SAS® OPTGRAPH
Procedure 14.3
Graph Algorithms and Network Analysis
# Contents

Chapter 1.  The OPTGRAPH Procedure ........................................ 1

Index .................................................................................. 239
Credits

Documentation

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

Software

PROC OPTGRAPH Matthew Galati, Yi Liao

Support Groups

Software Testing Mustafa Kabul, Charles B. Kelly, Minghui Liu, Rob Pratt
Technical Support Tonya Chapman
Chapter 1
The OPTGRAPH Procedure

Contents

Overview: OPTGRAPH Procedure .................................................. 3
Getting Started: OPTGRAPH Procedure ....................................... 4
  Road Network Shortest Path .................................................. 4
  Authority in U.S. Supreme Court Precedence ............................ 7
Syntax: OPTGRAPH Procedure .................................................... 10
  Functional Summary ............................................................ 10
  PROC OPTGRAPH Statement ................................................. 17
  BICONCOMP Statement ....................................................... 20
  CENTRALITY Statement ....................................................... 20
  CLIQUE Statement ............................................................. 25
  COMMUNITY Statement ....................................................... 26
  CONCOMP Statement .......................................................... 29
  CORE Statement ............................................................... 30
  CYCLE Statement .............................................................. 30
  DATA_LINKS_VAR Statement ............................................... 32
  DATA_MATRIX_VAR Statement ............................................. 33
  DATA_NODES_VAR Statement .............................................. 33
  EIGENVECTOR Statement ..................................................... 34
  LINEAR_ASSIGNMENT Statement ......................................... 35
  MINCOSTFLOW Statement ..................................................... 36
  MINCUT Statement ............................................................ 37
  MINSPANTREE Statement ..................................................... 38
  PERFORMANCE Statement ................................................... 38
  REACH Statement ............................................................. 39
  SHORTPATH Statement ........................................................ 41
  SUMMARY Statement .......................................................... 42
  TRANSITIVE_CLOSURE Statement ......................................... 44
  TSP Statement ................................................................. 45
Details: OPTGRAPH Procedure .................................................. 50
  Graph Input Data ............................................................. 50
  Matrix Input Data ............................................................. 58
  Data Input Order ............................................................. 59
  Parallel Processing ............................................................ 60
  Numeric Limitations ......................................................... 60
  Size Limitations .............................................................. 62
  Common Notation and Assumptions ....................................... 62
Chapter 1: The OPTGRAPH Procedure

Biconnected Components and Articulation Points .......................... 62
Centrality .................................................................................. 66
Clique ....................................................................................... 87
Community Detection ................................................................. 90
Connected Components ................................................................. 99
Core Decomposition ..................................................................... 104
Cycle .......................................................................................... 108
Eigenvector Problem ................................................................... 114
Linear Assignment (Matching) ..................................................... 116
Minimum-Cost Network Flow ..................................................... 117
Minimum Cut ............................................................................... 125
Minimum Spanning Tree ............................................................. 129
Reach (Ego) Network ................................................................... 131
Shortest Path ............................................................................... 143
Summary ..................................................................................... 154
Transitive Closure ....................................................................... 163
Traveling Salesman Problem ....................................................... 165
Macro Variables .......................................................................... 173
ODS Table Names ........................................................................ 185

Examples: OPTGRAPH Procedure .............................................. 188
Example 1.1: Articulation Points in a Terrorist Network .............. 188
Example 1.2: Influence Centrality for Project Groups in a Research Department .................................................. 190
Example 1.3: Betweenness and Closeness Centrality for Computer Network Topology ............................................. 194
Example 1.4: Betweenness and Closeness Centrality for Project Groups in a Research Department ............................................. 197
Example 1.5: Eigenvector Centrality for Word Sense Disambiguation ................................................................. 200
Example 1.6: Centrality Metrics for Project Groups in a Research Department .................................................. 203
Example 1.7: Community Detection on Zachary’s Karate Club Data .................................................. 206
Example 1.8: Recursive Community Detection on Zachary’s Karate Club Data .................................................. 210
Example 1.9: Cycle Detection for Kidney Donor Exchange .......... 212
Example 1.10: Linear Assignment Problem for Minimizing Swim Times .................................................. 217
Example 1.11: Linear Assignment Problem, Sparse Format versus Dense Format .................................................. 220
Example 1.12: Minimum Spanning Tree for Computer Network Topology .................................................. 223
Example 1.13: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System .................................................. 224
Example 1.14: Reach Networks for Computation of Market Coverage of a Terrorist Network .................................................. 227
Example 1.15: Traveling Salesman Tour through US Capital Cities .................................................. 229

References .................................................................................. 235
Overview: OPTGRAPH Procedure

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

<table>
<thead>
<tr>
<th>Algorithm Class</th>
<th>PROC OPTGRAPH Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biconnected components</td>
<td>BICONCOMP</td>
</tr>
<tr>
<td>Centrality metrics</td>
<td>CENTRALITY</td>
</tr>
<tr>
<td>Maximal cliques</td>
<td>CLIQUE</td>
</tr>
<tr>
<td>Community detection</td>
<td>COMMUNITY</td>
</tr>
<tr>
<td>Connected components</td>
<td>CONCOMP</td>
</tr>
<tr>
<td>Core decomposition</td>
<td>CORE</td>
</tr>
<tr>
<td>Cycle detection</td>
<td>CYCLE</td>
</tr>
<tr>
<td>Eigenvector problem</td>
<td>EIGENVECTOR</td>
</tr>
<tr>
<td>Weighted matching</td>
<td>LINEAR_ASSIGNMENT</td>
</tr>
<tr>
<td>Minimum-cost network flow</td>
<td>MINCOSTFLOW</td>
</tr>
<tr>
<td>Minimum cut</td>
<td>MINCUT</td>
</tr>
<tr>
<td>Minimum spanning tree</td>
<td>MNSPANTREE</td>
</tr>
<tr>
<td>Reach networks</td>
<td>REACH</td>
</tr>
<tr>
<td>Shortest path</td>
<td>SHORTESTPATH</td>
</tr>
<tr>
<td>Graph summary</td>
<td>SUMMARY</td>
</tr>
<tr>
<td>Transitive closure</td>
<td>TRANSITIVE_CLOSURE</td>
</tr>
<tr>
<td>Traveling salesman</td>
<td>TSP</td>
</tr>
</tbody>
</table>

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

You can check the SAS log for the version number being used in any invocation of PROC OPTGRAPH. The following statements check the version:

```sas
proc optgraph;
run;
```

Then the log displays the version number as shown in Figure 1.1.
Getting Started: OPTGRAPH Procedure

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

This section shows two introductory examples for getting started with the OPTGRAPH procedure. For more detail about the input formats expected and the various algorithms available, see the sections “Details: OPTGRAPH Procedure” on page 50 and “Examples: OPTGRAPH Procedure” on page 188.

Road Network Shortest Path

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

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

```sas
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
; run;
```

```sas
data LinkSetInRoadNC10am;
  set LinkSetInRoadNC10am;
  time_to_travel = miles * 1/miles_per_hour * 60;
run;
```
Using PROC OPTGRAPH, you want to find the route that yields the shortest path between home (614 Capital Blvd) and the SAS headquarters (SAS Campus Drive). This can be done with the SHORTPATH statement as follows:

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

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

**Figure 1.2** Shortest Path for Road Network at 10:00 A.M.

<table>
<thead>
<tr>
<th>order</th>
<th>start_inter</th>
<th>end_inter</th>
<th>time_to_travel</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>614CapitalBlvd</td>
<td>Capital/WadeAve</td>
<td>1.4400</td>
</tr>
<tr>
<td>2</td>
<td>Capital/WadeAve</td>
<td>WadeAve/RaleighExpy</td>
<td>4.5000</td>
</tr>
<tr>
<td>3</td>
<td>WadeAve/RaleighExpy</td>
<td>RaleighExpy/US40W</td>
<td>3.0000</td>
</tr>
<tr>
<td>4</td>
<td>RaleighExpy/US40W</td>
<td>US40W/HarrisonAve</td>
<td>1.4182</td>
</tr>
<tr>
<td>5</td>
<td>US40W/HarrisonAve</td>
<td>SASCampusDrive</td>
<td>1.2000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>11.5582</strong></td>
</tr>
</tbody>
</table>

**Figure 1.3** Shortest Path for Road Network at 10:00 A.M. in Google Maps
Now suppose that it is rush hour (5:00 p.m.) and the time to traverse the roads has changed because of traffic patterns. You want to find the route that is the shortest path for going home from SAS headquarters under different speed assumptions due to traffic. The following data set lists approximate travel times and speeds for driving in the opposite direction:

```sas
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
;
```

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

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

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

Now, the output data set ShortPath, shown in Figure 1.4, shows the best route for going home. Because the traffic on Wade Avenue is usually heavy at this time of day, the route home is different from the route to work.
Authority in U.S. Supreme Court Precedence

