET_CONCOMP_" on page 100 for more information about this macro variable.

Connected Components of a Simple Undirected Graph

This section illustrates the use of the connected components algorithm on the simple undirected graph G shown in Figure 2.20.

Figure 2.20 A Simple Undirected Graph G



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

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

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

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

The data set NodeSetOut contains the connected components of the input graph and is shown in Figure 2.21.

Figure 2.21 Connected Components of a Simple Undirected Graph

```
                 node    concomp

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

Notice that the graph was defined by using only the links data set. As seen in Figure 2.20, this graph also contains a singleton node labeled J, which has no associated links. By definition, this node defines its own component. But because the input graph was defined with the links data set alone, it did not show up in the results data set. To define a graph with nodes that have no associated links, you should also define the input nodes data set. In this case, define the nodes data set NodeSetIn as follows:

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

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

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

The resulting data set NodeSetOut includes the singleton node J as its own component, as shown in Figure 2.22.

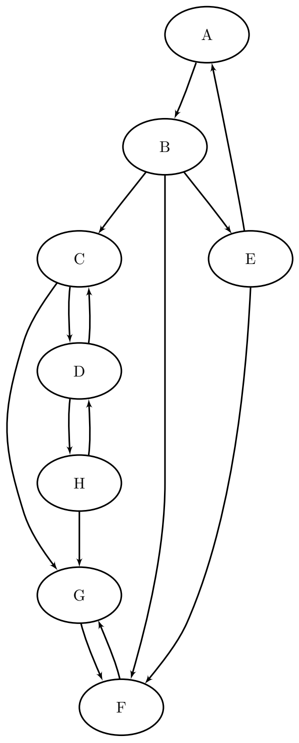
Figure 2.22 Connected Components of a Simple Undirected Graph

```
node    concomp

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

Connected Components of a Simple Directed Graph

This section illustrates the use of the connected components algorithm on the simple directed graph G shown in Figure 2.23.

Figure 2.23 A Simple Directed Graph G



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

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

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

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

The data set NodeSetOut, shown in Figure 2.24, now contains the connected components of the input graph.

Figure 2.24 Connected Components of a Simple Directed Graph

```
node    concomp

 A         3
 B         3
 C         2
 E         3
 F         1
 G         1
 D         2
 H         2
```

The connected components are represented graphically in Figure 2.25.

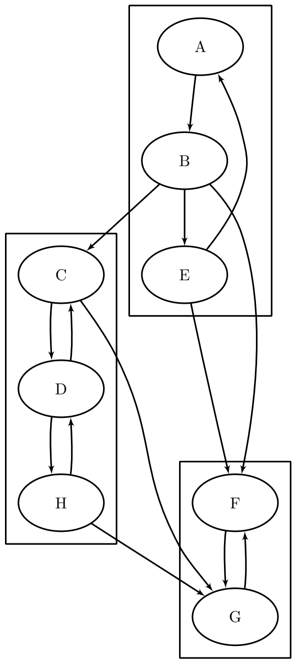


Figure 2.25 Strongly Connected Components of G

Cycle

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

In PROC OPTNET, you can find cycles (or just count the cycles) of an input graph by invoking the CYCLE statement. The options for this statement are described in the section "CYCLE Statement" on page 25. To find the cycles and report them in an output data set, use the OUT= option. To simply count the cycles, do not use the OUT= option.

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

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

The cycle detection algorithm reports status information in a macro variable called _OROPTNET_CYCLE_. See the section "Macro Variable _OROPTNET_CYCLE_" on page 101 for more information about this macro variable.

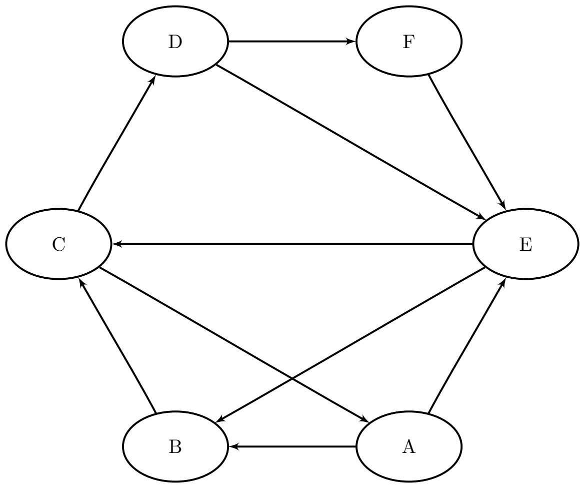
The algorithm used by PROC OPTNET to compute all cycles is a variant of the algorithm found in Johnson 1975. This algorithm runs in time $O((|N| + |A|)(c + 1))$, where c is the number of elementary cycles in the graph. So, the algorithm should scale to large graphs that contain few cycles. However, some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

If MODE=ALL_CYCLES and there are many cycles, the OUT= data set can become very large. It might be beneficial to check the number of cycles before you try to create the OUT= data set. When you specify MODE=FIRST_CYCLE, the algorithm returns the first cycle found and stops processing. This should run relatively quickly. On large-scale graphs, the MINLINKWEIGHT= and MAXLINKWEIGHT= options can be relatively expensive and might increase the computation time. See the section "CYCLE Statement" on page 25 for more information about these options.

Cycle Detection of a Simple Directed Graph

This section provides a simple example for using the cycle detection algorithm on the simple directed graph G shown in Figure 2.26. Two other examples are Example 2.2, which shows the use of cycle detection for optimizing a kidney donor exchange, and Example 2.6, which shows another application of cycle detection.

Figure 2.26 A Simple Directed Graph G



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

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

The following statements check whether the graph has a cycle:

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

The result is written to the log of the procedure, as shown Figure 2.27.

Figure 2.27 PROC OPTNET Log: Check the Existence of a Cycle in a Simple Directed Graph

```
NOTE: ---------------------------------------------------------------------------
NOTE: Running OPTNET version 12.3.
NOTE: ---------------------------------------------------------------------------
NOTE: Data input used 0.01 (cpu: 0.02) seconds.
NOTE: The number of nodes in the input graph is 6.
NOTE: The number of links in the input graph is 10.
NOTE: ---------------------------------------------------------------------------
NOTE: Processing CYCLE statement.
NOTE: The graph does have a cycle.
NOTE: Processing cycles used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------
STATUS=OK  CYCLE=OK
STATUS=OK  NUM_CYCLES=1  CPU_TIME=0.00  REAL_TIME=0.00
```

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

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

The result is written to the log of the procedure, as shown in Figure 2.28.

Figure 2.28 PROC OPTNET Log: Count the Number of Cycles in a Simple Directed Graph

```
NOTE: ---------------------------------------------------------------------------
NOTE: Running OPTNET version 12.3.
NOTE: ---------------------------------------------------------------------------
NOTE: Data input used 0.01 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 6.
NOTE: The number of links in the input graph is 10.
NOTE: ---------------------------------------------------------------------------
NOTE: Processing CYCLE statement.
NOTE: The graph has 7 cycles.
NOTE: Processing cycles used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------
STATUS=OK  CYCLE=OK
STATUS=OK  NUM_CYCLES=7  CPU_TIME=0.00  REAL_TIME=0.00
```

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

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

The data set Cycles now contains the first cycle found in the input graph; it is shown in Figure 2.29.

Figure 2.29 First Cycle Found in a Simple Directed Graph

```
cycle    order    node

  1        1       A
  1        2       B
  1        3       C
  1        4       A
```

The first cycle found in the input graph is shown graphically in Figure 2.30.

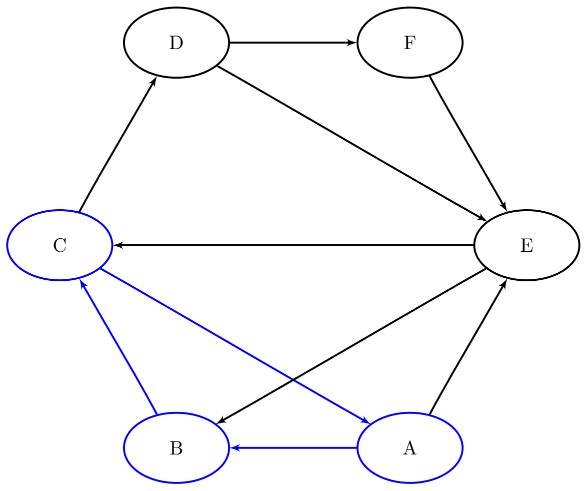


Figure 2.30 $A \rightarrow B \rightarrow C \rightarrow A$

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

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

The data set Cycles now contains all of the cycles in the input graph; it is shown in Figure 2.31.