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

Given such a citation network, you can then use a metric known as authority score to rank the importance of these cases. This metric is explained in more detail in the section “Hub and Authority Scoring” on page 77. Figure 1.6 shows a small representative subset of the citation network for landmark abortion decisions from the example in Fowler and Joen 2008.
The data set Cases stores a mapping between case name and the case identifier:

``` SAS
data Cases;
  length case_id 8 case_name $80;
  input case_id 1-5 case_name $ 7-80;
  datalines;
12061 Jacobson v. Massachusetts, 197 U.S. 11 (1905)
25347 Roe vs. Wade, 410 U.S. 113 (1973)
29003 Webster vs. Repro-Health Services, 492 U.S. 490 (1989)
29153 Cruzan v. Director, MO Dept of Health, 497 U.S. 261 (1990)
29155 Georgia v. South Carolina, 497 U.S. 376 (1990)
29459 Planned Parenthood of SE PA vs. Casey, 505 U.S. 833 (1992)
;```

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

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

**Figure 1.6** Citation Network for Some U.S. Supreme Court Cases
You can calculate the authority scores of each case by using the CENTRALITY statement with the AUTH= option, as follows:

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

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

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

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

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

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

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

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

PROC OPTGRAPH options ;

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

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

Performance Statement:
PERFORMANCE < options > ;

PROC OPTGRAPH statements are divided into four main categories: the PROC statement, the data input statements, the algorithm statements, and the PERFORMANCE statement. The PROC statement invokes the procedure and sets option values that are used across multiple algorithms. The data input statements control the names of the variables that PROC OPTGRAPH expects in the data input. The algorithm statements determine which algorithms are run and set options for each individual algorithm. The PERFORMANCE statement specifies performance options for multithreaded computing.

The section “Functional Summary” on page 10 provides a quick reference for each of the options for each statement. Each statement is then described in more detail in its own section; the PROC OPTGRAPH statement is described first, and sections that describe all other statements are presented in alphabetical order.

Functional Summary

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

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PROC OPTGRAPH Options</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Input</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the link data set</td>
<td>DATA_LINKS=</td>
</tr>
<tr>
<td>Specifies the matrix data set</td>
<td>DATA_MAT matrix=</td>
</tr>
<tr>
<td>Specifies the node data set</td>
<td>DATA_NODES=</td>
</tr>
<tr>
<td>Specifies the node subset data set</td>
<td>DATA_NODES_SUB=</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the link output data set</td>
<td>OUT_LINKS=</td>
</tr>
<tr>
<td>Specifies the node output data set</td>
<td>OUT_NODES=</td>
</tr>
<tr>
<td><strong>Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the subgraph filter level</td>
<td>FILTER_SUBGRAPH=</td>
</tr>
<tr>
<td>Specifies the graph direction</td>
<td>GRAPH_DIRECTION=</td>
</tr>
<tr>
<td>Specifies the internal graph format</td>
<td>GRAPH_INTERNAL_FORMAT=</td>
</tr>
<tr>
<td>Includes self links</td>
<td>INCLUDE_SELFLINK</td>
</tr>
<tr>
<td>Specifies the overall log level</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies whether time units are in CPU time or real time</td>
<td>TIMETYPE=</td>
</tr>
<tr>
<td><strong>Data Input Statements</strong></td>
<td></td>
</tr>
<tr>
<td><strong>DATA_LINKS_VAR Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the data set variable name for the <em>from</em> nodes</td>
<td>FROM=</td>
</tr>
<tr>
<td>Specifies the data set variable name for the link flow lower bounds</td>
<td>LOWER=</td>
</tr>
<tr>
<td>Specifies the data set variable name for the <em>to</em> nodes</td>
<td>TO=</td>
</tr>
<tr>
<td>Specifies the data set variable name for the link flow upper bounds</td>
<td>UPPER=</td>
</tr>
<tr>
<td>Specifies the data set variable name for the link weights</td>
<td>WEIGHT=</td>
</tr>
<tr>
<td><strong>DATA_MATRIX_VAR</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the data set variable names for the matrix</td>
<td></td>
</tr>
<tr>
<td><strong>DATA_NODES_VAR Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the data set variable name for cluster identifiers</td>
<td>CLUSTER=</td>
</tr>
<tr>
<td>Specifies the data set variable name for the nodes</td>
<td>NODE=</td>
</tr>
<tr>
<td>Specifies the data set variable name for node weights</td>
<td>WEIGHT=</td>
</tr>
<tr>
<td>Specifies the data set variable name for auxiliary node weights</td>
<td>WEIGHT2=</td>
</tr>
<tr>
<td><strong>Algorithm Statements</strong></td>
<td></td>
</tr>
<tr>
<td><strong>BICONCOMP Option</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the log level for biconnected components</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td><strong>CENTRALITY Options</strong></td>
<td></td>
</tr>
<tr>
<td>Calculates authority centrality and specifies the type to process</td>
<td>AUTH=</td>
</tr>
<tr>
<td>Calculates betweenness centrality and specifies the type to process</td>
<td>BETWEEN=</td>
</tr>
<tr>
<td>Specifies whether to normalize the betweenness calculation</td>
<td>BETWEEN_NORM=</td>
</tr>
<tr>
<td>Decomposes the calculations for centrality by cluster (or subgraph)</td>
<td>BY_CLUSTER</td>
</tr>
<tr>
<td>Calculates closeness centrality and specifies the type to process</td>
<td>CLOSE=</td>
</tr>
<tr>
<td>Specifies the accounting method for no paths in closeness</td>
<td>CLOSE_NOPATH=</td>
</tr>
<tr>
<td>Calculates the node clustering coefficients</td>
<td>CLUSTERING_COEF</td>
</tr>
</tbody>
</table>
### Table 1.2  (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calculates degree centrality and specifies the type to process</td>
<td>DEGREE=</td>
</tr>
<tr>
<td>Calculates eigenvector centrality and specifies the type to process</td>
<td>EIGEN=</td>
</tr>
<tr>
<td>Specifies the algorithm to use for eigenvector calculation</td>
<td>EIGEN_ALGORITHM=</td>
</tr>
<tr>
<td>Specifies the maximum number of iterations for eigenvector calculation</td>
<td>EIGEN_MAXITER=</td>
</tr>
<tr>
<td>Calculates hub centrality and specifies the type to process</td>
<td>HUB=</td>
</tr>
<tr>
<td>Calculates influence centrality and specifies the type to process</td>
<td>INFLUENCE=</td>
</tr>
<tr>
<td>Specifies the iteration log frequency (nodes)</td>
<td>LOGFREQNODE=</td>
</tr>
<tr>
<td>Specifies the iteration log frequency (seconds)</td>
<td>LOGFREQTIME=</td>
</tr>
<tr>
<td>Specifies the log level for centrality</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the subgraph node size to run separately</td>
<td>SUBSIZESWITCH=</td>
</tr>
<tr>
<td>Specifies the data set variable to use for weight2 in centrality</td>
<td>WEIGHT2=</td>
</tr>
</tbody>
</table>

#### CLIQUE Options

| Specifies the log level for clique calculations | LOGLEVEL= |
| Specifies the maximum number of cliques to return during clique calculations | MAXCLIQUES= |
| Specifies the maximum amount of time to spend calculating cliques | MAXTIME= |
| Specifies the output data set for cliques | OUT= |

#### COMMUNITY Options

| Specifies the community detection algorithm | ALGORITHM= |
| Specifies the percentage of small-weight links to be removed | LINK_REMOVAL_RATIO= |
| Specifies the log level for community detection | LOGLEVEL= |
| Specifies the maximum number of iterations for community detection | MAXITER= |
| Specifies the output data set for inter-community links | OUT_COMMLINKS= |
| Specifies the output data set for community summary table | OUT_COMMUNITY= |
| Specifies the output data set for community level summary table | OUT_LEVEL= |
| Specifies the output data set for community overlap table | OUT_OVERLAP= |
| Specifies the random factor in the parallel label propagation algorithm | RANDOM_FACTOR= |
| Specifies the random seed for the parallel label propagation algorithm | RANDOM_SEED= |
| Applies the recursive option to break large communities | RECURSIVE |
| Specifies the resolution list for community detection | RESOLUTION_LIST= |
| Specifies the modularity tolerance value for community detection | TOLERANCE= |

#### CONCOMP Options

| Specifies the algorithm to use for connected components | ALGORITHM= |
| Specifies the log level for connected components | LOGLEVEL= |

#### CORE Options

| Specifies the type of core to process | LINKS= |
| Specifies the log level for the core algorithm | LOGLEVEL= |
| Specifies the maximum amount of time to spend in the core algorithm | MAXTIME= |

#### CYCLE Options

| Specifies the log level for the cycle algorithm | LOGLEVEL= |
| Specifies the maximum number of cycles to return during cycle calculations | MAXCYCLES= |
Table 1.2  (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies the maximum length for the cycles found</td>
<td>MAXLENGTH=</td>
</tr>
<tr>
<td>Specifies the maximum link weight for the cycles found</td>
<td>MAXLINKWEIGHT=</td>
</tr>
<tr>
<td>Specifies the maximum node weight for the cycles found</td>
<td>MAXNODEWEIGHT=</td>
</tr>
<tr>
<td>Specifies the maximum amount of time to spend calculating cycles</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Specifies the minimum length for the cycles found</td>
<td>MINLENGTH=</td>
</tr>
<tr>
<td>Specifies the minimum link weight for the cycles found</td>
<td>MINLINKWEIGHT=</td>
</tr>
<tr>
<td>Specifies the minimum node weight for the cycles found</td>
<td>MINNODEWEIGHT=</td>
</tr>
<tr>
<td>Specifies the mode for the cycle calculations</td>
<td>MODE=</td>
</tr>
<tr>
<td>Specifies the output data set for cycles</td>
<td>OUT=</td>
</tr>
</tbody>
</table>

**EIGENVECTOR Options**

- Specifies the algebraic type of eigenvalues to calculate: EIGENVALUES=
- Specifies the log level for eigenvector calculations: LOGLEVEL=
- Specifies the maximum number of iterations for eigenvector calculation: MAXITER=
- Specifies the number of eigenvectors to calculate: NEIGEN=
- Specifies the output data set for eigenvectors: OUT=

**LINEAR_ASSIGNMENT Options**

- Specifies the data set variable names for the linear assignment identifiers: ID=( )
- Specifies the log level for the linear assignment algorithm: LOGLEVEL=
- Specifies the output data set for linear assignment: OUT=
- Specifies the data set variable names for costs (or weights): WEIGHT=( )

**MINCOSTFLOW Options**

- Specifies the iteration log frequency: LOGFREQ=
- Specifies the log level for the minimum-cost network flow algorithm: LOGLEVEL=
- Specifies the maximum amount of time to spend calculating the optimal flow: MAXTIME=

**MINCUT Options**

- Specifies the log level for the minimum-cut algorithm: LOGLEVEL=
- Specifies the maximum number of cuts to return: MAXNUMCUTS=
- Specifies the maximum weight of the cuts to return: MAXWEIGHT=
- Specifies the output data set for minimum cut: OUT=

**MINSPAN_TREE Options**

- Specifies the log level for the minimum spanning tree algorithm: LOGLEVEL=
- Specifies the output data set for minimum spanning tree: OUT=

**REACH Options**

- Decomposes the calculations for reach by cluster (or subgraph): BY_CLUSTER
- Calculates the directed reach counts: DIGRAPH
- Treats each node as a source in reach calculations: EACH_SOURCE
- Ignores the source node in reach counts: IGNORE_SELF
- Specifies the maximum number of links to allow in the reach calculations: MAXREACH=
- Specifies the iteration log frequency (seconds): LOGFREQTIME=
Table 1.2 (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies the log level for reach calculations</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the output data set for reach counts</td>
<td>OUT_COUNTS=</td>
</tr>
<tr>
<td>Specifies the output data set for reach counts (limit=1)</td>
<td>OUT_COUNTS1=</td>
</tr>
<tr>
<td>Specifies the output data set for reach counts (limit=2)</td>
<td>OUT_COUNTS2=</td>
</tr>
<tr>
<td>Specifies the output data set for reach links</td>
<td>OUT_LINKS=</td>
</tr>
<tr>
<td>Specifies the output data set for reach nodes</td>
<td>OUT_NODES=</td>
</tr>
<tr>
<td><strong>SHORTPATH Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the iteration log frequency (nodes)</td>
<td>LOGFREQ=</td>
</tr>
<tr>
<td>Specifies the log level for shortest paths</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the output data set for shortest paths</td>
<td>OUT_PATHS=</td>
</tr>
<tr>
<td>Specifies the output data set for shortest path summaries</td>
<td>OUT_WEIGHTS=</td>
</tr>
<tr>
<td>Specifies the type of output for shortest paths results</td>
<td>PATHS=</td>
</tr>
<tr>
<td>Specifies the sink node for shortest paths calculations</td>
<td>SINK=</td>
</tr>
<tr>
<td>Specifies the source node for shortest paths calculations</td>
<td>SOURCE=</td>
</tr>
<tr>
<td>Specifies whether to use weights in calculating shortest paths</td>
<td>USEWEIGHT=</td>
</tr>
<tr>
<td>Specifies the data set variable name for the auxiliary link weights</td>
<td>WEIGHT2=</td>
</tr>
<tr>
<td><strong>SUMMARY Options</strong></td>
<td></td>
</tr>
<tr>
<td>Calculates information about biconnected components</td>
<td>BICONCOMP</td>
</tr>
<tr>
<td>Decomposes the calculations for summary by cluster (or subgraph)</td>
<td>BY_CLUSTER</td>
</tr>
<tr>
<td>Calculates information about connected components</td>
<td>CONCOMP</td>
</tr>
<tr>
<td>Calculates the approximate diameter and chooses the weight type</td>
<td>DIAMETER_APPROX=</td>
</tr>
<tr>
<td>Specifies the iteration log frequency (nodes)</td>
<td>LOGFREQNODE=</td>
</tr>
<tr>
<td>Specifies the iteration log frequency (seconds)</td>
<td>LOGFREQTIME=</td>
</tr>
<tr>
<td>Specifies the log level for summary calculations</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the output data set for summary results</td>
<td>OUT=</td>
</tr>
<tr>
<td>Calculates information about shortest paths and chooses the weight type</td>
<td>SHORTPATH=</td>
</tr>
<tr>
<td>Specifies the subgraph node size to run separately</td>
<td>SUBSIZESWITCH=</td>
</tr>
<tr>
<td><strong>TRANSITIVE_CLOSURE Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the log level for transitive closure</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the output data set for transitive closure results</td>
<td>OUT=</td>
</tr>
<tr>
<td><strong>TSP Options</strong></td>
<td></td>
</tr>
<tr>
<td>Specifies the stopping criterion based on the absolute objective gap</td>
<td>ABSOBJGAP=</td>
</tr>
<tr>
<td>Specifies the level of conflict search</td>
<td>CONFLICTSEARCH=</td>
</tr>
<tr>
<td>Specifies the cutoff value for branch-and-bound node removal</td>
<td>CUTOFF=</td>
</tr>
<tr>
<td>Specifies the overall cut strategy level</td>
<td>CUTSTRATEGY=</td>
</tr>
<tr>
<td>Emphasizes feasibility or optimality</td>
<td>EMPHASIS=</td>
</tr>
<tr>
<td>Specifies the initial and primal heuristics level</td>
<td>HEURISTICS=</td>
</tr>
<tr>
<td>Specifies the frequency of printing the branch-and-bound node log</td>
<td>LOGFREQ=</td>
</tr>
<tr>
<td>Specifies the log level for the traveling salesman algorithm</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the maximum number of branch-and-bound nodes to be processed</td>
<td>MAXNODES=</td>
</tr>
<tr>
<td>Specifies the maximum number of solutions to be found</td>
<td>MAXSOLS=</td>
</tr>
</tbody>
</table>
Table 1.2  (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies the maximum amount of time to spend in the algorithm</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Specifies whether to use a mixed integer linear programming solver</td>
<td>MILP=</td>
</tr>
<tr>
<td>Specifies the branch-and-bound node selection strategy</td>
<td>NODESEL=</td>
</tr>
<tr>
<td>Specifies the output data set for traveling salesman problem</td>
<td>OUT=</td>
</tr>
<tr>
<td>Specifies the probing level</td>
<td>PROBE=</td>
</tr>
<tr>
<td>Specifies the stopping criterion based on the relative objective gap</td>
<td>RELOBJGAP=</td>
</tr>
<tr>
<td>Specifies the number of simplex iterations to be performed on each variable</td>
<td>STRONGITER=</td>
</tr>
<tr>
<td>Specifies the number of candidates for the strong branching strategy</td>
<td>STRONGLEN=</td>
</tr>
<tr>
<td>Specifies the stopping criterion based on the target objective value</td>
<td>TARGET=</td>
</tr>
<tr>
<td>Specifies the rule for selecting branching variable</td>
<td>VARSEL=</td>
</tr>
</tbody>
</table>

For more information about the options available for the PERFORMANCE statement, see the section “PERFORMANCE Statement” on page 38.

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

Table 1.3  Supported Input Formats and Graph Types by Statement

<table>
<thead>
<tr>
<th>Statement</th>
<th>Input Format</th>
<th>DIRECTION</th>
<th>INTERNAL_FORMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Graph</td>
<td>Matrix</td>
<td>UNDIRECTED</td>
</tr>
<tr>
<td>BICONCOMP</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CENTRALITY</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AUTH=, HUB=</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>EIGEN=</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>BETWEEN=, CLOSE=,</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CLUSTERING_COEF,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEGREE=,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INFLUENCE=,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CENTRALITY / BY_CLUSTER</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>AUTH=, HUB=</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>EIGEN=</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>BETWEEN=, CLOSE=,</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CLUSTERING_COEF,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEGREE=,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INFLUENCE=,</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLIQUE</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>COMMUNITY</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>ALGORITHM=</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>LOUVAIN, LABEL_PROP</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>PARALLEL_LABEL_PROP</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>
Table 1.3 (continued)

<table>
<thead>
<tr>
<th>Statement</th>
<th>Input Format</th>
<th>DIRECTION</th>
<th>INTERNAL_FORMAT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>UNDIRECTED</td>
<td>DIRECTED</td>
</tr>
<tr>
<td></td>
<td></td>
<td>THIN</td>
<td>FULL</td>
</tr>
<tr>
<td>CONCOMP</td>
<td>Graph</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ALGORITHM=DFS</td>
<td>Matrix</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>UNION_FIND</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CORE</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CYCLE</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>EIGENVECTOR</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>LINEAR_ASSIGNMENT</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MINCOSTFLOW</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MINCUT</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MINSPAN TREE</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>REACH</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>REACH / BY_CLUSTER</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SHORTEST</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SUMMARY</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SUMMARY / BY_CLUSTER</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>TRANSITIVE_CLOSURE</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>TSP</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

Table 1.4 indicates for each algorithm statement in the OPTGRAPH procedure which output data set options you can specify and whether the algorithm populates the data sets specified in the OUT_NODES= and OUT_LINKS= options in the PROC OPTGRAPH statement.

Table 1.4 Output Options by Statement

<table>
<thead>
<tr>
<th>Statement</th>
<th>OUT_NODES</th>
<th>OUT_LINKS</th>
<th>Algorithm Statement Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>BICONCOMP</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CENTRALITY</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>AUTH=, CLOSE=,</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CLUSTERING_COEF,</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DEGREE=, EIGEN=,</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>HUB=, INFLUENCE=</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>BETWEEN=</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CLIQUE</td>
<td>OUT=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMUNITY</td>
<td>OUT_COMM_LINKS=,</td>
<td>OUT_COMMUNITY=,</td>
<td>OUT_LEVEL=,</td>
</tr>
<tr>
<td>ALGORITHM=LOUVAIN,</td>
<td>X</td>
<td>OUT_COMM_LINKS=,</td>
<td>OUT_COMMUNITY=,</td>
</tr>
<tr>
<td>LABEL_PROP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARALLEL_LABEL_PROP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONCOMP</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CORE</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
### Table 1.4 (continued)

<table>
<thead>
<tr>
<th>Statement</th>
<th>OUT_NODES</th>
<th>OUT_LINKS</th>
<th>Algorithm Statement Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>CYCLE</td>
<td></td>
<td></td>
<td>OUT=</td>
</tr>
<tr>
<td>EIGENVECTOR</td>
<td></td>
<td></td>
<td>OUT=</td>
</tr>
<tr>
<td>LINEAR_ASSIGNMENT</td>
<td></td>
<td></td>
<td>OUT=</td>
</tr>
<tr>
<td>MINCOSTFLOW</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINCUT</td>
<td>X</td>
<td></td>
<td>OUT=</td>
</tr>
<tr>
<td>MINSPANTREE</td>
<td></td>
<td></td>
<td>OUT=</td>
</tr>
<tr>
<td>REACH</td>
<td></td>
<td></td>
<td>OUT_COUNTS=, OUT_LINKS=, OUT_NODES= OUT_COUNTS1=,</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>OUT_NODES=</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHORTPATH</td>
<td>X</td>
<td></td>
<td>OUT=</td>
</tr>
<tr>
<td>SUMMARY</td>
<td></td>
<td></td>
<td>OUT=</td>
</tr>
<tr>
<td>TRANSITIVE_CLOSURE</td>
<td></td>
<td></td>
<td>OUT=</td>
</tr>
<tr>
<td>TSP</td>
<td>X</td>
<td></td>
<td>OUT=</td>
</tr>
</tbody>
</table>

### PROC OPTGRAPH Statement

**PROC OPTGRAPH < options > ;**

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

**DATA_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 50 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 58 for more information.

**DATA NODES=SAS-data-set**

**NODES=SAS-data-set**

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