Figure 2.31 All Cycles in a Simple Directed Graph

```
cycle    order    node

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

The cycles are shown graphically in Figure 2.32 through Figure 2.34.

Figure 2.32 Cycles

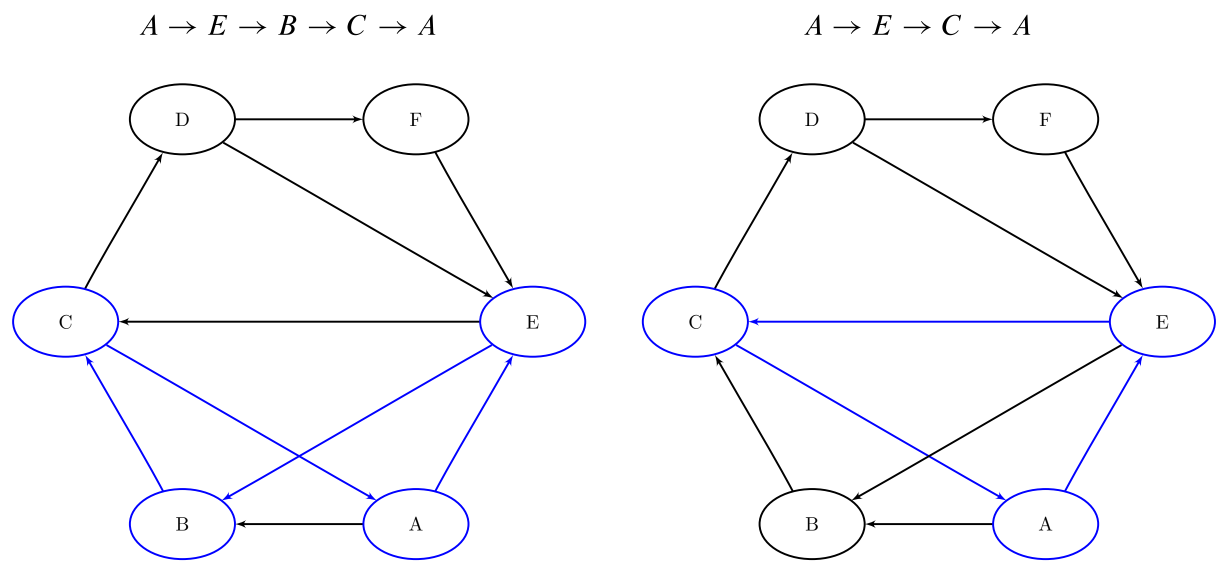

$A \rightarrow E \rightarrow B \rightarrow C \rightarrow A$
$A \rightarrow E \rightarrow C \rightarrow A$
D
F
C
E
B
A

Figure 2.33 Cycles

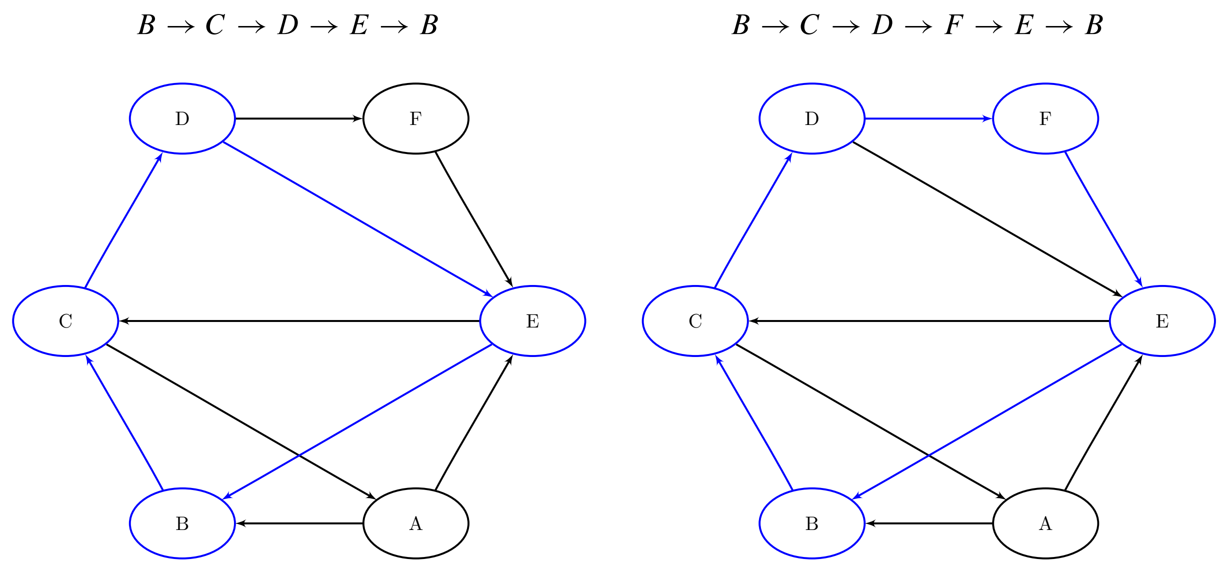
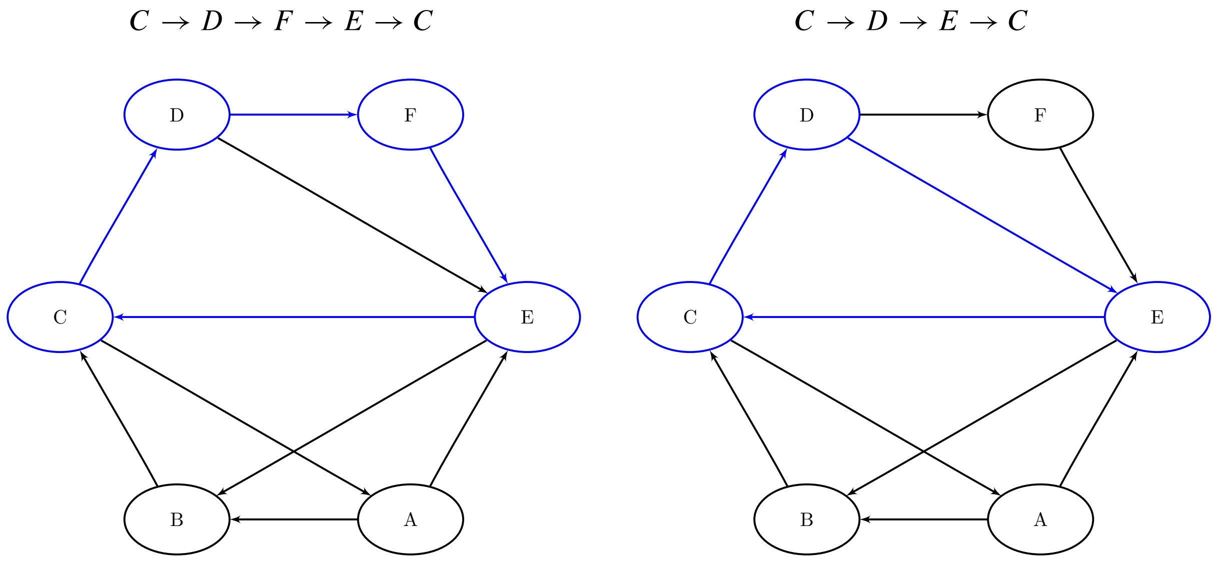

$B \rightarrow C \rightarrow D \rightarrow E \rightarrow B$
$B \rightarrow C \rightarrow D \rightarrow F \rightarrow E \rightarrow B$
D
F
C
E
B
A

Figure 2.34 Cycles



Linear Assignment (Matching)

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

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

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

This model allows for some elements of set S (workers) to go unassigned (if $|S| > |T|$). However, if $|S| < |T|$, then the following optimization problem is solved:

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

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

In PROC OPTNET, the linear assignment problem solver can be invoked by using the LINEAR_ASSIGNMENT statement. The options for this statement are described in the section "LINEAR_ASSIGNMENT Statement" on page 29. The algorithm used in PROC OPTNET for solving LAP is based on augmentation of shortest paths (Jonker and Volgenant 1987). This algorithm can be applied to either matrix data input (see the section "Matrix Input Data" on page 46) or graph data input (see the section "Graph Input Data" on page 39) as long as the graph is bipartite.

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

The linear assignment problem solver reports status information in a macro variable called _OROPTNET_LAP_. See the section "Macro Variable _OROPTNET_LAP_" on page 101 for more information about this macro variable.

For a detailed example, see "Example 2.3: Linear Assignment Problem for Minimizing Swim Times" on page 112.

Minimum Cut

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

In PROC OPTNET, the minimum cut algorithm can be invoked by using the experimental MINCUT statement. The options for this statement are described in the section "MINCUT Statement (Experimental)" on page 31. This algorithm can be used only on undirected graphs.

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

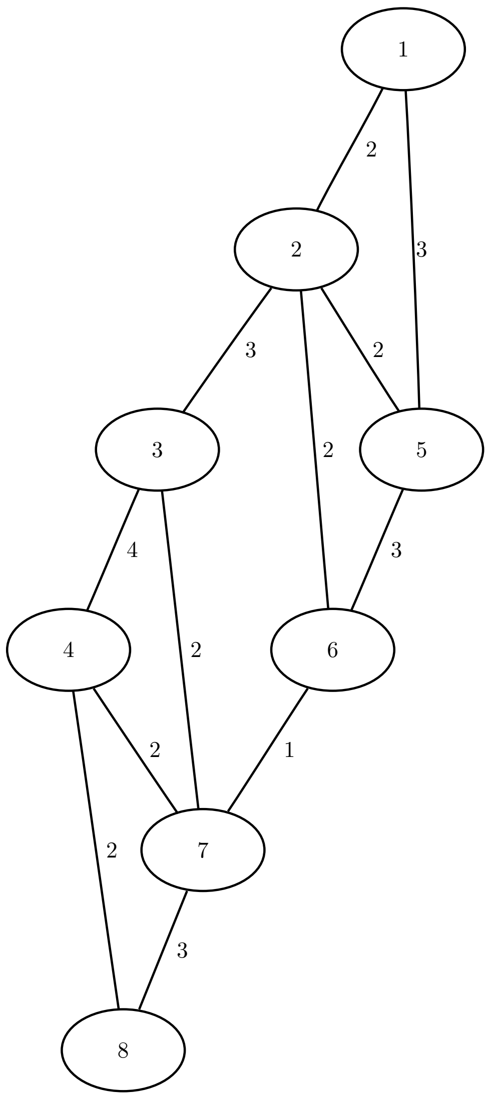
The minimum cut algorithm reports status information in a macro variable called _OROPTNET_MINCUT_. See the section "Macro Variable _OROPTNET_MINCUT_" on page 102 for more information about this macro variable.

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

Minimum Cut for a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 2.35.

Figure 2.35 A Simple Undirected Graph



The links data set can be represented as follows:

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

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

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

The progress of the procedure is shown in Figure 2.36.

Figure 2.36 PROC OPTNET Log for Minimum Cut

```
NOTE: -------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------
NOTE: Running OPTNET version 12.3.
NOTE: -------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: There were 12 observations read from the data set WORK.LINKSETIN.
NOTE: Data input used 0.01 (cpu: 0.00) seconds.
NOTE: Building the input graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The input graph storage is using 0.0 MBs of memory.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 12.
NOTE: -------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------
NOTE: Processing MINCUT statement.
NOTE: The MINCUT algorithm is experimental in this release.
NOTE: The minimum cut algorithm found 3 cuts.
NOTE: The cut 1 has weight 4.
NOTE: The cut 2 has weight 5.
NOTE: The cut 3 has weight 5.
NOTE: Processing the minimum cut used 0.00 (cpu: 0.00) seconds.
NOTE: -------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------
NOTE: Creating nodes data set output.
NOTE: Creating minimum cut data set output.
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: -------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------
NOTE: The data set WORK.NODESETOUT has 8 observations and 4 variables.
NOTE: The data set WORK.MINCUT has 6 observations and 4 variables.
STATUS=OK  MINCUT=OK
STATUS=OK  CPU_TIME=0.00  REAL_TIME=0.00
```

The data set NodeSetOut now contains the partition of the nodes for each cut, shown in Figure 2.37.

Figure 2.37 Minimum Cut Node Partition

```
node    mincut_1    mincut_2    mincut_3

  1         1           1           1
  2         1           1           0
  5         1           1           0
  3         0           1           0
  6         1           1           0
  4         0           1           0
  7         0           1           0
  8         0           0           0
```

The data set MinCut contains the links in the cut-sets for each cut. This data set is shown in Figure 2.38 along with each cut separately.

Figure 2.38 Minimum Cut Sets

```
                         mincut    from    to    weight

                            1        2      3       3
                            1        6      7       1
                            2        4      8       2
                            2        7      8       3
                            3        1      2       2
                            3        1      5       3


----------------------------------- mincut=1 -----------------------------------

                                from    to    weight

                                   2     3       3
                                   6     7       1
                              ------          ------
                              mincut             4


----------------------------------- mincut=2 -----------------------------------

                                from    to    weight

                                   4     8       2
                                   7     8       3
                              ------          ------
                              mincut             5


----------------------------------- mincut=3 -----------------------------------

                                from    to    weight

                                   1     2       2
                                   1     5       3
                              ------          ------
                              mincut             5
                                              ======
                                                14
```

Minimum Spanning Tree

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

In PROC OPTNET, the minimum spanning tree algorithm can be invoked by using the MINSPANTREE statement. The options for this statement are described in the section "MINSPANTREE Statement" on page 31. This algorithm can be used only on undirected graphs.

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

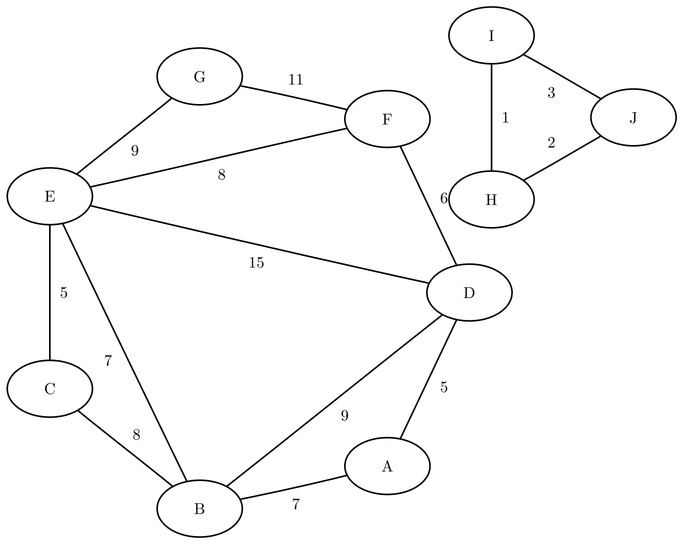
The minimum spanning tree algorithm reports status information in a macro variable called _OROPT-NET_MST_. See the section "Macro Variable _OROPTNET_MST_" on page 102 for more information about this macro variable.

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

Minimum Spanning Tree for a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 2.39.

Figure 2.39 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 MinSpan-Forest:

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

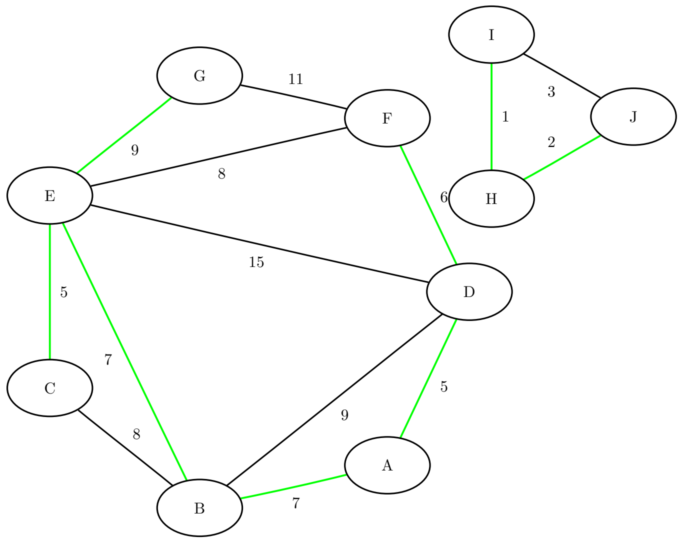
The data set MinSpanForest now contains the links that belong to a minimum spanning forest, which is shown in Figure 2.40.

Figure 2.40 Minimum Spanning Forest

```
from    to    weight

 H      I        1
 H      J        2
 C      E        5
 A      D        5
 D      F        6
 B      E        7
 A      B        7
 E      G        9
              ======
                42
```

The minimal cost links are shown in green in Figure 2.41.

Figure 2.41 Minimum Spanning Forest



For a more detailed example, see Example 2.5.

Minimum-Cost Network Flow

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

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

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

In PROC OPTNET, the minimum-cost network flow solver can be invoked by using the MINCOSTFLOW statement. The options for this statement are described in the section "MINCOSTFLOW Statement" on page 30.

The minimum-cost network flow solver reports status information in a macro variable called _OROPTNET_MCF_. See the section "Macro Variable _OROPTNET_MCF_" on page 101 for more information about this macro variable.

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

The input for the network is the standard graph input described in the section "Graph Input Data" on page 39. The links data set, which is specified in the DATA_LINKS= option in the PROC OPTNET statement, contains the following columns:

- weight defines the link cost c_{ij}
- lower defines the link lower bound l_{ij} (the default is 0)
- upper defines the link upper bound u_{ij} (the default is ∞)

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

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

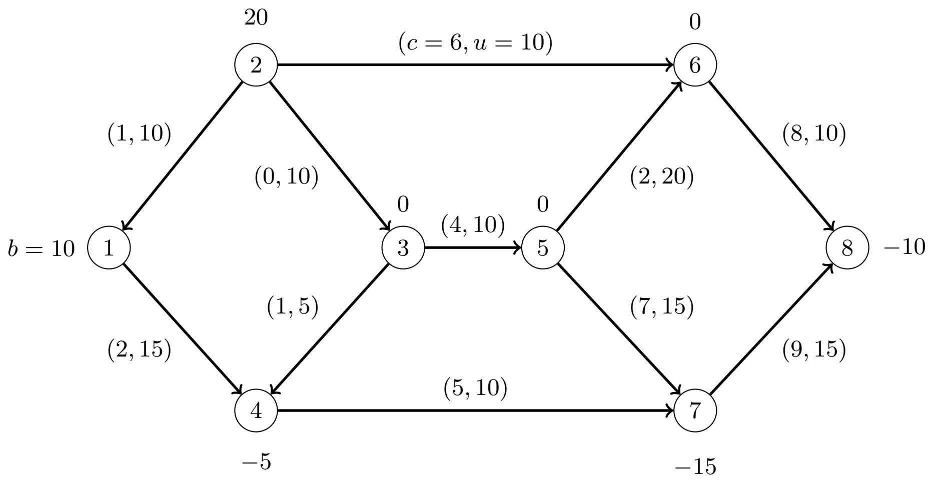
To define a pure network where the node supply must be met exactly, use the weight variable only. You do not need to specify all the node supply bounds. For any missing node, the solver uses its default values.

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

Minimum Cost Network Flow for a Simple Directed Graph

The following example demonstrates how to use the network simplex solver to find a minimum-cost flow in a directed graph. Consider the directed graph in Figure 2.42, which appears in Ahuja, Magnanti, and Orlin (1993).

Figure 2.42 Minimum-Cost Network Flow Problem: Data



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

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

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

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

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

The progress of the procedure is shown in Figure 2.43.

Figure 2.43 PROC OPTNET Log for Minimum-Cost Network Flow

```
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Running OPTNET version 12.3.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: Reading the nodes data set.
NOTE: There were 5 observations read from the data set WORK.NODESETIN.
NOTE: There were 11 observations read from the data set WORK.LINKSETIN.
NOTE: Data input used 0.01 (cpu: 0.00) seconds.
NOTE: Building the input graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The input graph storage is using 0.0 MBs of memory.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 11.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Processing MINCOSTFLOW statement.
NOTE: The network has 1 connected component.
                        Primal          Primal            Dual
      Iteration      Objective  Infeasibility  Infeasibility     Time
              1              0     20.0000000     89.0000000     0.00
              2              0     20.0000000     89.0000000     0.00
              3      5.0000000     15.0000000     84.0000000     0.00
              4      5.0000000     15.0000000     83.0000000     0.00
              5     75.0000000     15.0000000     83.0000000     0.00
              6     75.0000000     15.0000000     79.0000000     0.00
              7    130.0000000     10.0000000     76.0000000     0.00
              8    270.0000000              0              0     0.00
NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: The minimum cost network flow is 270.
NOTE: Processing the minimum cost network flow used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------
NOTE: Creating links data set output.
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: The data set WORK.LINKSETOUT has 11 observations and 5 variables.
STATUS=OK  MCF=OPTIMAL
STATUS=OPTIMAL  OBJECTIVE=270  CPU_TIME=0.00  REAL_TIME=0.00
```

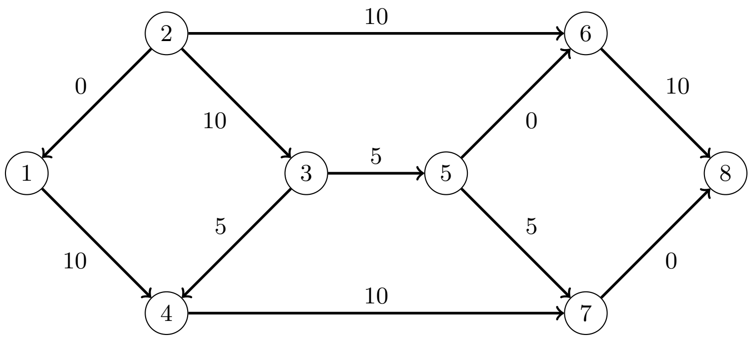
The optimal solution is displayed in Figure 2.44.

Figure 2.44 Minimum-Cost Network Flow Problem: Optimal Solution

```
Obs    from    to    upper    weight    mcf_flow

  1      1      4      15        2         10
  2      2      1      10        1          0
  3      2      3      10        0         10
  4      2      6      10        6         10
  5      3      4       5        1          5
  6      3      5      10        4          5
  7      4      7      10        5         10
  8      5      6      20        2          0
  9      5      7      15        7          5
 10      6      8      10        8         10
 11      7      8      15        9          0
```

The optimal solution is represented graphically in Figure 2.45.

Figure 2.45 Minimum-Cost Network Flow Problem: Optimal Solution



Shortest Path

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

In PROC OPTNET, shortest paths can be calculated by invoking the SHORTPATH statement. The options for this statement are described in the section "SHORTPATH Statement" on page 32.

The shortest path algorithm reports status information in a macro variable called _OROPTNET_SHORTPATH_. See the section "Macro Variable _OROPTNET_SHORTPATH_" on page 102 for more information about this macro variable.

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

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

For weighted graphs, the algorithm uses the weight variable that is defined in the links data set to evaluate a path's total weight (or cost). You can also use the WEIGHT2= option in the SHORTPATH statement to define an auxiliary weight. The auxiliary weight is not used in the algorithm to evaluate a path's total weight. It is simply calculated for the sake of reporting the total auxiliary weight for each shortest path.

Output Data Sets

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

OUT_PATHS= Data Set

This data set contains the links present in the shortest path for each of the source-sink pairs. For large graphs and a large requested number of source-sink pairs, this output data set can be extremely large. In this case, the generation of the output can sometimes take longer than the computation of the shortest paths. For example, using the U.S. road network data for the state of New York, the data contain a directed graph with 264,346 nodes. Finding the shortest path for all pairs from only one source node results in 140,969,120 observations, which is a data set of size 11 GB. Finding shortest paths for all pairs from all nodes would produce an enormous output data set.

The OUT_PATHS= data set contains the following columns:

- source: the source node label of this shortest path
- sink: the sink node label of this shortest path
- order: for this source-sink pair, the order of this link in a shortest path
- from: the *from* node label of this link in a shortest path
- to: the *to* node label of this link in a shortest path
- weight: the weight of this link in a shortest path
- weight2: the auxiliary weight of this link

OUT_WEIGHTS= Data Set

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

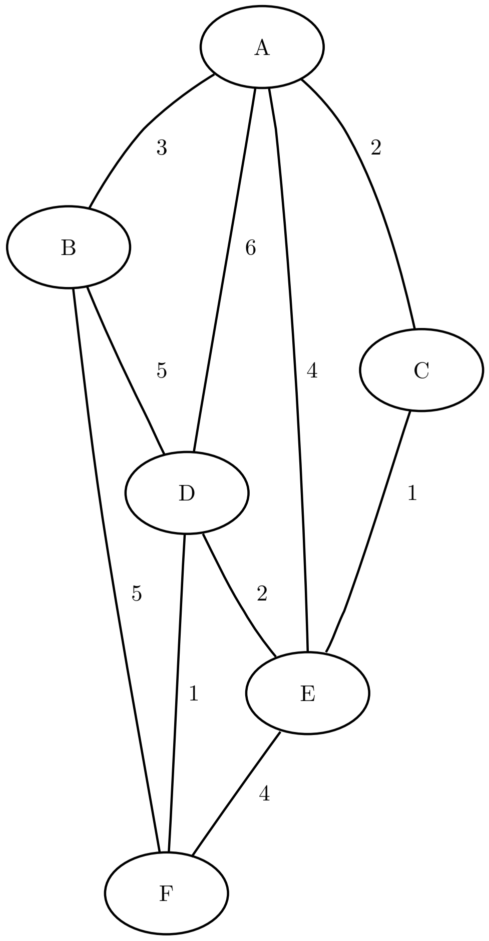
The data set contains the following columns:

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

Shortest Paths for All Pairs

This example illustrates the use of the shortest path algorithm for all source-sink pairs on the simple undirected graph G shown in Figure 2.46.

Figure 2.46 A Simple Undirected Graph G



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

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

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

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

The data set ShortPathP contains the shortest paths and is shown in Figure 2.47.

Figure 2.47 All-Pairs Shortest Paths