See the section “Node Input Data” on page 54 for more information.
DATA_NODES_SUB=SAS-data-set

NODES_SUB=SAS-data-set

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

See the section “Node Subset Input Data” on page 55 for more information.

FILTER_SUBGRAPH=number

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

See the section “Graph Input Data” on page 50 for more information.

GRAPH_DIRECTION=DIRECTED | UNDIRECTED

DIRECTION=DIRECTED | UNDIRECTED

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

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIRECTED</td>
<td>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 \to j)). The node (i) is called the source (tail) node, and (j) is called the sink (head) node.</td>
</tr>
<tr>
<td>UNDIRECTED</td>
<td>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.</td>
</tr>
</tbody>
</table>

By default, GRAPH_DIRECTION=UNDIRECTED. See the section “Graph Input Data” on page 50 for more information.

GRAPH_INTERNAL_FORMAT=FULL | THIN

INTERNAL_FORMAT=FULL | THIN

requests the internal graph format for the algorithms to use.

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FULL</td>
<td>Stores the graph in standard (full) format. This is the default.</td>
</tr>
<tr>
<td>THIN</td>
<td>Stores the graph in thin format. This option can improve performance in some cases both by reducing memory and by simplifying the construction of the internal data structures. The thin format causes PROC OPTGRAPH to skip the removal of duplicate links when it reads in the graph. So this option should be used with caution. For some algorithms, the thin format is not allowed and this option is ignored. Setting GRAPH_INTERNAL_FORMAT=THIN can often be helpful when you do calculations that are decomposed by subgraph.</td>
</tr>
</tbody>
</table>
See the section “Graph Input Data” on page 50 for more information.

\textbf{INCLUDE\_SELFLINK}

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

\textbf{LOGLEVEL=number | string}

controls the amount of information that is displayed in the SAS log. Each algorithm has its own specific log level. This setting sets the log level for all algorithms except those for which you specify the LOGLEVEL= option in the algorithm statement. Table 1.7 describes the valid values for this option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all procedure-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the input, output, and algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the input, output, and algorithmic processing</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the input, output, and algorithmic processing</td>
</tr>
</tbody>
</table>

By default, LOGLEVEL=BASIC.

\textbf{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.

\textbf{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.

\textbf{STANDARDIZED\_LABELS}

specifies that the input graph data is in a standardized format described in section “Standardized Labels” on page 56.

\textbf{TIMETYPE=number | string}

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

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CPU</td>
<td>Specifies units of CPU time</td>
</tr>
<tr>
<td>1</td>
<td>REAL</td>
<td>Specifies units of real time</td>
</tr>
</tbody>
</table>

By default, TIMETYPE=CPU.
BICONCOMP Statement

BICONCOMP < option > ;

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

See the section “Biconnected Components and Articulation Points” on page 62 for more information.

You can specify the following option in the BICONCOMP statement.

LOGLEVEL=number | string

controls the amount of information that is displayed in the SAS log. Table 1.9 describes the valid values for this option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all algorithm-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the algorithmic processing</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the algorithmic processing</td>
</tr>
</tbody>
</table>

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

CENTRALITY Statement

CENTRALITY < options > ;

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

The centrality metrics are described in the section “Centrality” on page 66.

You can specify the following options in the CENTRALITY statement.

AUTH=WEIGHT | UNWEIGHT | BOTH

specifies which type of authority centrality to calculate.

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEIGHT</td>
<td>Calculates authority centrality based on the weighted graph.</td>
</tr>
<tr>
<td>UNWEIGHT</td>
<td>Calculates authority centrality based on the unweighted graph.</td>
</tr>
<tr>
<td>BOTH</td>
<td>Calculates authority centrality based on both weighted and unweighted graphs.</td>
</tr>
</tbody>
</table>
If the input graph does not contain weights, then `WEIGHT` and `UNWEIGHT` both give the same results (using 1.0 for each link weight). This centrality metric can be used only for directed graphs. The authority centrality metric is described in the section “Hub and Authority Scoring” on page 77.

**BETWEEN=** `WEIGHT` | `UNWEIGHT` | `BOTH`

specifies which type of betweenness centrality to calculate.

**Table 1.11** Values for the BETWEEN= Option

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEIGHT</td>
<td>Calculates betweenness centrality based on the weighted graph.</td>
</tr>
<tr>
<td>UNWEIGHT</td>
<td>Calculates betweenness centrality based on the unweighted graph.</td>
</tr>
<tr>
<td>BOTH</td>
<td>Calculates betweenness centrality based on both weighted and unweighted graphs.</td>
</tr>
</tbody>
</table>

If the input graph does not contain weights, then `WEIGHT` and `UNWEIGHT` both give the same results (using 1.0 for each link weight). If the `OUT_NODES=` option is specified in the PROC OPTGRAPH statement, the node betweenness metric is produced. If the `OUT_LINKS=` option is specified, the link betweenness metric is produced. The betweenness centrality metric is described in the section “Betweenness Centrality” on page 73.

**BETWEEN_NORM=** `YES` | `NO`

specifies whether to normalize the betweenness centrality metrics.

**Table 1.12** Values for the BETWEEN_NORM= Option

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>Normalizes the betweenness metrics. This is the default.</td>
</tr>
<tr>
<td>NO</td>
<td>Does not normalize the betweenness metrics.</td>
</tr>
</tbody>
</table>

The normalization factor for betweenness centrality is described in the section “Betweenness Centrality” on page 73.

**BY_CLUSTER**

decomposes the calculations by cluster (or subgraph). If this option is specified, PROC OPTGRAPH looks for a definition of the clusters in the input data set specified by the `DATA_NODES=` option in the PROC OPTGRAPH statement. The use of the `BY_CLUSTER` option is described in the section “Processing by Cluster” on page 79.

**CLOSE=** `WEIGHT` | `UNWEIGHT` | `BOTH`

specifies which type of closeness centrality to calculate.

**Table 1.13** Values for the CLOSE= Option

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEIGHT</td>
<td>Calculates closeness centrality based on the weighted graph.</td>
</tr>
<tr>
<td>UNWEIGHT</td>
<td>Calculates closeness centrality based on the unweighted graph.</td>
</tr>
</tbody>
</table>
Table 1.13 (continued)

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOTH</td>
<td>Calculates closeness centrality based on both weighted and unweighted graphs.</td>
</tr>
</tbody>
</table>

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (using 1.0 for each link weight). The closeness centrality metric is described in the section “Closeness Centrality” on page 70.

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

Table 1.14 Values for the CLOSE_NOPATH= Option

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NNODES</td>
<td>Uses the number of nodes as a shortest path distance between disconnected nodes. This option cannot be used when calculating weighted closeness centrality.</td>
</tr>
<tr>
<td>DIAMETER</td>
<td>Uses the graph diameter (plus one) as a shortest path distance between disconnected nodes. This is the default.</td>
</tr>
<tr>
<td>ZERO</td>
<td>Uses zero as a shortest path distance between disconnected nodes.</td>
</tr>
<tr>
<td>HARMONIC</td>
<td>Uses the harmonic formula for calculating closeness centrality.</td>
</tr>
</tbody>
</table>

For each option, there is a slight variation in the formula for the closeness centrality metric. These differences are described in the section “Closeness Centrality” on page 70.

CLUSTERING_COEF
calculates the node clustering coefficient. The cluster coefficient is described in the section “Clustering Coefficient” on page 68.

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

Table 1.15 Values for the DEGREE= Option

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>Calculates degree based on in-links.</td>
</tr>
<tr>
<td>OUT</td>
<td>Calculates degree based on out-links.</td>
</tr>
<tr>
<td>BOTH</td>
<td>Calculates degree based on in-links and out-links.</td>
</tr>
</tbody>
</table>

For an undirected graph, the option values IN and BOTH are ignored, because there is only one notion of degree, which corresponds to the degree of out-links. The degree centrality metric is described in the section “Degree Centrality” on page 66.
EIGEN=WEIGHT | UNWEIGHT | BOTH

specifies which type of eigenvector centrality to calculate.

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEIGHT</td>
<td>Calculates eigenvector centrality based on the weighted graph.</td>
</tr>
<tr>
<td>UNWEIGHT</td>
<td>Calculates eigenvector centrality based on the unweighted graph.</td>
</tr>
<tr>
<td>BOTH</td>
<td>Calculates eigenvector centrality based on both weighted and unweighted graphs.</td>
</tr>
</tbody>
</table>

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (using 1.0 for each link weight). This centrality metric can be used only for undirected graphs. The eigenvector centrality metric is described in the section “Eigenvector Centrality” on page 75.

EIGEN_ALGORITHM=AUTOMATIC | JACOBI_DAVIDSON | POWER

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

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOMATIC</td>
<td>Requests that PROC OPTGRAPH automatically determine the eigensolver to use. This is the default.</td>
</tr>
<tr>
<td>JACOBI_DAVIDSON (JD)</td>
<td>Uses a variant of the Jacobi-Davidson algorithm for solving eigensystems (Sleijpen and van der Vorst 2000). This is used as the default for the eigenvector metric on undirected graphs and the hub and authority metrics.</td>
</tr>
<tr>
<td>POWER</td>
<td>Uses the power method to calculate eigenvectors. This is used as the default for the eigenvector metric on directed graphs.</td>
</tr>
</tbody>
</table>

EIGEN_MAXITER=number

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

HUB=WEIGHT | UNWEIGHT | BOTH

specifies which type of hub centrality to calculate.

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEIGHT</td>
<td>Calculates hub centrality based on the weighted graph.</td>
</tr>
<tr>
<td>UNWEIGHT</td>
<td>Calculates hub centrality based on the unweighted graph.</td>
</tr>
<tr>
<td>BOTH</td>
<td>Calculates hub centrality based on both weighted and unweighted graphs.</td>
</tr>
</tbody>
</table>
Chapter 1: The OPTGRAPH Procedure

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (using 1.0 for each link weight). This centrality metric can be used only for directed graphs. The hub centrality metric is described in the section “Hub and Authority Scoring” on page 77.

**INFLUENCE=**WEIGHT | UNWEIGHT | BOTH

specifies which type of influence centrality to calculate.

**Table 1.19** Values for the **INFLUENCE=** Option

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEIGHT</td>
<td>Calculates influence centrality based on the weighted graph.</td>
<td></td>
</tr>
<tr>
<td>UNWEIGHT</td>
<td>Calculates influence centrality based on the unweighted graph.</td>
<td></td>
</tr>
<tr>
<td>BOTH</td>
<td>Calculates influence centrality based on both weighted and unweighted graphs.</td>
<td></td>
</tr>
</tbody>
</table>

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (using 1.0 for each link weight). The influence centrality metric is described in the section “Influence Centrality” on page 67.

**LOGFREQNODE=**number

controls the frequency for displaying iteration logs for some of the centrality metrics. For computationally intensive algorithms such as betweenness and closeness centrality, this option displays progress every number nodes. If you also specify the BY_CLUSTERS option in this statement or a value greater than 1 for the NTHREADS= option in the PERFORMANCE statement, this option is ignored and the display frequency is determined by using the LOGFREQTIME= option instead. The value of number can be any integer greater than or equal to 1; the default is determined automatically based on the size of the graph. Setting this value too low can hurt performance on large-scale graphs.

**LOGFREQTIME=**number

controls the frequency for displaying iteration logs for some of the centrality metrics. For computationally intensive algorithms such as betweenness and closeness centrality, this option displays progress every number seconds. If you specify a value greater than 1 for the NTHREADS= option in the PERFORMANCE statement, PROC OPTGRAPH displays the number of nodes that have completed. If you specify the BY_CLUSTERS option, PROC OPTGRAPH displays the number of subgraphs that have completed. The value of number can be any integer greater than or equal to 1; the default is 5. Setting this value too low can hurt performance on large-scale graphs.

**LOGLEVEL=**number | string

controls the amount of information that is displayed in the SAS log. Table 1.20 describes the valid values for this option.

**Table 1.20** Values for **LOGLEVEL=** Option

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all algorithm-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the algorithmic processing including a progress log using the interval that is specified in the LOGFREQNODE= or LOGFREQTIME= option</td>
</tr>
</tbody>
</table>
The CLIQUE statement invokes an algorithm that finds maximal cliques on the input graph. Maximal cliques are described in the section “Clique” on page 87.

You can specify the following options in the CLIQUE statement:

**LOGLEVEL=number | string**

controls the amount of information that is displayed in the SAS log. Table 1.21 describes the valid values for this option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all algorithm-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the algorithmic processing</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the algorithmic processing</td>
</tr>
</tbody>
</table>

The default is the value that is specified in the LOGLEVEL= option in the PROC OPTGRAPH statement (or BASIC if that option is not specified).
MAXCLIQUES=number
specifies the maximum number of cliques to return during clique calculations. The default is the positive number that has the largest absolute value that can be represented in your operating environment.

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

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

COMMUNITY Statement

COMMUNITY < options > ;

The COMMUNITY statement invokes an algorithm that detects communities of the input graph. Community detection is described in the section “Community Detection” on page 90.

You can specify the following options in the COMMUNITY statement:

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

For more information about this option, see the sections “Community Detection” on page 90 and “Parallel Community Detection” on page 92.

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

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

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all algorithm-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the algorithmic processing</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the algorithmic processing</td>
</tr>
</tbody>
</table>

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

**MAXITER=number**

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

**OUT_COMM_LINKS=SAS-data-set**

specifies the output data set that describes the links between communities.

**OUT_COMMUNITY=SAS-data-set**

specifies the output data set that contains the number of nodes in each community.

**OUT_LEVEL=SAS-data-set**

specifies the output data set that contains community information at different resolution levels.

**OUT_OVERLAP=SAS-data-set**

specifies the output data set that describes the intensity of each node.

**RANDOM_FACTOR=number**

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

**RANDOM_SEED=number**

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

**RECURSIVE (options)**

requests that the algorithm recursively break down large communities into smaller ones until the specified conditions are satisfied. This option starts with the keyword RECURSIVE followed by any combination of three suboptions enclosed in parentheses—for example, RECURSIVE (MAX_COMM_SIZE=500) or RECURSIVE (MAX_COMM_SIZE=1000 MAX_DIAMETER=3 RELATION=AND).
Table 1.23  RECURSIVE options

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

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

RESOLUTION_LIST= num_list

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

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

For more information about the use of the RESOLUTION_LIST= option, see the section “Large Community” on page 93.

TOLERANCE= number

MODULARITY= number

specifies the tolerance value for when to stop iterations. When you specify ALGORITHM=LOUVAIN, the algorithm stops iterations when the percentage modularity gain between two consecutive iterations falls within the specified tolerance value. When you specify ALGORITHM=LABEL_PROP or ALGORITHM=PARALLEL_LABEL_PROP, the algorithm stops iterations when the percentage of label changes for all nodes in the graph falls within the tolerance specified by number. The valid range is strictly between 0 and 1. By default, TOLERANCE=0.01.
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 99.

You can specify the following options in the CONCOMP statement:

**ALGORITHM=DFS | UNION_FIND**

specifies the algorithm to use for calculating connected components.

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

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

**LOGLEVEL=number | string**

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

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all algorithm-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the algorithmic processing</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the algorithmic processing</td>
</tr>
</tbody>
</table>

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

CORE < options > ;

The CORE statement invokes an algorithm that finds the core decomposition of the input graph. Core decompositions are described in the section “Core Decomposition” on page 104.

You can specify the following options in the CORE statement:

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

Table 1.26  Values for the LINKS= Option

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IN</td>
<td>Calculates core based on in-links.</td>
<td></td>
</tr>
<tr>
<td>OUT</td>
<td>Calculates core based on out-links. This is the default.</td>
<td></td>
</tr>
<tr>
<td>BOTH</td>
<td>Calculates core based on in-links and out-links.</td>
<td></td>
</tr>
</tbody>
</table>

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

Table 1.27  Values for LOGLEVEL= Option

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all algorithm-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the algorithmic processing</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the algorithmic processing</td>
</tr>
</tbody>
</table>

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

MAXTIME=number
specifies the maximum amount of time to spend in the core decomposition algorithm. 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.

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 108.

You can specify the following options in the CYCLE statement:

**LOGLEVEL=** *number | string*

controls the amount of information that is displayed in the SAS log. Table 1.28 describes the valid values for this option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all algorithm-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the algorithmic processing</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the algorithmic processing</td>
</tr>
</tbody>
</table>

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

**MAXCYCLES=** *number*

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

**MAXLENGTH=** *number*

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

**MAXLINKWEIGHT=** *number*

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

**MAXNODEWEIGHT=** *number*

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

**MAXTIME=** *number*

specifies the maximum amount of time to spend finding cycles. The type of time (either CPU time or real time) is determined by the value of the **TIMETYPE=** option. The value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment. This option works only when you also specify **MODE=ALL_CYCLES**.
MINLENGTH=number
specifies the minimum number of links to allow in a cycle. Any cycle that has fewer links than number is removed from the results. By default MINLENGTH=1 and nothing is filtered. This option works only when you also specify MODE=ALL_CYCLES.

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

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

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

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL_CYCLES</td>
<td>Returns all (unique, elementary) cycles found.</td>
</tr>
<tr>
<td>FIRST_CYCLE</td>
<td>Returns the first cycle found. This is the default.</td>
</tr>
</tbody>
</table>

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

DATA_LINKS_VAR Statement

DATA_LINKS_VAR < options > ;
LINKS_VAR < options > ;

The DATA_LINKS_VAR statement enables you to explicitly define the data set variable names for PROC OPTGRAPH to use when it reads the data set that is specified in the DATA_LINKS= option in the PROC OPTGRAPH statement. The format of the links input data set is defined in the section “Link Input Data” on page 50.

You can specify the following options in the DATA_LINKS_VAR statement:

FROM=column
specifies the data set variable name for the from nodes. The value of the column variable can be numeric or character.
LOWER=column
specifies the data set variable name for the link flow lower bounds. The value of the column variable must be numeric.

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

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

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

**DATA_MATRIX_VAR Statement**

```
DATA_MATRIX_VAR <column1,column2,...> ;
MATRIX_VAR <column1,column2,...> ;
```

The DATA_MATRIX_VAR statement enables you to explicitly define the data set variable names for PROC OPTGRAPH to use when it reads the data set that is specified in the DATA_MATRIX= option in the PROC OPTGRAPH statement. The format of the matrix input data set is defined in the section “Matrix Input Data” on page 58. The value of each column variable must be numeric.

**DATA_NODES_VAR Statement**

```
DATA_NODES_VAR < options > ;
NODES_VAR < options > ;
```

The DATA_NODES_VAR statement enables you to explicitly define the data set variable names for PROC OPTGRAPH to use when it reads the data set that is specified in the DATA_NODES= option in the PROC OPTGRAPH statement. The format of the node input data set is defined in the section “Node Input Data” on page 54.

You can specify the following options in the DATA_NODES_VAR statement:

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

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

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

EIGENVECTOR Statement

EIGENVECTOR < options > ;
EIGEN < options > ;

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

Eigenvectors and eigenvalues are described in the section “Eigenvector Problem” on page 114.

You can specify the following options in the EIGENVECTOR statement:

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

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA</td>
<td>Calculates the n largest algebraic eigenvalues (and their corresponding eigenvectors), where n is the value of the NEIGEN= option. This is the default.</td>
</tr>
<tr>
<td>SA</td>
<td>Calculates the n smallest algebraic eigenvalues (and their corresponding eigenvectors), where n is the value of the NEIGEN= option.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all algorithm-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the algorithmic processing</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the algorithmic processing</td>
</tr>
</tbody>
</table>
The default is the value that is specified in the `LOGLEVEL=` option in the `PROC OPTGRAPH` statement (or `BASIC` if that option is not specified).

`MAXITER=number`

specifies the maximum number of matrix-vector multiplications used in the Jacobi-Davidson algorithm to calculate eigenvectors. By default, `MAXITER=10,000`.

`NEIGEN=number`

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

`OUT=SAS-data-set`

specifies the output data set to contain the eigenvectors (and eigenvalues) found.

---

**LINEAR_ASSIGNMENT Statement**

```sas
LINEAR_ASSIGNMENT < options > ;
LAP < options > ;
```

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

The linear assignment problem is described in the section “Linear Assignment (Matching)” on page 116.

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

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

- `LOGLEVEL=number | string`
  
  controls the amount of information that is displayed in the SAS log. Table 1.32 describes the valid values for this option.

<table>
<thead>
<tr>
<th><code>number</code></th>
<th><code>string</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all algorithm-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the algorithmic processing</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the algorithmic processing</td>
</tr>
</tbody>
</table>

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

**OUT=SAS-data-set**

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

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

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

---

**MINCOSTFLOW Statement**