```
                         ShortPathP

        source    sink    order    from    to    weight

          A        B        1       A      B        3
          A        C        1       A      C        2
          A        D        1       A      C        2
          A        D        2       C      E        1
          A        D        3       E      D        2
          A        E        1       A      C        2
          A        E        2       C      E        1
          A        F        1       A      C        2
          A        F        2       C      E        1
          A        F        3       E      D        2
          A        F        4       D      F        1
          B        A        1       B      A        3
          B        C        1       B      A        3
          B        C        2       A      C        2
          B        D        1       B      D        5
          B        E        1       B      A        3
          B        E        2       A      C        2
          B        E        3       C      E        1
          B        F        1       B      F        5
          C        A        1       C      A        2
          C        B        1       C      A        2
          C        B        2       A      B        3
          C        D        1       C      E        1
          C        D        2       E      D        2
          C        E        1       C      E        1
          C        F        1       C      E        1
          C        F        2       E      D        2
          C        F        3       D      F        1
          D        A        1       D      E        2
          D        A        2       E      C        1
          D        A        3       C      A        2
          D        B        1       D      B        5
          D        C        1       D      E        2
          D        C        2       E      C        1
          D        E        1       D      E        2
          D        F        1       D      F        1
          E        A        1       E      C        1
          E        A        2       C      A        2
          E        B        1       E      C        1
          E        B        2       C      A        2
          E        B        3       A      B        3
          E        C        1       E      C        1
          E        D        1       E      D        2
          E        F        1       E      D        2
          E        F        2       D      F        1
          F        A        1       F      D        1
          F        A        2       D      E        2
          F        A        3       E      C        1
          F        A        4       C      A        2
          F        B        1       F      B        5
          F        C        1       F      D        1
          F        C        2       D      E        2
          F        C        3       E      C        1
          F        D        1       F      D        1
          F        E        1       F      D        1
          F        E        2       D      E        2
```

The data set ShortPathW contains the path weight for the shortest paths of each source-sink pair and is shown in Figure 2.48.

Figure 2.48 All-Pairs Shortest Paths Summary

```
                 ShortPathW

                               path_
             source    sink    weight

               A        B        3
               A        C        2
               A        D        5
               A        E        3
               A        F        6
               B        A        3
               B        C        5
               B        D        5
               B        E        6
               B        F        5
               C        A        2
               C        B        5
               C        D        3
               C        E        1
               C        F        4
               D        A        5
               D        B        5
               D        C        3
               D        E        2
               D        F        1
               E        A        3
               E        B        6
               E        C        1
               E        D        2
               E        F        3
               F        A        6
               F        B        5
               F        C        4
               F        D        1
               F        E        3
```

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

The following statements display only the longest shortest paths:

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

The data set ShortPathLong now contains the longest shortest paths and is shown in Figure 2.49.

Figure 2.49 Longest Shortest Path

```
                        ShortPathLong

        source    sink    order    from    to    weight

          A        F        1       A      C        2
          A        F        2       C      E        1
          A        F        3       E      D        2
          A        F        4       D      F        1
          B        E        1       B      A        3
          B        E        2       A      C        2
          B        E        3       C      E        1
          E        B        1       E      C        1
          E        B        2       C      A        2
          E        B        3       A      B        3
          F        A        1       F      D        1
          F        A        2       D      E        2
          F        A        3       E      C        1
          F        A        4       C      A        2
```

Shortest Paths for a Subset of Source-Sink Pairs

This section illustrates the use of a node subset data set, the DATA_NODES_SUB= option, and the shortest path algorithm for calculating shortest paths between a subset of source-sink pairs. The data set variables source and sink are used as indicators to specify which pairs to process. The marked source nodes define a set S, and the marked sink nodes define a set T. PROC OPTNET then calculates all the source-sink pairs in the cross product of these two sets.

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

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

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

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

The data set ShortPath contains the shortest paths and is shown in Figure 2.50.

Figure 2.50 Shortest Paths for a Subset of Source-Sink Pairs

```
                    ShortPath

 source    sink    order    from    to    weight

   A        B        1       A      B        3
   A        F        1       A      C        2
   A        F        2       C      E        1
   A        F        3       E      D        2
   A        F        4       D      F        1
   C        B        1       C      A        2
   C        B        2       A      B        3
   C        F        1       C      E        1
   C        F        2       E      D        2
   C        F        3       D      F        1
```

Shortest Paths for a Subset of Source or Sink Pairs

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

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

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

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

You can use the same PROC OPTNET call as is used in the section "Shortest Paths for a Subset of Source-Sink Pairs" on page 83 to calculate all the shortest paths from nodes B and E. The data set ShortPath contains the shortest paths and is shown in Figure 2.51.

Figure 2.51 Shortest Paths for a Subset of Source Pairs

```
                         ShortPath

   source    sink    order    from    to    weight

     B        A        1       B      A        3
     B        C        1       B      A        3
     B        C        2       A      C        2
     B        D        1       B      D        5
     B        E        1       B      A        3
     B        E        2       A      C        2
     B        E        3       C      E        1
     B        F        1       B      F        5
     E        A        1       E      C        1
     E        A        2       C      A        2
     E        B        1       E      C        1
     E        B        2       C      A        2
     E        B        3       A      B        3
     E        C        1       E      C        1
     E        D        1       E      D        2
     E        F        1       E      D        2
     E        F        2       D      F        1
```

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

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

You can use the same PROC OPTNET call again to calculate all the shortest paths to nodes B and E. The data set ShortPath contains the shortest paths and is shown in Figure 2.52.

Figure 2.52 Shortest Paths for a Subset of Sink Pairs

```
                              ShortPath

             source    sink    order    from    to    weight

               A         B       1        A      B       3
               A         E       1        A      C       2
               A         E       2        C      E       1
               B         E       1        B      A       3
               B         E       2        A      C       2
               B         E       3        C      E       1
               C         B       1        C      A       2
               C         B       2        A      B       3
               C         E       1        C      E       1
               D         B       1        D      B       5
               D         E       1        D      E       2
               E         B       1        E      C       1
               E         B       2        C      A       2
               E         B       3        A      B       3
               F         B       1        F      B       5
               F         E       1        F      D       1
               F         E       2        D      E       2
```

Shortest Paths for One Source-Sink Pair

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

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

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

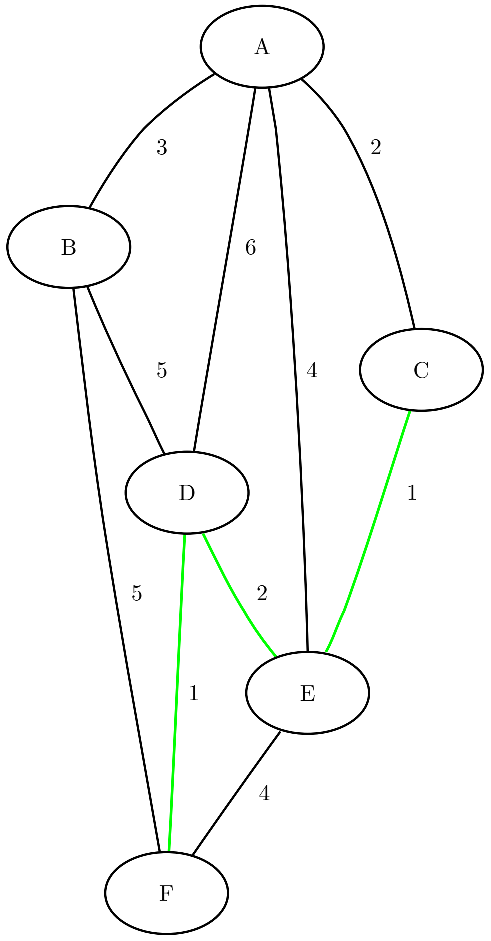
The data set ShortPath contains this shortest path and is shown in Figure 2.53.

Figure 2.53 Shortest Paths for One Source-Sink Pair

```
                              ShortPath

             source    sink    order    from    to    weight

               C         F       1        C      E       1
               C         F       2        E      D       2
               C         F       3        D      F       1
```

The shortest path is shown graphically in Figure 2.54.

Figure 2.54 Shortest Path between Nodes C and F



Shortest Paths with Auxiliary Weight Calculation

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

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

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

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

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

The data set ShortPathW contains the total path weight for shortest paths in each source-sink pair and is shown in Figure 2.55. Since the variable count in LinkSetIn is 1 for all links, the value in the output data set variable path_weights2 gives the number of links in each shortest path.

Figure 2.55 Shortest Paths Including Auxiliary Weights in Calculation

```
                              ShortPathW

                                       path_     path_
                    source    sink    weight    weight2

                      A        B         3         1
                      A        C         2         1
                      A        D         5         3
                      A        E         3         2
                      A        F         6         4
                      B        A         3         1
                      B        C         5         2
                      B        D         5         1
                      B        E         6         3
                      B        F         5         1
                      C        A         2         1
                      C        B         5         2
                      C        D         3         2
                      C        E         1         1
                      C        F         4         3
                      D        A         5         3
                      D        B         5         1
                      D        C         3         2
                      D        E         2         1
                      D        F         1         1
                      E        A         3         2
                      E        B         6         3
                      E        C         1         1
                      E        D         2         1
                      E        F         3         2
                      F        A         6         4
                      F        B         5         1
                      F        C         4         3
                      F        D         1         1
                      F        E         3         2
```

The section "Road Network Shortest Path" on page 13 shows an example of using the shortest path algorithm for minimizing travel to and from work based on traffic conditions.

Transitive Closure

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

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

In PROC OPTNET, the transitive closure algorithm can be invoked by using the TRANSITIVE_CLOSURE statement. The options for this statement are described in the section “TRANSITIVE_CLOSURE Statement” on page 34.

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

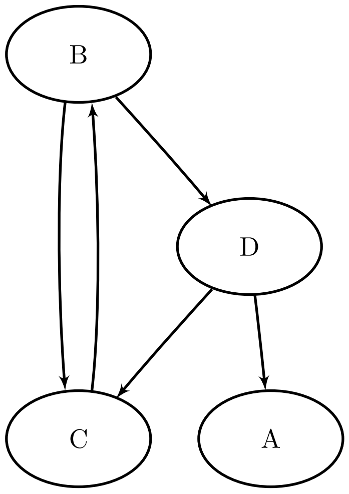
The transitive closure algorithm reports status information in a macro variable called _OROPTNET_TRANSCL_. See the section “Macro Variable _OROPTNET_TRANSCL_” on page 103 for more information about this macro variable.

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

Transitive Closure of a Simple Directed Graph

This example illustrates the use of the transitive closure algorithm on the simple directed graph G shown in Figure 2.56.

Figure 2.56 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 optnet
   graph_direction = directed
   data_links      = LinkSetIn;
   transitive_closure
      out          = TransClosure;
run;
```

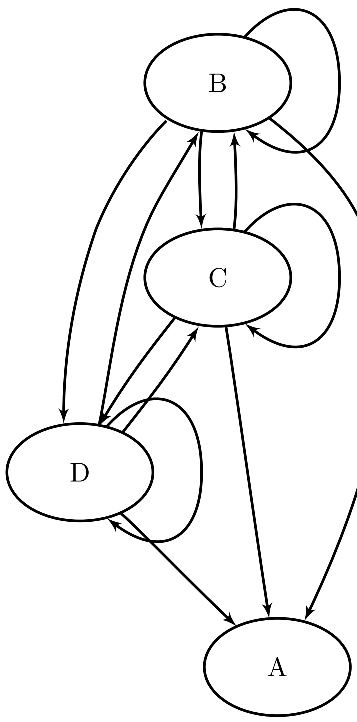
The data set TransClosure contains the transitive closure of *G* and is shown in Figure 2.57.