```
MINCOSTFLOW < options > ;
MCF < options > ;
```

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

The minimum-cost network flow problem is described in the section “Minimum-Cost Network Flow” on page 117.

You can specify the following options in the MINCOSTFLOW statement:

**LOGFREQ=number**

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

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

**LOGLEVEL=number | string**

controls the amount of information that is displayed in the SAS log. Table 1.33 describes the valid values for this option.

**Table 1.33  Values for LOGLEVEL= Option**

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all algorithm-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the algorithmic processing including a progress log using the interval that is specified in the LOGFREQ= option</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the algorithmic processing including a progress log using the interval that is specified in the LOGFREQ= option</td>
</tr>
</tbody>
</table>
MINCUT Statement

MINCUT < options > ;

The MINCUT statement invokes an algorithm that finds the minimum link-weighted cut of an input graph. The minimum-cut problem is described in the section “Minimum Cut” on page 125.

You can specify the following options in the MINCUT statement:

LOGLEVEL=number | string

controls the amount of information that is displayed in the SAS log. Table 1.34 describes the valid values for this option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all algorithm-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the algorithmic processing</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the algorithmic processing</td>
</tr>
</tbody>
</table>

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

MAXNUMCUTS=number

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

MAXWEIGHT=number

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

OUT=SAS-data-set

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

MINSPANTREE < options > ;

MST < options > ;

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

The minimum spanning tree problem is described in the section “Minimum Spanning Tree” on page 129.

You can specify the following options in the MINSPANTREE statement:

LOGLEVEL=number | string
    controls the amount of information that is displayed in the SAS log. Table 1.35 describes the valid values for this option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all algorithm-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the algorithmic processing</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the algorithmic processing</td>
</tr>
</tbody>
</table>

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

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

PERFORMANCE Statement

PERFORMANCE < performance-options > ;

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

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

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

DETAILS
    requests that PROC OPTGRAPH produce the Timing ODS table, which shows a breakdown of the time used in each step of the procedure.
**NTHREADS=number | CPUCOUNT**

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

Setting this option to a number greater than the number of available cores might result in reduced performance. Specifying a high number does not guarantee shorter solution time; the actual change in solution time depends on the computing hardware and the scalability of the underlying algorithms in the PROC OPTGRAPH. In some circumstances, the OPTGRAPH procedure might use fewer threads than the specified number because the procedure’s internal algorithms have determined that a smaller number is preferable.

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

```plaintext
proc optgraph
  data_links = LinkSetIn
  graph_direction = directed
  out_nodes = NodeSetOut;
  performance
    nthreads = 8;
run;
```

**REACH Statement**

```
REACH < options >;
```

The REACH statement invokes an algorithm that calculates the reach (ego) network on an input graph. The reach network is described in the section “Reach (Ego) Network” on page 131.

You can specify the following options in the REACH statement:

**BY_CLUSTER**

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

**DIGRAPH**

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

**EACH_SOURCE**

 treats each node as a source and calculates a reach network from each one.
Chapter 1: The OPTGRAPH Procedure

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

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

**LOGFREQTIME=** _number_
displays iteration logs for the reach algorithm every _number_ seconds. When PROC OPTGRAPH runs the reach algorithm, it displays the number of source networks that have completed. When you also specify the BY_CLUSTER option in the REACH statement, PROC OPTGRAPH displays the number of subgraphs that have completed. The value of _number_ can be any integer greater than or equal to 1; the default is 5. Setting this value too low can hurt performance on large-scale graphs.

**LOGLEVEL=** _number_
controls the amount of information that is displayed in the SAS log. Table 1.36 describes the valid values for this option.

<table>
<thead>
<tr>
<th><em>number</em></th>
<th><em>string</em></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all algorithm-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the algorithmic processing</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the algorithmic processing</td>
</tr>
</tbody>
</table>

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

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

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

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

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

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

```sas
SHORTPATH < options > ;
```

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

The shortest path algorithm is described in the section “Shortest Path” on page 143.

You can specify the following options in the SHORTPATH statement:

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

- **LOGLEVEL=number**
  controls the amount of information that is displayed in the SAS log. Table 1.37 describes the valid values for this option.

<table>
<thead>
<tr>
<th><code>number</code></th>
<th><code>string</code></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all algorithm-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the algorithmic processing</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the algorithmic processing</td>
</tr>
</tbody>
</table>

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

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

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

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

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL</td>
<td>Outputs shortest paths for all pairs of source-sinks.</td>
</tr>
<tr>
<td>LONGEST</td>
<td>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.</td>
</tr>
</tbody>
</table>
Table 1.38  (continued)

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHORTEST</td>
<td>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.</td>
</tr>
</tbody>
</table>

By default, PATHS=ALL.

**SINK**=sink-node

specifies the sink node for shortest paths calculations. This setting overrides the use of the variable sink in the data set that is specified in the DATA_N D多样化ES_SUB= option in the PROC OPTGRAPH statement.

**SOURCE**=source-node

specifies the source node for shortest paths calculations. This setting overrides the use of the variable source in the data set that is specified in the DATA_N D多样化ES_SUB= option in the PROC OPTGRAPH statement.

**USEWEIGHT**=YES | NO

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

Table 1.39  Values for the WEIGHT= Option

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>YES</td>
<td>Uses weights (if they exist) in shortest path calculations. This is the default.</td>
</tr>
<tr>
<td>NO</td>
<td>Does not use weights in shortest path calculations.</td>
</tr>
</tbody>
</table>

**WEIGHT2**=column

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

**SUMMARY Statement**

```plaintext
SUMMARY < options > ;
```

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

The summary metrics are described in the section “Summary” on page 154.

You can specify the following options in the SUMMARY statement:

**BICONCOMP**

specifies whether to calculate information about biconnected components. The graph must be undirected.
**BY_CLUSTER**
specifies whether to decompose the calculations by cluster (or subgraph). If this option is specified, PROC OPTGRAPH looks for a definition of the clusters in the input data set specified in the DATA_NODES= option.

**CONCOMP**
specifies whether to calculate information about connected components.

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

<table>
<thead>
<tr>
<th>Option Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEIGHT</td>
<td>Calculates approximate diameter based on the weighted graph.</td>
</tr>
<tr>
<td>UNWEIGHT</td>
<td>Calculates approximate diameter based on the unweighted graph.</td>
</tr>
<tr>
<td>BOTH</td>
<td>Calculates approximate diameter based on both weighted and unweighted graphs.</td>
</tr>
</tbody>
</table>

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

**LOGFREQNODE=number**
controls the frequency for displaying iteration logs for some of the summary metrics. For computationally intensive summary metrics such as shortest path, this option displays progress every number nodes. If you also specify the BY_CLUSTER option in this statement or a value greater than 1 for the NTHREADS= option in the PERFORMANCE statement, this option is ignored and the display frequency is determined by using the LOGFREQTIME= option instead. The value of number can be any integer greater than or equal to 1. The default is determined automatically based on the size of the graph. Setting this value too low can hurt performance on large-scale graphs.

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

**LOGLEVEL=number**
controls the amount of information that is displayed in the SAS log. Table 1.41 describes the valid values for this option.
Table 1.41  Values for LOGLEVEL= Option

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all algorithm-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the algorithmic processing</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the algorithmic processing</td>
</tr>
</tbody>
</table>

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

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

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

Table 1.42  Values for the SHORTPATH= Option

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEIGHT</td>
<td></td>
<td>Calculates shortest paths based on the weighted graph.</td>
</tr>
<tr>
<td>UNWEIGHT</td>
<td></td>
<td>Calculates shortest paths based on the unweighted graph.</td>
</tr>
<tr>
<td>BOTH</td>
<td></td>
<td>Calculates shortest paths based on both weighted and unweighted graphs.</td>
</tr>
</tbody>
</table>

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

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

---

**TRANSITIVE_CLOSURE Statement**

TRANSITIVE_CLOSURE < options >;
TRANSCL < options >;

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

Transitive closure is described in the section “Transitive Closure” on page 163.

You can specify the following options in the TRANSITIVE_CLOSURE statement:
**LOGLEVEL=number**
controls the amount of information that is displayed in the SAS log. Table 1.43 describes the valid values for this option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NONE</td>
<td>Turns off all algorithm-related messages in the SAS log</td>
</tr>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Displays a basic summary of the algorithmic processing</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Displays a summary of the algorithmic processing</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Displays a detailed summary of the algorithmic processing</td>
</tr>
</tbody>
</table>

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

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

---

**TSP Statement**

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

You can specify the following **options**:

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

**CONFLICTSEARCH=number | string**
specifies the level of conflict search that PROC OPTGRAPH performs. The solver performs a conflict search to find clauses that result from infeasible subproblems that arise in the search tree. Table 1.44 describes the valid values for this option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>–1</td>
<td>AUTOMATIC</td>
<td>Performs a conflict search based on a strategy that is determined by PROC OPTGRAPH</td>
</tr>
<tr>
<td>0</td>
<td>NONE</td>
<td>Disables conflict search</td>
</tr>
<tr>
<td>1</td>
<td>MODERATE</td>
<td>Performs a moderate conflict search</td>
</tr>
<tr>
<td>2</td>
<td>AGGRESSIVE</td>
<td>Performs an aggressive conflict search</td>
</tr>
</tbody>
</table>
By default, CONFLICTSEARCH=AUTOMATIC.

CUTOFF=number

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

CUTSTRATEGY=number | string

specifies the level of mixed integer linear programming cutting planes to be generated by PROC OPTGRAPH. TSP-specific cutting planes are always generated. Table 1.45 describes the valid values for this option.

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

By default, CUTSTRATEGY=None.

EMPHASIS=number | string

specifies a search emphasis option. Table 1.46 describes the valid values for this option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>BALANCE</td>
<td>Performs a balanced search</td>
</tr>
<tr>
<td>1</td>
<td>OPTIMAL</td>
<td>Emphasizes optimality over feasibility</td>
</tr>
<tr>
<td>2</td>
<td>FEASIBLE</td>
<td>Emphasizes feasibility over optimality</td>
</tr>
</tbody>
</table>

By default, EMPHASIS=BALANCE.

HEURISTICS=number | string

controls the level of initial and primal heuristics that PROC OPTGRAPH applies. This level determines how frequently PROC OPTGRAPH applies primal heuristics during the branch-and-bound tree search. It also affects the maximum number of iterations that are allowed in iterative heuristics. Some computationally expensive heuristics might be disabled by the solver at less aggressive levels. Table 1.47 lists the valid values for this option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>−1</td>
<td>AUTOMATIC</td>
<td>Applies the default level of heuristics</td>
</tr>
<tr>
<td>0</td>
<td>NONE</td>
<td>Disables all initial and primal heuristics</td>
</tr>
</tbody>
</table>

By default, CONFLICTSEARCH=AUTOMATIC.
Table 1.47 (continued)

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BASIC</td>
<td>Applies basic initial and primal heuristics at low frequency</td>
</tr>
<tr>
<td>2</td>
<td>MODERATE</td>
<td>Applies most initial and primal heuristics at moderate frequency</td>
</tr>
<tr>
<td>3</td>
<td>AGGRESSIVE</td>
<td>Applies all initial primal heuristics at high frequency</td>
</tr>
</tbody>
</table>

By default, HEURISTICS=AUTOMATIC.

**LOGFREQ=**\(\text{number}\)

specifies how often to print information in the branch-and-bound node log. The value of \(\text{number}\) can be any nonnegative integer up to the largest four-byte signed integer, which is \(2^{31} - 1\). The default value is 100. If \(\text{number}\) is set to 0, then the node log is disabled. If \(\text{number}\) 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 \(\text{number}\). An entry is also made each time a better integer solution is found.

**LOGLEVEL=**\(\text{number} \mid \text{string}\)

controls the amount of information displayed in the SAS log by the solver, from a short description of presolve information and summary to details at each branch-and-bound node. Table 1.48 describes the valid values for this option.

Table 1.48 Values for LOGLEVEL= Option

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

The default value is MODERATE.

**MAXNODES=**\(\text{number}\)

specifies the maximum number of branch-and-bound nodes to be processed. The value of \(\text{number}\) 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=**\(\text{number}\)

specifies a stopping criterion. If \(\text{number}\) solutions have been found, then the procedure stops. The value of \(\text{number}\) 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=**\(\text{number}\)

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 \(\text{number}\) 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.
**Chapter 1: The OPTGRAPH Procedure**

**MILP=** specifies whether to use a mixed integer linear programming (MILP) solver for solving the traveling salesman problem. The MILP solver attempts to find the overall best TSP tour by using a branch-and-bound based algorithm. This algorithm can be expensive for large-scale problems. If MILP=OFF, then PROC OPTGRAPH uses its initial heuristics to find a feasible, but not necessarily optimal, tour as quickly as possible. Table 1.49 describes the valid values for this option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ON</td>
<td>Uses a mixed integer linear programming solver</td>
</tr>
<tr>
<td>0</td>
<td>OFF</td>
<td>Does not use a mixed integer linear programming solver</td>
</tr>
</tbody>
</table>

By default, MILP=ON.

**NODESEL=** specifies the branch-and-bound node selection strategy option. Table 1.50 describes the valid values for this option.

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

By default, NODESEL=AUTOMATIC. For more information about node selection, see Chapter 13, “The OPTMILP Procedure” (SAS/OR User’s Guide: Mathematical Programming).

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

**PROBE=** specifies a probing option. Table 1.51 describes the valid values for this option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>–1</td>
<td>AUTOMATIC</td>
<td>Uses an automatic probing strategy</td>
</tr>
<tr>
<td>0</td>
<td>NONE</td>
<td>Disables probing</td>
</tr>
<tr>
<td>1</td>
<td>MODERATE</td>
<td>Uses the probing moderately</td>
</tr>
<tr>
<td>2</td>
<td>AGGRESSIVE</td>
<td>Uses the probing aggressively</td>
</tr>
</tbody>
</table>

By default, PROBE=NONE.
**RELOBJGAP=** *number*

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

\[
|\text{BestInteger} - \text{BestBound}| / (10^{-10} + |\text{BestBound}|)
\]

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

**STRONGITER=** *number | AUTOMATIC*

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

**STRONGLEN=** *number | AUTOMATIC*

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

**TARGET=** *number*

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

**VARSEL=** *number | string*

specifies the rule for selecting the branching variable. **Table 1.52** describes the valid values for this option.

<table>
<thead>
<tr>
<th>number</th>
<th>string</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>–1</td>
<td>AUTOMATIC</td>
<td>Uses automatic branching variable selection</td>
</tr>
<tr>
<td>0</td>
<td>MAXINFEAS</td>
<td>Chooses the variable that has maximum infeasibility</td>
</tr>
<tr>
<td>1</td>
<td>MININFEAS</td>
<td>Chooses the variable that has minimum infeasibility</td>
</tr>
<tr>
<td>2</td>
<td>PSEUDO</td>
<td>Chooses a branching variable based on pseudocost</td>
</tr>
<tr>
<td>3</td>
<td>STRONG</td>
<td>Uses the strong branching variable selection strategy</td>
</tr>
</tbody>
</table>

By default, **VARSEL=AUTOMATIC**. For more information about variable selection, see Chapter 13, “The OPTMILP Procedure” (SAS/OR User’s Guide: Mathematical Programming).
Graph Input Data

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

Consider the directed graph shown in Figure 1.8.

![Figure 1.8 A Simple Directed Graph](image)

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

Link Input Data

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

- **from**: the from node (this variable can be numeric or character)
- **to**: the to node (this variable can be numeric or character)
- **weight**: the link weight (this variable must be numeric)
- **weight2**: the auxiliary link weight (this variable must be numeric)
- **lower**: the link flow lower bound (this variable must be numeric)
- **upper**: the link flow upper bound (this variable must be numeric)
As described in the `GRAPH_DIRECTION=` option, if the graph is undirected, the `from` and `to` labels are interchangeable. If the weights are not given for algorithms that call for link weights, they are all assumed to be 1.

The data set variable names can have any values that you want. If you use nonstandard names, you must identify the variables by using the `DATA_LINKS_VAR` statement, as described in the section “`DATA_LINKS_VAR` Statement” on page 32.

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

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

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

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

```sas
proc optgraph
   data_links = LinkSetInA;
run;

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

The directed graph $G$ shown in Figure 1.8 can be represented by the following links data set `LinkSetIn`:

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

The following statements read in this graph, declare it as a directed graph, and output the resulting links and nodes data sets. These statements do not run any algorithms, so the resulting output contains only the input graph.
proc optgraph
  graph_direction = directed
  data_links = LinkSetIn
  out_nodes = NodeSetOut
  out_links = LinkSetOut;
run;

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

**Figure 1.9** Node Data Set of a Simple Directed Graph

<table>
<thead>
<tr>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
</tr>
<tr>
<td>B</td>
</tr>
<tr>
<td>C</td>
</tr>
<tr>
<td>D</td>
</tr>
<tr>
<td>E</td>
</tr>
<tr>
<td>F</td>
</tr>
<tr>
<td>G</td>
</tr>
<tr>
<td>H</td>
</tr>
<tr>
<td>I</td>
</tr>
</tbody>
</table>

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

**Figure 1.10** Link Data Set of a Simple Directed Graph

<table>
<thead>
<tr>
<th>Obs</th>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>D</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>B</td>
<td>F</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>D</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>E</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>E</td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>F</td>
<td>G</td>
<td>6</td>
</tr>
<tr>
<td>12</td>
<td>G</td>
<td>H</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>G</td>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>H</td>
<td>G</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>H</td>
<td>I</td>
<td>3</td>
</tr>
</tbody>
</table>

If you define this graph as undirected, then reciprocal links (for example, $D \rightarrow E$ and $D \leftarrow E$) are treated as the same link, and duplicates are removed. PROC OPTGRAPH takes the first occurrence of the link and ignores the others. By default, GRAPH_DIRECTION=UNDIRECTED, so you can just remove this option to declare the graph as undirected.
```sas
proc optgraph
  data_links = LinkSetIn
  out_nodes = NodeSetOut
  out_links = LinkSetOut;
run;
```

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

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

```
NOTE: Running OPTGRAPH version 14.3.
NOTE: The OPTGRAPH procedure is executing in single-machine mode.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
WARNING: Link (E,D) in observation 9 of the DATA_LINKS= data set is a duplicate and is ignored.
WARNING: Link (H,G) in observation 14 of the DATA_LINKS= data set is a duplicate and is ignored.
NOTE: The number of nodes in the input graph is 9.
NOTE: The number of links in the input graph is 13.
NOTE: Data output used 0.01 (cpu: 0.00) seconds.
```

The data set NodeSetOut is equivalent to the one shown in Figure 1.9. However, the new links data set LinkSetOut shown in Figure 1.12 contains two fewer links than before, because duplicates are removed.

**Figure 1.12** Link Data Set of a Simple Undirected Graph

<table>
<thead>
<tr>
<th>Obs</th>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>D</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>B</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>B</td>
<td>F</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>D</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>E</td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>F</td>
<td>G</td>
<td>6</td>
</tr>
<tr>
<td>11</td>
<td>G</td>
<td>H</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>G</td>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>H</td>
<td>I</td>
<td>3</td>
</tr>
</tbody>
</table>

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

**Node Input Data**

The DATA_NODES= option in the PROC OPTGRAPH statement defines the data set that contains the list of nodes in the graph. This data set is used to define clusters (subgraphs) or to assign node weights.

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

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

The variable `cluster` is used to define clusters (subgraphs) for decomposing the input graph into subgraphs for processing. This is useful for the algorithms specified in the CENTRALITY, REACH, and SUMMARY statements. The use of the variable `cluster` is explained in more detail in the section “Processing by Cluster” on page 79.

You can specify any values that you want for the data set variable names. If you use nonstandard names, you must identify the variables by using the DATA_NODES_VAR statement, as described in the section “DATA_NODES_VAR Statement” on page 33.

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

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

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

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

For some algorithms, you might want to process only a subset of the nodes that appear in the input graph. You can accomplish this by using the DATA_NODES_SUB= option in the PROC OPTGRAPH statement. You can use the node subset data set in conjunction with the SHORTPATH or REACH statement. (See the sections “Shortest Path” on page 143 and “Reach (Ego) Network” on page 131, respectively.) The node subset data set is expected to contain some combination of the following variables:

- **node**: the node label (this variable can be numeric or character)
- **source**: whether to process this node as a source node in shortest path algorithms (this variable must be numeric)
- **sink**: whether to process this node as a sink node in shortest path algorithms (this variable must be numeric)
- **reach**: for the reach algorithm, the index of the source subgraph for processing (this variable must be numeric)

Table 1.53 shows how PROC OPTGRAPH processes nodes for each algorithm type. The missing indicator (.) can also be used in place of 0 to designate that a node is not to be processed.

<table>
<thead>
<tr>
<th>Algorithm Type</th>
<th>Variable Designations</th>
<th>Example Shown In:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shortest path</td>
<td>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.</td>
<td>The section “Shortest Path” on page 143</td>
</tr>
<tr>
<td>Reach</td>
<td>A value of 0 for the reach variable designates that the node is not to be processed. A value greater than 0 defines a marker for the source subgraph to which this node belongs. All nodes with the same marker are processed together as source nodes.</td>
<td>The section “Reach (Ego) Network” on page 131</td>
</tr>
</tbody>
</table>

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

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

The data set NodeSubSetIn indicates that you want to process the following:
• the reach network from the subgraph defined by node A
• the reach network from the subgraph defined by nodes F and E
• the shortest paths for the source-sink pairs in \( \{A, E\} \times \{F\} \)

**Standardized Labels**

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

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

Consider the following links data set that uses numeric labels:

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

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

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

The log is shown in Figure 1.13.
Figure 1.13  PROC OPTGRAPH Log: A Simple Undirected Graph

NOTE: Running OPTGRAPH version 14.3.
NOTE: The OPTGRAPH procedure is executing in single-machine mode.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 4.
NOTE: The number of links in the input graph is 3.
NOTE: Data output used 0.01 (cpu: 0.00) seconds.
NOTE: The data set WORK.NODESETOUT has 4 observations and 1 variables.
NOTE: The data set WORK.LINKSETOUT has 3 observations and 3 variables.

The data set NodeSetOut, shown in Figure 1.14, contains the unique numeric node labels, \{0, 1, 3, 5\}.

Figure 1.14  Node Data Set of a Simple Directed Graph

<table>
<thead>
<tr>
<th>Obs</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

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

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

The log that results from using standardized labels is shown in Figure 1.15.
The data set NodeSetOut, shown in Figure 1.16, now contains all node labels from 0 to 5, based on the assumptions when you use the STANDARDIZED LABELS option.

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

Matrix Input Data

This section describes the matrix input format that you can use with some of the algorithms in PROC OPTGRAPH. The DATA MATRIX= option in the PROC OPTGRAPH statement defines the data set that contains the matrix values. You can specify any values that you want for the data set variable names (the columns) by using the DATA MATRIX VAR statement, as described in the section “DATA MATRIX VAR Statement” on page 33. If you do not specify a DATA MATRIX VAR statement, then PROC OPTGRAPH assumes that all numeric variables in the data set are to be used in defining the matrix.

The following statements find the principal eigenvector of the square symmetric matrix that is defined in the data set Matrix:
data Matrix;
  input col1-col5;
datalines;
1 0 2 6 1
0 2 3 0 1
2 3 1 0 2
6 0 0 0 0
1 1 2 0 0;
;
proc optgraph
  data_matrix = Matrix;
eigenvector
    eigenvalues = LA
    nEigen = 1
    out = EigenVector;
run;

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

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

---

**Data Input Order**

Many algorithms are sensitive to the order in which PROC OPTGRAPH reads the data. If the order of the nodes or links is changed, either by you or by some parameter setting, the final result might change. In some cases, this difference is simply a permutation of identifiers (for example, connected components). In other cases, when you use local optimization (for example, community detection) or discrete branching decisions (for example, the traveling salesman problem), the final result might be a local (or alternative) solution. Two
parameters that could have such an effect are the STANDARDIZED_LABELS and NTHREADS= options. Both of these options can change the internal order of the nodes and links.

---

**Parallel Processing**

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

In addition, a number of the algorithms in PROC OPTGRAPH can also take advantage of multiple cores. There are two ways in which PROC OPTGRAPH can decompose the computational work in order to take advantage of parallel processing: by node and by subgraph.

To process the nodes of the graph individually, set the NTHREADS= option to some value greater than 1. You can do this for the centrality metrics closeness (see the section “Closeness Centrality” on page 70) and betweenness (see the section “Betweenness Centrality” on page 73). An example of this is shown in “Example 1.4: Betweenness and Closeness Centrality for Project Groups in a Research Department” on page 197.

To process the subgraphs of the original graph individually, set the NTHREADS= option to some value greater than 1, and designate the clusters in each node by using the cluster variable in the nodes data set, as described in the section “Node Input Data” on page 54. You can do this for centrality metrics, reach networks, and summary statistics. (See the sections “Centrality” on page 66, “Reach (Ego Network)” on page 131, and “Summary” on page 154, respectively.) A common use for this feature is to first decompose the original graph into communities or components. (See the sections “Community Detection” on page 90 and “Connected Components” on page 99, respectively.) Then, from these results, define the clusters in the node data set and run the analysis of each subgraph individually and in parallel. PROC OPTGRAPH takes care of all the accounting with the associated decomposition and returns results in terms of the original graph. An example of this process for centrality is shown in the section “Processing by Cluster” on page 79.