Figure 2.57 Transitive Closure of a Simple Directed Graph

```
                    Transitive Closure

                       from    to

                        B      C
                        B      D
                        C      B
                        D      A
                        D      C
                        C      C
                        C      D
                        B      B
                        D      B
                        D      D
                        B      A
                        C      A
```

The transitive closure of G is shown graphically in Figure 2.58.

Figure 2.58 Transitive Closure of G



For a more detailed example, see Example 2.6.

Traveling Salesman Problem

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

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

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

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

In PROC OPTNET, the traveling salesman problem solver can be invoked by using the TSP statement. The options for this statement are described in the section "TSP Statement" on page 34.

The traveling salesman problem solver reports status information in a macro variable called _OROPTNET_TSP_. See the section "Macro Variable _OROPTNET_TSP_" on page 103 for more information about this macro variable.

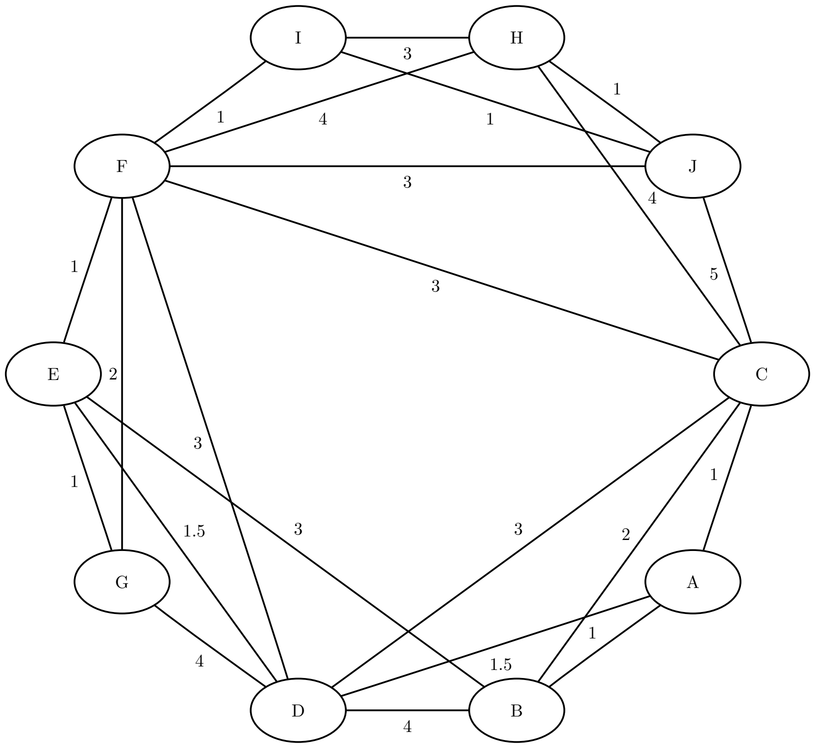
The algorithm used in PROC OPTNET for solving TSP is based on a variant of the branch-and-cut process described in (Applegate et al. 2006).

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

Traveling Salesman Problem of a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 2.59.

Figure 2.59 A Simple Undirected Graph



The links data set can be represented as follows:

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

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

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

The progress of the procedure is shown in Figure 2.60.

Figure 2.60 PROC OPTNET Log: Optimal Traveling Salesman Tour of a Simple Undirected Graph

```
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Running OPTNET version 12.3.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: There were 22 observations read from the data set WORK.LINKSETIN.
NOTE: Data input used 0.01 (cpu: 0.02) seconds.
NOTE: Building the input graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The input graph storage is using 0.0 MBs of memory.
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 22.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Processing TSP statement.
NOTE: The initial TSP heuristics found a tour with cost 16 using 0.00 (cpu:
      0.00) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       2     16.0000000     16.0000000    0.00%       0
             0       0       2     16.0000000     16.0000000    0.00%       0
NOTE: Optimal.
NOTE: Objective = 16.
NOTE: Processing the traveling salesman problem used 0.03 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Creating nodes data set output.
NOTE: Creating traveling salesman data set output.
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: The data set WORK.NODESETOUT has 10 observations and 2 variables.
NOTE: The data set WORK.TSPTOUR has 10 observations and 3 variables.
STATUS=OK  TSP=OPTIMAL
STATUS=OPTIMAL  OBJECTIVE=16  RELATIVE_GAP=0  ABSOLUTE_GAP=0
PRIMAL_INFEASIBILITY=0  BOUND_INFEASIBILITY=0  INTEGER_INFEASIBILITY=0
BEST_BOUND=16  NODES=1  ITERATIONS=15  CPU_TIME=0.00  REAL_TIME=0.03
```

The data set NodeSetOut now contains a sequence of nodes in the optimal tour and is shown in Figure 2.61.

Figure 2.61 Nodes in the Optimal Traveling Salesman Tour

```
                    Traveling Salesman Problem

                                    tsp_
                          node     order

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

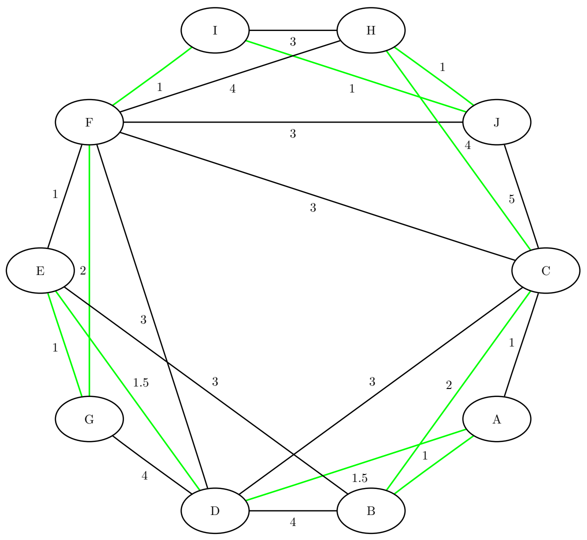
The data set TSPTour now contains the links in the optimal tour and is shown in Figure 2.62.

Figure 2.62 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 2.63.

Figure 2.63 Optimal Traveling Salesman Tour



Macro Variable _OROPTNET_

The OPTNET procedure defines a macro variable named _OROPTNET_. This variable contains a character string that indicates the status of the OPTNET procedure upon termination. The various terms of the variable are interpreted as follows:

STATUS

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

OK	The procedure terminated normally.
OUT_OF_MEMORY	Insufficient memory was allocated to the procedure.
ERROR	The procedure encountered an error.

BICONCOMP

indicates the status of the biconnected components algorithm at termination. This algorithm is described in the section "Biconnected Components and Articulation Points" on page 47. The BICONCOMP term can take one of the following values:

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

CLIQUE

indicates the status of the clique-finding algorithms at termination. These algorithms are described in the section "Clique" on page 51. The CLIQUE term can take one of the following values:

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

CONCOMP

indicates the status of the connected components algorithm at termination. This algorithm is described in the section "Connected Components" on page 54. The CONCOMP term can take one of the following values:

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

CYCLE

indicates the status of the cycle detection algorithm at termination. This algorithm is described in the section "Cycle" on page 59. The CYCLE term can take one of the following values:

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

LAP

indicates the status of the linear assignment solver at termination. This solver is described in the section "Linear Assignment (Matching)" on page 65. The LAP term can take one of the following values:

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

MCF

indicates the status of the minimum-cost network flow solver at termination. This solver is described in the section "Minimum-Cost Network Flow" on page 73. The MCF term can take one of the following values:

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

MINCUT

indicates the status of the minimum cut algorithm at termination. This algorithm is described in the section “Minimum Cut” on page 66. The MINCUT term can take one of the following values:

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

MINSPANTREE

indicates the status of the minimum spanning tree solver at termination. This solver is described in the section “Minimum Spanning Tree” on page 70. The MINSPANTREE term can take one of the following values:

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

SHORTPATH

indicates the status of the shortest path algorithms at termination. These algorithms are described in the section “Shortest Path” on page 77. The SHORTPATH term can take one of the following values:

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

TRANSITIVE_CLOSURE

indicates the status of the transitive closure algorithm at termination. This algorithm is described in the section “Transitive Closure” on page 89. The TRANSITIVE_CLOSURE term can take one of the following values:

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

TSP

indicates the status of the traveling salesman problem solver at termination. This algorithm is described in the section “Traveling Salesman Problem” on page 91. The TSP term can take one of the following values:

OPTIMAL	The solution is optimal.
OPTIMAL_AGAP	The solution is optimal within the absolute gap specified in the ABSOBJGAP= option.
OPTIMAL_RGAP	The solution is optimal within the relative gap specified in the RELOBJGAP= option.
OPTIMAL_COND	The solution is optimal, but some infeasibilities (primal, bound, or integer) exceed tolerances due to scaling or choice of a small INTTOL= value.

TARGET	The solution is not worse than the target specified in the TARGET= option.
INFEASIBLE	The problem is infeasible.
UNBOUNDED	The problem is unbounded.
INFEASIBLE_OR_UNBOUNDED	The problem is infeasible or unbounded.
SOLUTION_LIM	The solver reached the maximum number of solutions specified in the MAXSOLS= option.
NODE_LIM_SOL	The solver reached the maximum number of nodes specified in the MAXNODES= option and found a solution.
NODE_LIM_NOSOL	The solver reached the maximum number of nodes specified in the MAXNODES= option and did not find a solution.
TIME_LIM_SOL	The solver reached the execution time limit specified in the MAXTIME= option and found a solution.
TIME_LIM_NOSOL	The solver reached the execution time limit specified in the MAXTIME= option and did not find a solution.
HEURISTIC_SOL	The solver used only heuristics and found a solution.
HEURISTIC_NOSOL	The solver used only heuristics and did not find a solution.
ABORT_SOL	The solver was stopped by the user but still found a solution.
ABORT_NOSOL	The solver was stopped by the user and did not find a solution.
OUTMEM_SOL	The solver ran out of memory but still found a solution.
OUTMEM_NOSOL	The solver ran out of memory and either did not find a solution or failed to output the solution due to insufficient memory.
FAIL_SOL	The solver stopped due to errors but still found a solution.
FAIL_NOSOL	The solver stopped due to errors and did not find a solution.

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

Macro Variable _OROPTNET_BICONCOMP_

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

STATUS

indicates the status of the algorithm at termination. The STATUS term takes the same value as the term BICONCOMP in the _OROPTNET_ macro as defined in the section “Macro Variable _OROPTNET_” on page 96.

NUM_COMPONENTS

indicates the number of biconnected components found by the algorithm.

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

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

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

Macro Variable _OROPTNET_CLIQUE_

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

STATUS
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term CLIQUE in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 96.

NUM_CLIQUES
indicates the number of cliques found by the algorithm.

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

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

Macro Variable _OROPTNET_CONCOMP_

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

STATUS
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term CONCOMP in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 96.

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

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

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

Macro Variable _OROPTNET_CYCLE_

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

STATUS
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term CYCLE in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 96.

NUM_CYCLES
indicates the number of cycles found by the algorithm.

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

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

Macro Variable _OROPTNET_LAP_

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

STATUS
indicates the status of the solver at termination. The STATUS term takes the same value as the term LAP in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 96.

OBJECTIVE
indicates the total weight of the minimum linear assignment.

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

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

Macro Variable _OROPTNET_MCF_

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

STATUS
indicates the status of the solver at termination. The STATUS term takes the same value as the term MCF in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 96.

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

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

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

Macro Variable _OROPTNET_MINCUT_

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

STATUS
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term MINCUT in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 96.

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

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

Macro Variable _OROPTNET_MST_

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

STATUS
indicates the status of the solver at termination. The STATUS term takes the same value as the term MINSPANTREE in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 96.

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

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

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

Macro Variable _OROPTNET_SHORTPATH_

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

STATUS
: indicates the status of the algorithm at termination. The STATUS term takes the same value as the term SHORTPATH in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 96.

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

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

Macro Variable _OROPTNET_TRANSCL_

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

STATUS
: indicates the status of the algorithm at termination. The STATUS term takes the same value as the term TRANSITIVE_CLOSURE in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 96.

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

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

Macro Variable _OROPTNET_TSP_

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

STATUS
: indicates the status of the solver at termination. The STATUS term takes the same value as the term TSP in the _OROPTNET_ macro as defined in the section "Macro Variable _OROPTNET_" on page 96.

OBJECTIVE
: indicates the objective value obtained by the solver at termination.

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

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

ABSOLUTE_GAP
specifies the absolute gap between the best integer objective (BestInteger) and the objective of the best remaining node (BestBound) upon termination of the solver. The absolute gap is equal to

$$| \text{BestInteger} - \text{BestBound} |$$

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

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

INTEGER_INFEASIBILITY
indicates the maximum (absolute) violation of the integrality of integer variables that are returned by the solver.

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

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

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

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

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

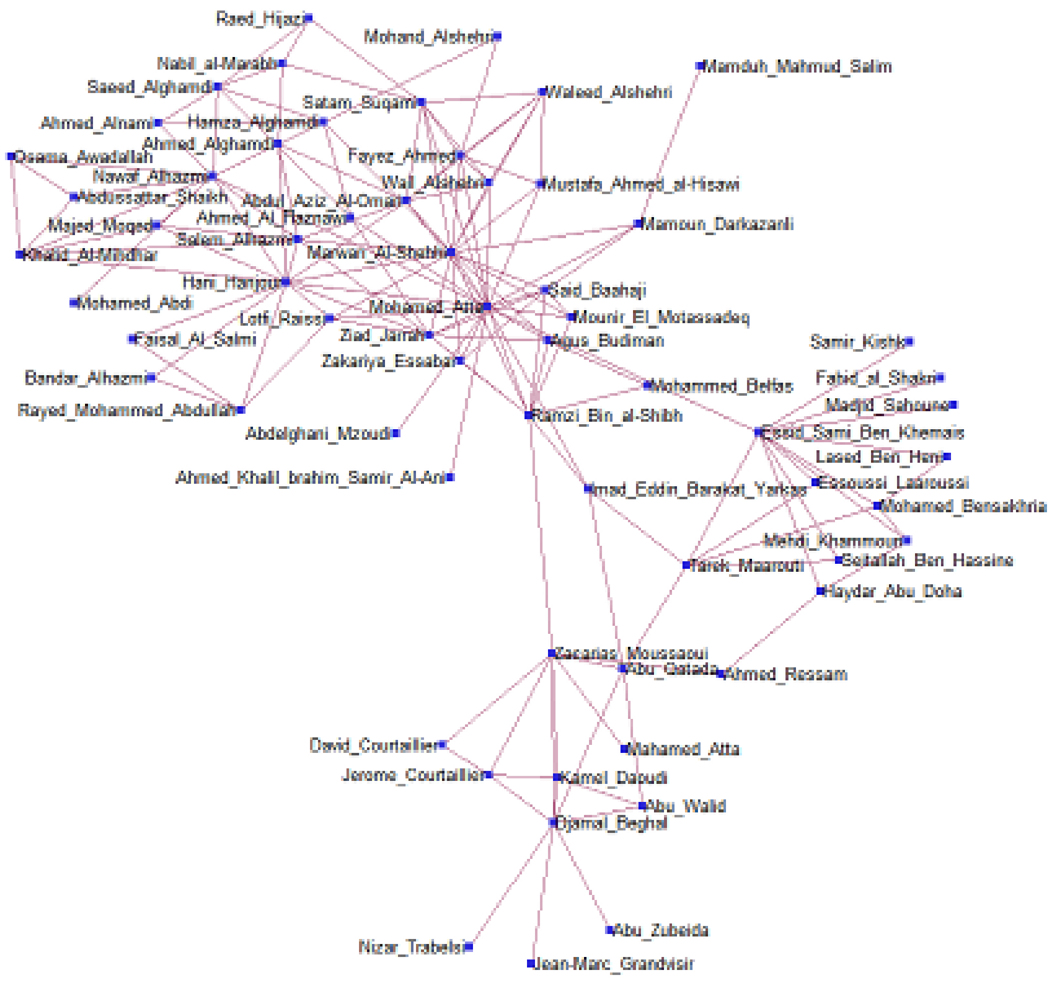
NOTE: The time reported in PRESOLVE_TIME and SOLUTION_TIME is either CPU time (default) or real time. The type is determined by the TIMETYPE= option.

Examples: OPTNET Procedure

Example 2.1: Articulation Points in a Terrorist Network

This example considers the terrorist communications network from the attacks on the U.S. on September 11, 2001, described in Krebs 2002. Figure 2.64 shows this network, which was constructed after the attacks, based on collected intelligence information. The image was created using SAS/GRAPH® Network Visualization Workshop 2.1 (see the *SAS/GRAPH: Network Visualization Workshop User's Guide*).

Figure 2.64 Terrorist Communications Network from 9/11



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

```
data LinkSetInTerror911;
   length from $25 to $32;
   input from to;
   datalines;
Abu_Zubeida                 Djamal_Beghal
Jean-Marc_Grandvisir        Djamal_Beghal
Nizar_Trabelsi              Djamal_Beghal
Abu_Walid                   Djamal_Beghal
Abu_Qatada                  Djamal_Beghal
Zacarias_Moussaoui          Djamal_Beghal
Jerome_Courtaillier         Djamal_Beghal
Kamel_Daoudi                Djamal_Beghal
Abu_Walid                   Kamel_Daoudi
Abu_Walid                   Abu_Qatada
Kamel_Daoudi                Zacarias_Moussaoui
```

```
   Kamel_Daoudi                Jerome_Courtaillier
   Jerome_Courtaillier         Zacarias_Moussaoui
   Jerome_Courtaillier         David_Courtaillier
   Zacarias_Moussaoui          David_Courtaillier
   Zacarias_Moussaoui          Ahmed_Ressam
   Zacarias_Moussaoui          Abu_Qatada
   Zacarias_Moussaoui          Ramzi_Bin_al-Shibh
   Zacarias_Moussaoui          Mahamed_Atta
   Ahmed_Ressam                Haydar_Abu_Doha
   Mehdi_Khammoun              Haydar_Abu_Doha
   Essid_Sami_Ben_Khemais      Haydar_Abu_Doha
   Mehdi_Khammoun              Essid_Sami_Ben_Khemais
   Mehdi_Khammoun              Mohamed_Bensakhria
   ...
   ;
```

Suppose that this communications network had been discovered before the attack on 9/11. If the investigators' goal was to disrupt the flow of communication between different groups within the organization, then they would want to focus on the people who are articulation points in the network.

To find the articulation points, use the following statements:

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

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

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

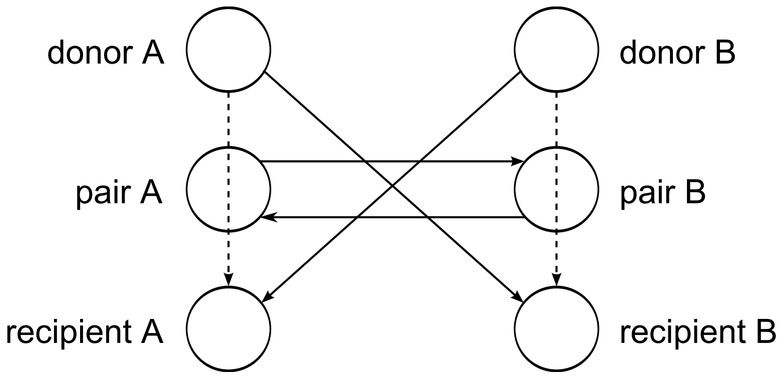
Output 2.1.1 Articulation Points of Terrorist Communications Network from 9/11

```
node                       artpoint

Djamal_Beghal                  1
Zacarias_Moussaoui             1
Essid_Sami_Ben_Khemais         1
Mohamed_Atta                   1
Mamoun_Darkazanli              1
Nawaf_Alhazmi                  1
```

Example 2.2: Cycle Detection for Kidney Donor Exchange

This example looks at an application of cycle detection to help create a kidney donor exchange. Suppose someone needs a kidney transplant and a family member is willing to donate one. If the donor and recipient are incompatible (because of blood types, tissue mismatch, and so on), the transplant cannot happen. Now suppose two donor-recipient pairs A and B are in this situation, but donor A is compatible with recipient B and donor B is compatible with recipient A. Then two transplants can take place in a two-way swap, shown graphically in Figure 2.65. More generally, an n-way swap can be performed involving n donors and n recipients (Willingham 2009).

Figure 2.65 Kidney Donor Exchange Two-Way Swap



To model this problem, define a directed graph as follows. Each node is an incompatible donor-recipient pair. Link (i, j) exists if the donor from node i is compatible with the recipient from node j. The link weight is a measure of the quality of the match. By introducing dummy links with weight 0, you can also include altruistic donors with no recipients, or recipients without donors. The idea is to find a maximum weight node-disjoint union of directed cycles. You want the union to be node-disjoint so that no kidney is donated more than once, and you want cycles so that the donor from node i gives up a kidney if and only if the recipient from node i receives a kidney.

Without any other constraints, the problem could be solved as a linear assignment problem, as described in the section "Linear Assignment (Matching)" on page 65. But doing so would allow arbitrarily long cycles in the solution. Because of practical considerations (such as travel) and to mitigate risk, each cycle must have no more than L links. The kidney exchange problem is to find a maximum weight node-disjoint union of short directed cycles.

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

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

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

The following statements use PROC OPTNET to generate all cycles with length greater than or equal to 2 and less than or equal to 10:

```
/* generate all cycles with 2 <= length <= max_length */
%let max_length = 10;
proc optnet
   loglevel        = moderate
   graph_direction = directed
   data_links      = LinkSetIn;
   cycle
      minLength    = 2
      maxLength    = &max_length
      out          = Cycles
      mode         = all_cycles;
run;
%put &_OROPTNET_;
%put &_OROPTNET_CYCLE_;
```

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

Output 2.2.1 Cycles for Kidney Donor Exchange PROC OPTNET Log

```
NOTE: -------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------
NOTE: Running OPTNET version 12.3.
NOTE: -------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: There were 194 observations read from the data set WORK.LINKSETIN.
NOTE: Data input used 0.01 (cpu: 0.00) seconds.
NOTE: Building the input graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The input graph storage is using 0.0 MBs of memory.
NOTE: The number of nodes in the input graph is 97.
NOTE: The number of links in the input graph is 194.
NOTE: -------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------
NOTE: Processing CYCLE statement.
NOTE: The graph has 224 cycles.
NOTE: Processing cycles used 5.59 (cpu: 5.58) seconds.
NOTE: -------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------
NOTE: Creating cycle data set output.
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: -------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------
NOTE: The data set WORK.CYCLES has 2124 observations and 3 variables.
STATUS=OK  CYCLE=OK
STATUS=OK  NUM_CYCLES=224  CPU_TIME=5.58  REAL_TIME=5.59
```

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

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

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

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

The constraint (incomp_pair) ensures that each node (incompatible pair) in the graph is intersected at most once. That is, a donor can donate a kidney only once. You can use PROC OPTMODEL to solve this mixed integer linear programming problem as follows:

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

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

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

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

   /* call solver */
   solve with milp;

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

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

Output 2.2.2 Cycles for Kidney Donor Exchange PROC OPTMODEL Log