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

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

---

**Numeric Limitations**

Extremely large or extremely small numerical values might cause computational difficulties for some of the algorithms in PROC OPTGRAPH. For this reason, each algorithm restricts the magnitude of the data values to a particular threshold number. If the user data values exceed this threshold, PROC OPTGRAPH issues an error message. The value of the threshold limit is different for each algorithm and depends on the operating environment. The threshold limits are listed in Table 1.54, where $M$ is defined as the largest absolute value representable in your operating environment.
Table 1.54  Threshold Limits by Statement

<table>
<thead>
<tr>
<th>Statement</th>
<th>Matrix</th>
<th>Graph Links</th>
<th>Graph Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>weight</td>
<td>weight2</td>
<td>lower</td>
</tr>
<tr>
<td>CENTRALITY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUTH=, EIGEN=, HUB=</td>
<td>1e20</td>
<td>√M</td>
<td>√M</td>
</tr>
<tr>
<td>BETWEEN=, CLOSE=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INFLUENCE=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMMUNITY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CYCLE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EIGENVECTOR</td>
<td>1e20</td>
<td>1e20</td>
<td></td>
</tr>
<tr>
<td>LINEAR_ASSIGNMENT</td>
<td></td>
<td>√M</td>
<td>√M</td>
</tr>
<tr>
<td>MINCOSTFLOW</td>
<td>1e15</td>
<td>1e15</td>
<td>1e15</td>
</tr>
<tr>
<td>MINCUT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MINSPANTREE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REACH</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHORTPATH</td>
<td></td>
<td>√M</td>
<td>√M</td>
</tr>
<tr>
<td>SUMMARY</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIAMETER_APPROX=, SHORTPATH=</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TSP</td>
<td>1e20</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To obtain these limits, use the SAS function constant. For example, the following DATA step assigns $\sqrt{M}$ to a variable `x` and prints that value to the log:

```sas
data _null_
  x = constant('SQRTBIG');
  put x=;
run;
```

**Missing Values**

For most of the algorithms in PROC OPTGRAPH, there is no valid interpretation for a missing value. If the user data contain a missing value, PROC OPTGRAPH issues an error message. One exception is for the minimum-cost network flow solver when you are setting the link or node bounds. In this case, a missing value is interpreted as the default bound value, as described in the section “Minimum-Cost Network Flow” on page 117. Another exception is the linear assignment problem when you are using the matrix input format. A missing value in this case defines an invalid assignment between a row and a column of the matrix. An example of this is shown in the section “Linear Assignment (Matching)” on page 116.

**Negative Link Weights**

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

- CENTRALITY
  - AUTH=, BETWEEN=, CLOSE=, EIGEN=, HUB=
COMMUNITY

MINCUT

Zero Link Weights

For the community detection algorithm, a zero-valued link weight is not allowed. If a zero-valued link weight is provided, the community detection algorithm issues an error message.

Size Limitations

PROC OPTGRAPH can handle any graph whose number of nodes and links are less than or equal to 2,147,483,647 (the maximum that can be represented by a 32-bit integer). This maximum also applies to 64-bit systems. For graphs of two billion nodes (or links), memory limitations also become a limiting factor. For example, see the discussion of memory requirements for the community detection algorithm in the section “Memory Requirement” on page 92.

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

Common Notation and Assumptions

This section introduces some common notation and assumptions used throughout the chapter.

A complete graph, denoted $K(N)$, is a graph in which every pair of nodes in $N$ is connected by a link. The number of links in $K(N)$ is described in Table 1.55.

<table>
<thead>
<tr>
<th>Graph Direction</th>
<th>Default</th>
<th>INCLUDE_SELFLINK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Directed</td>
<td>$</td>
<td>N</td>
</tr>
<tr>
<td>Undirected</td>
<td>$</td>
<td>N</td>
</tr>
</tbody>
</table>

Biconnected Components and Articulation Points

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

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

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

The biconnected components algorithm reports status information in a macro variable called _OPTGRAPH_BICONCOMP_. For more information about this macro variable see the section “Macro Variable _OPTGRAPH_BICONCOMP_” on page 177.

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

### Biconnected Components of a Simple Undirected Graph

This section illustrates the use of the biconnected components algorithm on the simple undirected graph $G$ that is shown in Figure 1.17.

**Figure 1.17** A Simple Undirected Graph $G$

The undirected graph $G$ can be represented by the following links data set LinkSetInBiCC:
Chapter 1: The OPTGRAPH Procedure

```sas
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:

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

The data set LinkSetOut now contains the biconnected components of the input graph, as shown in Figure 1.18.

**Figure 1.18** Biconnected Components of a Simple Undirected Graph

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>biconcomp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>G</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>I</td>
<td>3</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
<td>3</td>
</tr>
<tr>
<td>H</td>
<td>I</td>
<td>3</td>
</tr>
</tbody>
</table>

In addition, the data set NodeSetOut contains the articulation points of the input graph, as shown in Figure 1.19.

**Figure 1.19** Articulation Points of a Simple Undirected Graph

<table>
<thead>
<tr>
<th>node</th>
<th>arpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>0</td>
</tr>
</tbody>
</table>
The biconnected components are shown graphically in Figure 1.20 and Figure 1.21.

**Figure 1.20** Biconnected Components $C^1$ and $C^2$

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

**Figure 1.21** Biconnected Components $C^3$ and $C^4$

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

For a more detailed example, see “Example 1.1: Articulation Points in a Terrorist Network” on page 188.
Chapter 1: The OPTGRAPH Procedure

Centrality

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

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

The CENTRALITY statement reports status information in a macro variable called _OPTGRAPH_CENTRALITY_. For more information about this macro variable, see the section “Macro Variable _OPTGRAPH_CENTRALITY_” on page 178.

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

Degree Centrality

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

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

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

As a simple example, consider again the directed graph in Figure 1.8 with data set LinkSetIn defined in the section “Link Input Data” on page 50. The following statements calculate the degree centrality for both in- and out-degree:

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

The node data set NodeSetOut now contains the degree centrality of the input graph. For a directed graph, the data set provides the in-degree (variable centr_degree_in), the out-degree (variable centr_degree_out), and the degree that is the sum of in- and out-degrees (variable centr_degree). This data set is shown in Figure 1.22.
Influence Centrality

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

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

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

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

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

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

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

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

```plaintext
proc optgraph
  graph_direction = directed
  data_links = LinkSetIn
  out_nodes = NodeSetOut;
  centrality
    influence = unweight;
run;
```
The node data set NodeSetOut now contains the unweighted influence centrality of the input graph, including the $C_1$ variable centr_influence1_unwt and the $C_2$ variable centr_influence2_unwt. This data set is shown in Figure 1.23.

**Figure 1.23** Influence Centrality of a Simple Directed Graph

<table>
<thead>
<tr>
<th>node</th>
<th>centr_influence1_unwt</th>
<th>centr_influence2_unwt</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.33333</td>
<td>0.55556</td>
</tr>
<tr>
<td>B</td>
<td>0.33333</td>
<td>0.44444</td>
</tr>
<tr>
<td>C</td>
<td>0.11111</td>
<td>0.22222</td>
</tr>
<tr>
<td>D</td>
<td>0.11111</td>
<td>0.22222</td>
</tr>
<tr>
<td>E</td>
<td>0.22222</td>
<td>0.22222</td>
</tr>
<tr>
<td>F</td>
<td>0.11111</td>
<td>0.22222</td>
</tr>
<tr>
<td>G</td>
<td>0.22222</td>
<td>0.22222</td>
</tr>
<tr>
<td>H</td>
<td>0.22222</td>
<td>0.22222</td>
</tr>
<tr>
<td>I</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

For a more detailed example, see “Example 1.2: Influence Centrality for Project Groups in a Research Department” on page 190.

**Clustering Coefficient**

The clustering coefficient for a node is the number of links between the nodes within its neighborhood divided by the number of links that could possibly exist between them (their induced complete graph).

Let $N_i$ represent the list of nodes that are connected to node $i$ (excluding itself). The formula for the clustering coefficient is

$$C(i) = \frac{|\{(u, v) \in A : u, v \in N_i\}|}{|K(N_i)|}$$

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

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

The algorithm that PROC OPTGRAPH uses to compute the clustering coefficient runs in time $O(|N|^3)$. Therefore, this algorithm is not expected to scale to very large graphs.

Consider the three undirected graphs on four nodes shown in Figure 1.24.
Define the three link data sets as follows:

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

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

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

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

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

proc optgraph
  data_links = LinkSetInCC2
  out_nodes = NodeSetOut2;
  centrality
    clustering_coef;
```
Chapter 1: The OPTGRAPH Procedure

run;

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

The node data sets provide the clustering coefficients for each graph (variable centr_cluster), as shown in Figure 1.25 through Figure 1.27.

! Figure 1.25 Clustering Coefficient of a Simple Undirected Graph 1

<table>
<thead>
<tr>
<th>node</th>
<th>centr_cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
</tr>
</tbody>
</table>

! Figure 1.26 Clustering Coefficient of a Simple Undirected Graph 2

<table>
<thead>
<tr>
<th>node</th>
<th>centr_cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.33333</td>
</tr>
<tr>
<td>B</td>
<td>0.00000</td>
</tr>
<tr>
<td>C</td>
<td>1.00000</td>
</tr>
<tr>
<td>D</td>
<td>1.00000</td>
</tr>
</tbody>
</table>

! Figure 1.27 Clustering Coefficient of a Simple Undirected Graph 3

<table>
<thead>
<tr>
<th>node</th>
<th>centr_cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
</tr>
</tbody>
</table>

Closeness Centrality

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

Define $d_{uv}$ to be the shortest path distance from node $u$ to node $v$.

Closeness Centrality for an Undirected Graph

For an undirected graph, $R(u) = \{v \in N : d_{uv} < \infty\}$ is the set of reachable nodes from node $u$. The set of unreachable nodes from node $u$ is $N \setminus R(u) = \{v \in N : d_{uv} = \infty\}$. The CLOSE_NOPATH= option specifies how to handle unreachable nodes.
For the special case in which all nodes are unreachable from node \( u \), the closeness centrality is defined as 0. Otherwise, closeness centrality is calculated as

\[
C_c(u) = s(u) \left( \frac{n(u)}{\sum_{v \in R(u)} d_{uv} + |N \setminus R(u)| \cdot p} \right)
\]

where \( p \) defines a penalty parameter for unreachable nodes, \( n(u) \) defines the number of nodes that are considered in calculating the average, and \( s(u) \) is a scaling factor, as shown in Table 1.56.

<table>
<thead>
<tr>
<th>CLOSE_NOPATH=</th>
<th>( p )</th>
<th>( n(u) )</th>
<th>( s(u) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIAMETER</td>
<td>( \max { d_{ij} : d_{ij} &lt; \infty } + 1 )</td>
<td>(</td>
<td>N</td>
</tr>
<tr>
<td>NNODES</td>
<td>(</td>
<td>N</td>
<td>)</td>
</tr>
<tr>
<td>ZERO</td>
<td>0</td>
<td>(</td>
<td>R(u)</td>
</tr>
</tbody>
</table>

**Closeness Centrality for a Directed Graph**

For a directed graph, \( R^{out}(u) = \{ v \in N : d_{uv} < \infty \} \) is the set of reachable nodes from node \( u \), whereas \( R^{in}(u) = \{ v \in N : d_{vu} < \infty \} \) is the set of nodes from which a finite path exists to node \( u \). The set of unreachable nodes from node \( u \) is \( N \setminus R^{out}(u) = \{ v \in N : d_{uv} = \infty \} \), whereas the set of nodes from which a finite path to node \( u \) does not exist is \( N \setminus R^{in}(u) = \{ v \in N : d_{vu} = \infty \} \).

For the special case in which all nodes are unreachable from node \( u \), the out-closeness centrality is defined as 0. Otherwise, out-closeness centrality is calculated as

\[
C_c^{out}(u) = s^{out}(u) \left( \frac{n^{out}(u)}{\sum_{v \in R^{out}(u)} d_{uv} + |N \setminus R^{out}(u)| \cdot p} \right)
\]

where \( n^{out}(u) \) defines the number of nodes that are considered in calculating the average and \( s^{out}(u) \) is a scaling factor, as shown in Table 1.57.

For the special case in which node \( u \) is unreachable from all other nodes, the in-closeness centrality is defined as 0. Otherwise, in-closeness centrality is calculated as

\[
C_c^{in}(u) = s^{in}(u) \left( \frac{n^{in}(u)}{\sum_{v \in R^{in}(u)} d_{vu} + |N \setminus R^{in}(u)| \cdot p} \right)
\]

where \( n^{in}(u) \) defines the number of nodes that are considered in calculating the average and \( s^{in}(u) \) is a scaling factor, as shown in Table 1.57.
Table 1.57  Formulas for CLOSE_NOPATH= Option for Directed Graphs

<table>
<thead>
<tr>
<th>CLOSE_NOPATH=</th>
<th>$n^{out}(u)$</th>
<th>$s^{out}(u)$</th>
<th>$n^{in}(u)$</th>
<th>$s^{in}(u)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIAMETER</td>
<td>$</td>
<td>N</td>
<td>- 1$</td>
<td>1</td>
</tr>
<tr>
<td>NNODES</td>
<td>$</td>
<td>N</td>
<td>- 1$</td>
<td>1</td>
</tr>
<tr>
<td>ZERO</td>
<td>$</td>
<td>R^{out}(u)</td>
<td>- 1$</td>
<td>$</td>
</tr>
</tbody>
</table>

The overall closeness centrality for directed graphs is calculated as

$$C_c(u) = \frac{C_c^{out}(u) + C_c^{in}(u)}{2}$$

**Harmonic Centrality**

Harmonic centrality, as described in Rochat (2009), is a variant of closeness centrality that attempts to simplify the treatment of unreachable nodes by calculating the average of the reciprocal of the shortest path distances from a particular node to all other nodes. The formula for harmonic centrality is

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

To enable the calculation of harmonic centrality, use the CLOSE_NOPATH=HARMONIC option.