```
NOTE: There were 194 observations read from the data set WORK.LINKSETIN.
NOTE: There were 1900 observations read from the data set WORK.CYCLES0.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 224 variables (0 free, 0 fixed).
NOTE: The problem has 224 binary and 0 integer variables.
NOTE: The problem has 63 linear constraints (63 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 1900 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 0 variables and 35 constraints.
NOTE: The MILP presolver removed 518 constraint coefficients.
NOTE: The MILP presolver modified 116 constraint coefficients.
NOTE: The presolved problem has 224 variables, 28 constraints, and 1382
      constraint coefficients.
NOTE: The MILP solver is called.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       3     22.7780692   1080.2049611   97.89%       0
             0       1       3     22.7780692     26.5638757   14.25%       0
             0       1       4     23.2747070     26.0203249   10.55%       0
             0       1       4     23.2747070     26.0203023   10.55%       0
             0       1       4     23.2747070     26.0202987   10.55%       0
             0       1       6     26.0202871     26.0202871    0.00%       0
NOTE: The MILP solver added 5 cuts with 599 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 26.020287142.
NOTE: The data set WORK.SOLUTION has 6 observations and 2 variables.
STATUS=OK ALGORITHM=BAC SOLUTION_STATUS=OPTIMAL OBJECTIVE=26.020287142
RELATIVE_GAP=0 ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=0 BOUND_INFEASIBILITY=0
INTEGER_INFEASIBILITY=0 BEST_BOUND=26.020287142 NODES=1 ITERATIONS=110
PRESOLVE_TIME=0.03 SOLUTION_TIME=0.11
```

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

Output 2.2.3 Maximum Quality Solution for Kidney Donor Exchange

```
                  cycle_
          c       weight

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

Example 2.3: Linear Assignment Problem for Minimizing Swim Times

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

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

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

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

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

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

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

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

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

```
NOTE: --------------------------------------------------------------------------
NOTE: Running OPTNET version 12.3.
NOTE: --------------------------------------------------------------------------
NOTE: The number of columns in the input matrix is 4.
NOTE: The number of rows in the input matrix is 6.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: --------------------------------------------------------------------------
NOTE: Processing LINEAR_ASSIGNMENT statement.
NOTE: The minimum cost linear assignment is 111.5.
NOTE: Processing the linear assignment problem used 0.00 (cpu: 0.00) seconds.
NOTE: --------------------------------------------------------------------------
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: --------------------------------------------------------------------------
NOTE: The data set WORK.LINEARASSIGNF has 4 observations and 4 variables.
STATUS=OK  LAP=OPTIMAL
STATUS=OPTIMAL  OBJECTIVE=111.5  CPU_TIME=0.00  REAL_TIME=0.00
```

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

```
NOTE: --------------------------------------------------------------------------
NOTE: Running OPTNET version 12.3.
NOTE: --------------------------------------------------------------------------
NOTE: The number of columns in the input matrix is 4.
NOTE: The number of rows in the input matrix is 4.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: --------------------------------------------------------------------------
NOTE: Processing LINEAR_ASSIGNMENT statement.
NOTE: The minimum cost linear assignment is 96.6.
NOTE: Processing the linear assignment problem used 0.00 (cpu: 0.00) seconds.
NOTE: --------------------------------------------------------------------------
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: --------------------------------------------------------------------------
NOTE: The data set WORK.LINEARASSIGNM has 4 observations and 4 variables.
STATUS=OK  LAP=OPTIMAL
STATUS=OPTIMAL  OBJECTIVE=96.6  CPU_TIME=0.00  REAL_TIME=0.00
```

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

Output 2.3.3 Optimal Assignments for Best Female Swim Times

```
name      sex    assign     cost

Karen      F     breast     32.6
Jan        F     fly        27.1
Carol      F     free       24.0
Ellen      F     back       27.8
                           =====
                           111.5
```

Output 2.3.4 Optimal Assignments for Best Male Swim Times

```
name        sex    assign    cost

Jim          M     free      22.4
Mike         M     breast    24.0
Sam          M     back      27.2
Clayton      M     fly       23.0
                             ====
                             96.6
```

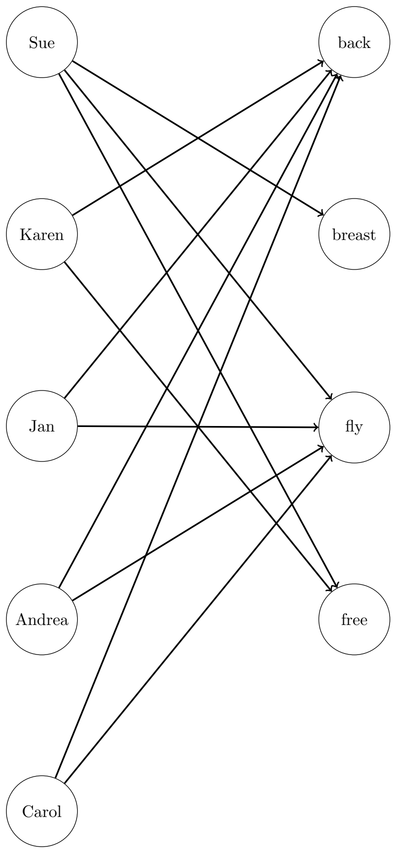
Example 2.4: Linear Assignment Problem, Sparse Format versus Dense Format

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

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

Recall that the linear assignment problem can also be interpreted as the minimum-weight matching in a bipartite graph. The eligible assignments define links between the rows (swimmers) and the columns (strokes), as in Figure 2.66.

Figure 2.66 Bipartite Graph for Linear Assignment Problem



Because of this, you can represent the same data in RelayTimesMatrix with a links data set as follows:

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

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

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

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

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

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

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

Output 2.4.1 Optimal Assignments for Swim Times (Dense Input)

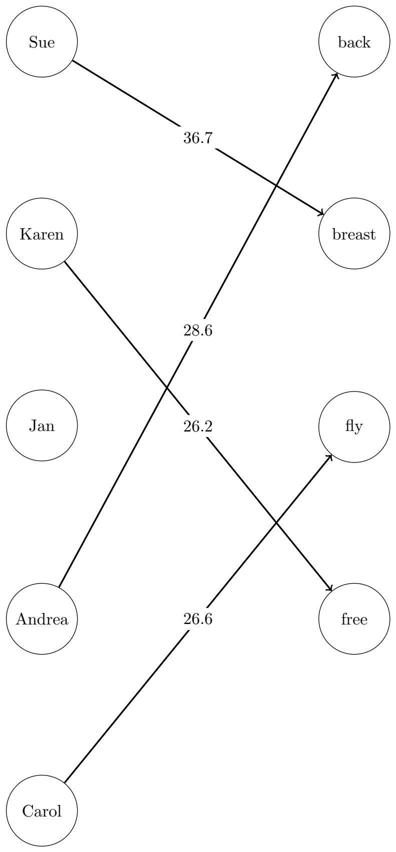
```
name      sex    assign     cost

Sue        F     breast     36.7
Karen      F     free       26.2
Andrea     F     back       28.6
Carol      F     fly        26.6
                           =====
                           118.1
```

Output 2.4.2 Optimal Assignments for Swim Times (Sparse Input)

```
name      attr       cost

Sue       breast     36.7
Karen     free       26.2
Andrea    back       28.6
Carol     fly        26.6
                    =====
                    118.1
```

The optimal assignments are shown graphically in Figure 2.67.

Figure 2.67 Optimal Assignments for Swim Times

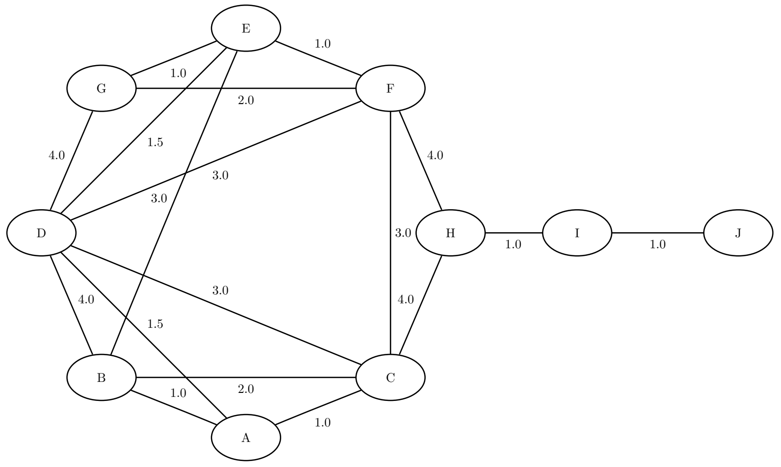


For large problems where a number of links are forbidden, the sparse format can be faster and can save a great deal of memory. Consider an example that uses the format of the DATA_MATRIX= option with 15,000 columns ($|S| = 15,000$) and 4,000 rows ($|T| = 4,000$). To store the dense matrix in memory, PROC OPTNET needs to allocate approximately $|S| \cdot |T| \cdot 8/1024/1024 = 457$ MB. If the data have mostly ineligible links, then the sparse (graph) format that uses the DATA_LINKS= option is much more efficient with respect to memory. For example, if the data have only 5% of the eligible links ($15,000 \cdot 4,000 \cdot 0.05 = 3,000,000$), then the dense storage would still need 457 MB. The sparse storage for the same example needs approximately $|S| \cdot |T| \cdot 0.05 \cdot 12/1024/1024 = 34$ MB. If the problem is fully dense (all links are eligible), then the dense format that uses the DATA_MATRIX= option is the most efficient.

Example 2.5: Minimum Spanning Tree for Computer Network Topology

Consider the problem of designing a small network of computers. In designing the network, the goal is to make sure that each machine in the office can reach every other machine. To accomplish this goal, Ethernet lines must be constructed and run between the machines. The construction costs for each possible link are based approximately on distance and are shown in Figure 2.68. Besides distance, the costs also reflect some restrictions due to physical boundaries. To connect all the machines in the office at minimal cost, you need to find a minimum spanning tree on the network of possible links.

Figure 2.68 Potential Office Computer Network



Define the link data set as follows:

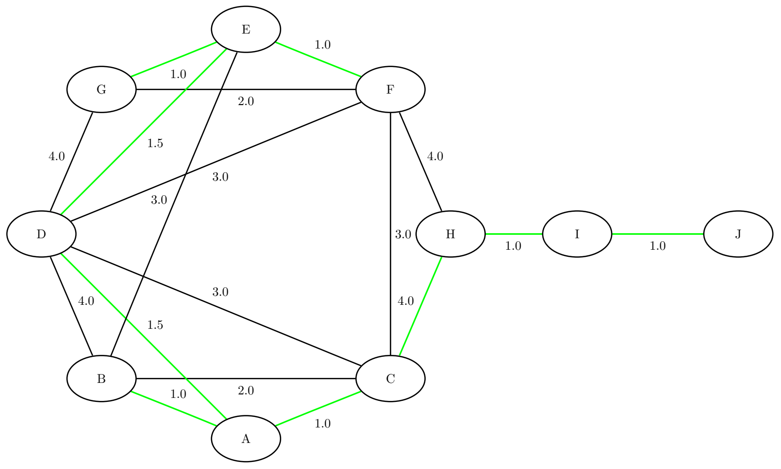
```
data LinkSetInCompNet;
   input from $ to $ weight @@;
   datalines;
A B 1.0  A C 1.0  A D 1.5  B C 2.0  B D 4.0
B E 3.0  C D 3.0  C F 3.0  C H 4.0  D E 1.5
D F 3.0  D G 4.0  E F 1.0  E G 1.0  F G 2.0
F H 4.0  H I 1.0  I J 1.0
;
```

The following statements find a minimum spanning tree:

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

Output 2.5.1 shows the resulting data set MinSpanTree, which is displayed graphically in Figure 2.69 with the minimal cost links shown in green.

Figure 2.69 Minimum Spanning Tree for Office Computer Network



Output 2.5.1 Minimum Spanning Tree of a Computer Network Topology

```
from    to    weight

 H      I       1.0
 E      G       1.0
 E      F       1.0
 A      B       1.0
 A      C       1.0
 I      J       1.0
 D      E       1.5
 A      D       1.5
 C      H       4.0
              ======
               13.0
```

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

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

The bug tracking system needs to check for two situations as users declare a bug to be a DUP. The first situation is called a *circular dependence*. Consider bugs A, B, C, and D in the tracking system. The first user declares that A is a DUP of B and that C is a DUP of D. Then, a second user declares that B is a DUP of C, and a third user declares that D is a DUP of A. You now have a circular dependence, and no primary bug is defined on which the development team should focus. You can easily see this circular dependence in the graph representation, because $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$. Finding such circular dependencies can be done using cycle detection, which is described in the section "Cycle" on page 59. However, the second situation that needs to be checked is more general. If a user declares that A is a DUP of B and another user declares that B is a DUP of C, this chain of duplicates is already an issue. The bug tracking system needs to provide one primary bug to which the rest of the bugs are duplicated. The existence of these chains can be identified by calculating the transitive closure of the directed graph that is defined by the DUP links.

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

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

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

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

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

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

The progress of the procedure is shown in Output 2.6.1.

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

```
NOTE: -------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------
NOTE: Running OPTNET version 12.3.
NOTE: -------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: There were 12 observations read from the data set WORK.DEFECTLINKS.
NOTE: Data input used 0.01 (cpu: 0.00) seconds.
NOTE: Building the input graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The input graph storage is using 0.0 MBs of memory.
NOTE: The number of nodes in the input graph is 16.
NOTE: The number of links in the input graph is 12.
NOTE: -------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------
NOTE: Processing CYCLE statement.
NOTE: The graph has 1 cycle.
NOTE: Processing cycles used 0.00 (cpu: 0.00) seconds.
NOTE: -------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------
NOTE: Processing TRANSITIVE_CLOSURE statement.
NOTE: Processing the transitive closure used 0.00 (cpu: 0.00) seconds.
NOTE: -------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------
NOTE: Creating transitive closure data set output.
NOTE: Creating cycle data set output.
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: -------------------------------------------------------------------------
NOTE: -------------------------------------------------------------------------
NOTE: The data set WORK.CYCLES has 4 observations and 3 variables.
NOTE: The data set WORK.TRANSCLOSURE has 20 observations and 2 variables.
STATUS=OK  CYCLE=OK  TRANSITIVE_CLOSURE=OK
STATUS=OK  NUM_CYCLES=1  CPU_TIME=0.00  REAL_TIME=0.00
STATUS=OK  CPU_TIME=0.00  REAL_TIME=0.00
NOTE: Table WORK.CHAINS created, with 8 rows and 2 columns.
```

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

Output 2.6.2 Cycle in Bug Tracking System

```
cycle    order      node

  1        1      S0152674
  1        2      S0153280
  1        3      S0153307
  1        4      S0152674
```

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

Output 2.6.3 Chains in Bug Tracking System

```
               linked
defectId       Defect

S0152674     S0152674
S0152674     S0153307
S0153280     S0152674
S0153280     S0153280
S0153307     S0153280
S0153307     S0153307
S0162973     S0165405
S0325026     S0643230
```

Example 2.7: Traveling Salesman Tour through US Capital Cities

Consider a cross-country trip where you want to travel the fewest miles to visit all of the capital cities in all US states except Alaska and Hawaii. Finding the optimal route is an instance of the traveling salesman problem, which is described in section "Traveling Salesman Problem" on page 91.

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

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

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

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

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

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

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

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

```
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Running OPTNET version 12.3.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: There were 1176 observations read from the data set WORK.CITIESDIST.
NOTE: Data input used 0.01 (cpu: 0.00) seconds.
NOTE: Building the input graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The input graph storage is using 0.1 MBs of memory.
NOTE: The number of nodes in the input graph is 49.
NOTE: The number of links in the input graph is 1176.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Processing TSP statement.
NOTE: The initial TSP heuristics found a tour with cost 10645.918753 using 0.22
      (cpu: 0.20) seconds.
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
          Node  Active    Sols    BestInteger      BestBound      Gap    Time
             0       1       1  10645.9187534  10040.5139714    6.03%       0
             0       1       1  10645.9187534  10241.6970024    3.95%       0
             0       1       1  10645.9187534  10262.9074205    3.73%       0
             0       1       1  10645.9187534  10293.2995080    3.43%       0
             0       1       1  10645.9187534  10350.0790852    2.86%       0
             0       1       1  10645.9187534  10549.5506188    0.91%       0
             0       1       1  10645.9187534  10576.0823291    0.66%       0
             0       1       1  10645.9187534  10590.3709358    0.52%       0
             0       1       1  10645.9187534  10590.8162090    0.52%       0
             0       1       1  10645.9187534  10590.9748294    0.52%       0
             0       1       1  10645.9187534  10607.8528157    0.36%       0
             0       1       6  10645.9187534  10607.8528157    0.36%       0
NOTE: The MILP solver added 16 cuts with 4213 cut coefficients at the root.
             1       1       7  10627.7543183  10607.8528157    0.19%       0
             2       0       7  10627.7543183  10627.7543183    0.00%       0
NOTE: Optimal.
NOTE: Objective = 10627.754318.
NOTE: Processing the traveling salesman problem used 0.35 (cpu: 0.30) seconds.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: Creating nodes data set output.
NOTE: Creating traveling salesman data set output.
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------------------
NOTE: ---------------------------------------------------------------------------
NOTE: The data set WORK.TSPTOURNODES has 49 observations and 2 variables.
NOTE: The data set WORK.TSPTOURLINKS has 49 observations and 3 variables.
STATUS=OK  TSP=OPTIMAL
STATUS=OPTIMAL  OBJECTIVE=10627.754318  RELATIVE_GAP=0  ABSOLUTE_GAP=0
PRIMAL_INFEASIBILITY=0  BOUND_INFEASIBILITY=0  INTEGER_INFEASIBILITY=0
BEST_BOUND=10627.754318  NODES=3  ITERATIONS=169  CPU_TIME=0.30  REAL_TIME=0.35
```

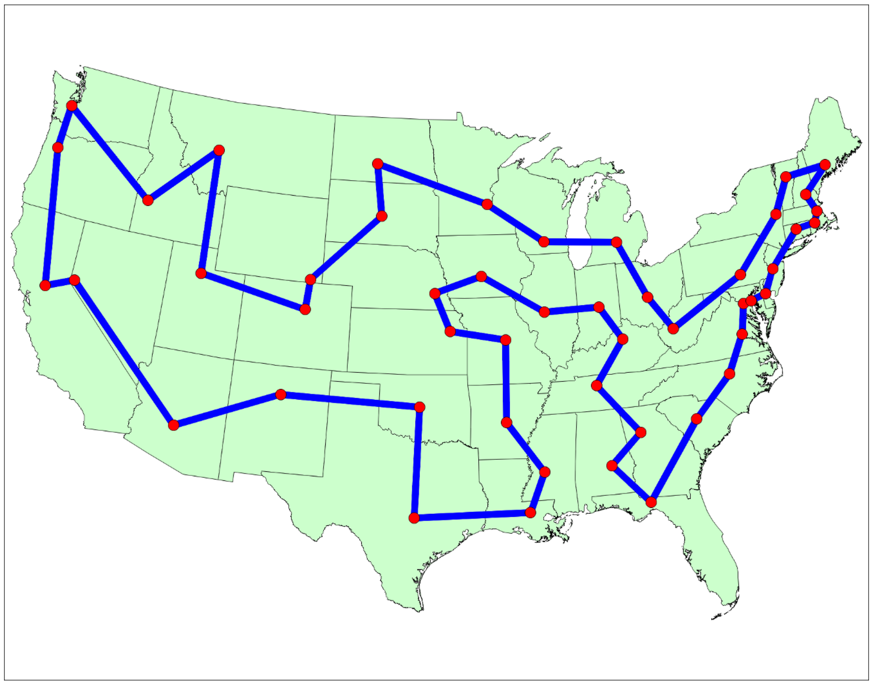
The following PROC GPROJECT and PROC GMAP statements produce a graphical display of the solution:

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

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

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

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

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

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

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

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

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

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

Output 2.7.2 Optimal Traveling Salesman Tour through US Capital Cities



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

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

The data set TSPTourLinksDirected is shown in Figure 2.70.

Figure 2.70 Links in the Optimal Traveling Salesman Tour

```
City Name             City Name                distance

Montgomery            Tallahassee                177.14
Tallahassee           Columbia                   311.23
Columbia              Raleigh                    182.99
Raleigh               Richmond                   135.58
Richmond              Washington                  97.96
Washington            Annapolis                   27.89
Annapolis             Dover                       54.01
Dover                 Trenton                     83.88
Trenton               Hartford                   151.65
Hartford              Providence                  65.56
Providence            Boston                      38.41
Boston                Concord                     66.30
Concord               Augusta                    117.36
Augusta               Montpelier                 139.32
Montpelier            Albany                     126.19
Albany                Harrisburg                 230.24
Harrisburg            Charleston                 287.34
Charleston            Columbus                   134.64
Columbus              Lansing                    205.08
Lansing               Madison                    246.88
Madison               Saint Paul                 226.25
Saint Paul            Bismarck                   391.25
Bismarck              Pierre                     170.27
Pierre                Cheyenne                   317.90
Cheyenne              Denver                      98.33
Denver                Salt Lake City             373.05
Salt Lake City        Helena                     403.40
Helena                Boise City                 291.20
Boise City            Olympia                    401.31
Olympia               Salem                      146.00
Salem                 Sacramento                 447.40
Sacramento            Carson City                101.51
Carson City           Phoenix                    577.84
Phoenix               Santa Fe                   378.27
Santa Fe              Oklahoma City              474.92
Oklahoma City         Austin                     357.38
Austin                Baton Rouge                394.78
Baton Rouge           Jackson                    139.75
Jackson               Little Rock                206.87
Little Rock           Jefferson City             264.75
Jefferson City        Topeka                     191.67
Topeka                Lincoln                    132.94
Lincoln               Des Moines                 168.10
Des Moines            Springfield                243.02
Springfield           Indianapolis               186.46
Indianapolis          Frankfort                  129.90
Frankfort             Nashville-Davidson         175.58
Nashville-Davidson    Atlanta                    212.61
Atlanta               Montgomery                 145.39
                                             ==========
                                              10,627.75
```

References

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

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

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," `http://maps.google.com`, accessed March 16, 2011.

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

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

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

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

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

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, `http://www.cnn.com/2009/HEALTH/12/14/kidney.transplant/index.html`, accessed March 16, 2011.

Index

Your Turn