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

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

Consider again the directed graph in Figure 1.8 with the data set LinkSetIn, which is defined in the section “Link Input Data” on page 50. The following statements calculate the closeness centrality for both the weighted and unweighted graphs:

```plaintext
proc optgraph
  graph_direction = directed
  data_links = LinkSetIn
  out_nodes = NodeSetOut;
  centrality
    close = both;
run;
```
The node data set NodeSetOut now contains the weighted and unweighted directed closeness centrality of the input graph. The data set provides the unweighted closeness (the centr_close_unwt variable), in-closeness (the centr_close_in_unwt variable), and out-closeness (the centr_close_out_unwt variable). It also provides the weighted variants centr_close_wt, centr_close_in_wt, and centr_close_out_wt. This data set is shown in Figure 1.28.

**Figure 1.28** Closeness Centrality of a Simple Directed Graph

<table>
<thead>
<tr>
<th>node</th>
<th>centr_close_wt</th>
<th>centr_close_in_wt</th>
<th>centr_close_out_wt</th>
<th>centr_close_unwt</th>
<th>centr_close_in_unwt</th>
<th>centr_close_out_unwt</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.08000</td>
<td>0.00000</td>
<td>0.16000</td>
<td>0.22222</td>
<td>0.00000</td>
<td>0.44444</td>
</tr>
<tr>
<td>B</td>
<td>0.11621</td>
<td>0.08696</td>
<td>0.14545</td>
<td>0.33333</td>
<td>0.22222</td>
<td>0.44444</td>
</tr>
<tr>
<td>C</td>
<td>0.11496</td>
<td>0.09877</td>
<td>0.13115</td>
<td>0.27885</td>
<td>0.25000</td>
<td>0.30769</td>
</tr>
<tr>
<td>D</td>
<td>0.12007</td>
<td>0.12903</td>
<td>0.11111</td>
<td>0.29178</td>
<td>0.30769</td>
<td>0.27586</td>
</tr>
<tr>
<td>E</td>
<td>0.12662</td>
<td>0.13559</td>
<td>0.11765</td>
<td>0.32000</td>
<td>0.32000</td>
<td>0.32000</td>
</tr>
<tr>
<td>F</td>
<td>0.11849</td>
<td>0.14286</td>
<td>0.09412</td>
<td>0.30725</td>
<td>0.34783</td>
<td>0.26667</td>
</tr>
<tr>
<td>G</td>
<td>0.10882</td>
<td>0.11765</td>
<td>0.10000</td>
<td>0.32500</td>
<td>0.40000</td>
<td>0.25000</td>
</tr>
<tr>
<td>H</td>
<td>0.10299</td>
<td>0.10959</td>
<td>0.09639</td>
<td>0.27885</td>
<td>0.30769</td>
<td>0.25000</td>
</tr>
<tr>
<td>I</td>
<td>0.06349</td>
<td>0.12698</td>
<td>0.00000</td>
<td>0.18182</td>
<td>0.36364</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

**Betweenness Centrality**

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

The formula for node betweenness centrality is

\[ C_B(u) = \sum_{s \neq u \neq t \in N} \frac{\sigma_{st}(u)}{\sigma_{st}} \]

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

The formula for link betweenness centrality is

\[ C_B(u, v) = \sum_{s,t \in N} \frac{\sigma_{st}(u, v)}{\sigma_{st}} \]

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

By default, this metric is normalized by dividing through by two times the number of pairs of nodes, not including \( u \), which is \((|N|−1)(|N|−2)\). You can disable this normalization by using the BETWEEN_NORM= option.

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

Betweenness centrality is calculated according to the value of the BETWEEN= option in the CENTRALITY statement. The node betweenness results are provided in the node output data set that is specified in the
Chapter 1: The OPTGRAPH Procedure

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

The algorithm used by PROC OPTGRAPH to compute betweenness centrality relies on calculating shortest paths for all source-sink pairs and runs in time $O(|N|^2 \times (|N| \log |N| + |A|))$. Therefore, it is not expected to scale to very large graphs. Similar to closeness centrality, because shortest path computations can be calculated independently (for each source node), the algorithm can be sped up by using the NTHREADS= option in the PERFORMANCE statement. When closeness and betweenness centrality are run together, PROC OPTGRAPH calculates both metrics in one run.

Consider again the directed graph in Figure 1.8 with data set LinkSetIn defined in the section “Link Input Data” on page 50. The following statements calculate the betweenness centrality for both the weighted and unweighted graphs:

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

The node data set NodeSetOut now contains the weighted (variable centr_between_wt) and unweighted (variable centr_between_unwt) node betweenness centrality of the input graph. This data set is shown in Figure 1.29.

**Figure 1.29** Node Betweenness Centrality of a Simple Directed Graph

<table>
<thead>
<tr>
<th>node</th>
<th>centr_between_wt</th>
<th>centr_between_unwt</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>B</td>
<td>0.07738</td>
<td>0.07738</td>
</tr>
<tr>
<td>C</td>
<td>0.12202</td>
<td>0.00595</td>
</tr>
<tr>
<td>D</td>
<td>0.00000</td>
<td>0.00595</td>
</tr>
<tr>
<td>E</td>
<td>0.33482</td>
<td>0.17857</td>
</tr>
<tr>
<td>F</td>
<td>0.26786</td>
<td>0.26786</td>
</tr>
<tr>
<td>G</td>
<td>0.22321</td>
<td>0.21429</td>
</tr>
<tr>
<td>H</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>I</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

In addition, the link data set LinkSetOut contains the weighted (variable centr_between_wt) and unweighted (variable centr_between_unwt) link betweenness centrality of the input graph. This data set is shown in Figure 1.30.
**Eigenvector Centrality**

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

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

where $x_i$ is the eigenvector centrality of node $i$, $\lambda$ is a constant, $\delta_i$ is the set of nodes that connect to node $i$, and $A_{ij}$ is the weight of the link from node $i$ to node $j$.

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

$$ Ax = \lambda x $$

As the preceding equation shows, $x$ is the eigenvector and $\lambda$ is the eigenvalue. Because $x$ should be positive, only the principal eigenvector that corresponds to the largest eigenvalue is of interest.

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

The following example illustrates the use of eigenvector centrality on the undirected graph $G$ shown in Figure 1.31.
The graph can be represented by the following links data set LinkSetIn:

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

The following statements compute the eigenvector centrality:

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

The data set NodeSetOut now contains the eigenvector centrality of each node. It is shown in Figure 1.32.
Even though nodes F and D both have the same degree of 2, node F has a higher eigenvector centrality than node D. This is because node F links to two important nodes (B and E), whereas node D links to one important node (B) and one unimportant node (A).

For a more detailed example, see “Example 1.5: Eigenvector Centrality for Word Sense Disambiguation” on page 200.

Hub and Authority Scoring

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

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

\[
\begin{align*}
x_i &= \alpha \sum_{j \in N} A_{ij} y_j \\
y_i &= \beta \sum_{j \in N} A_{ji} x_j
\end{align*}
\]

where \(x_i\) is the authority centrality of node \(i\), \(y_i\) is the hub centrality of node \(i\), \(A_{ij}\) is the weight of the link from node \(i\) to node \(j\), and \(\alpha\) and \(\beta\) are constants.

The definition can be written in matrix form as follows:

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

The following example illustrates the use of hub and authority scoring on the directed graph $G$ shown in Figure 1.33. Each node represents a web page. If web page $i$ has a hyperlink that points to web page $j$, then there is a directed link from $i$ to $j$.

![Figure 1.33 Hub and Authority Centrality Example of a Simple Directed Graph](image)

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

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

The following statements compute hub and authority centrality:

```plaintext
proc optgraph
  graph_direction = directed
  data_links = LinkSetIn
  out_nodes = NodeSetOut;
  centrality
    hub = unweight
    auth = unweight;
run;
```
The data set NodeSetOut now contains the hub and authority scores of each node. It is shown in Figure 1.34.

**Figure 1.34** Hub and Authority Centrality Output

<table>
<thead>
<tr>
<th>node</th>
<th>centr_hub_unwt</th>
<th>centr_auth_unwt</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>C</td>
<td>0.54135</td>
<td>0.00000</td>
</tr>
<tr>
<td>D</td>
<td>0.59703</td>
<td>0.11466</td>
</tr>
<tr>
<td>A</td>
<td>0.00000</td>
<td>0.10287</td>
</tr>
<tr>
<td>E</td>
<td>0.66549</td>
<td>0.84725</td>
</tr>
<tr>
<td>F</td>
<td>1.00000</td>
<td>0.11466</td>
</tr>
<tr>
<td>G</td>
<td>0.45865</td>
<td>0.00000</td>
</tr>
<tr>
<td>H</td>
<td>0.45865</td>
<td>0.00000</td>
</tr>
<tr>
<td>I</td>
<td>1.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>J</td>
<td>1.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>K</td>
<td>1.00000</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

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

**Weight Interpretation**

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

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

For a more detailed example, see “Example 1.6: Centrality Metrics for Project Groups in a Research Department” on page 203, which uses the WEIGHT2= option.

**Processing by Cluster**

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

**Centrality by Cluster for a Simple Undirected Graph**

Consider the graph depicted in Figure 1.35.
Figure 1.35 Undirected Graph

The following statements create the data set LinkSetIn:

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

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

Figure 1.36 Subgraph $N^0 = \{A, B, C, D, E\}$
The following data sets define the three induced subgraphs:

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

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

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

To calculate centrality metrics on the three subgraphs, you could run PROC OPTGRAPH three times, as follows:
proc optgraph
  data_links = LinkSetIn0
  out_nodes = NodeSetOut0;
  centrality
    degree = out
    influence = unweight
    close = unweight
    between = unweight
    eigen = unweight;
run;

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

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

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

**Figure 1.39** Centrality for Induced Subgraph 0

<table>
<thead>
<tr>
<th>node</th>
<th>centr_degree_out</th>
<th>centr_eigen_unwt</th>
<th>centr_close_unwt</th>
<th>centr_between_unwt</th>
<th>centr_influence1_unwt</th>
<th>centr_influence2_unwt</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>0.89897</td>
<td>0.80000</td>
<td>0.08333</td>
<td>0.6</td>
<td>1.6</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>0.70711</td>
<td>0.66667</td>
<td>0.00000</td>
<td>0.4</td>
<td>1.4</td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.58333</td>
<td>0.8</td>
<td>1.6</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>0.70711</td>
<td>0.66667</td>
<td>0.00000</td>
<td>0.4</td>
<td>1.4</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>0.37236</td>
<td>0.57143</td>
<td>0.00000</td>
<td>0.2</td>
<td>0.8</td>
</tr>
</tbody>
</table>

**Figure 1.40** Centrality for Induced Subgraph 1

<table>
<thead>
<tr>
<th>node</th>
<th>centr_degree_out</th>
<th>centr_eigen_unwt</th>
<th>centr_close_unwt</th>
<th>centr_between_unwt</th>
<th>centr_influence1_unwt</th>
<th>centr_influence2_unwt</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>3</td>
<td>1.00000</td>
<td>1.00</td>
<td>0.16667</td>
<td>0.75</td>
<td>1.75</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
<td>1.00000</td>
<td>1.00</td>
<td>0.16667</td>
<td>0.75</td>
<td>1.75</td>
</tr>
<tr>
<td>H</td>
<td>2</td>
<td>0.78078</td>
<td>0.75</td>
<td>0.00000</td>
<td>0.50</td>
<td>1.50</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
<td>0.78078</td>
<td>0.75</td>
<td>0.00000</td>
<td>0.50</td>
<td>1.50</td>
</tr>
</tbody>
</table>
A much more efficient way to process these graphs is to define the partition by using the `cluster` variable in the `nodes` data set and using the `BY_CLUSTER` option. Define the partitions of the original graph as follows:

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

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

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

Assuming that your machine has at least three cores, all three subgraphs are processed simultaneously with one call to PROC OPTGRAPH. The progress of the procedure is shown in Figure 1.42.
The results are shown in Figure 1.43.
Centrality by Community for a Simple Undirected Graph

The partition defined in the data set NodeSetIn could have also been calculated by PROC OPTGRAPH using a method called community detection. This method is discussed in the section “Community Detection” on page 90. First, call the community detection method as follows:

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

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

---

**Figure 1.43** Centrality for All Induced Subgraphs

<table>
<thead>
<tr>
<th>node</th>
<th>cluster</th>
<th>centr_degree_out</th>
<th>centr_eigen_unwt</th>
<th>centr_close_unwt</th>
<th>centr_between_unwt</th>
<th>centr_influence1_unwt</th>
<th>centr_influence2_unwt</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>3</td>
<td>0.89897</td>
<td>0.80000</td>
<td>0.08333</td>
<td>0.60000</td>
<td>1.60000</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>2</td>
<td>0.70711</td>
<td>0.66667</td>
<td>0.00000</td>
<td>0.40000</td>
<td>1.40000</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>4</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.58333</td>
<td>0.80000</td>
<td>1.60000</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>2</td>
<td>0.70711</td>
<td>0.66667</td>
<td>0.00000</td>
<td>0.40000</td>
<td>1.40000</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>1</td>
<td>0.37236</td>
<td>0.57143</td>
<td>0.00000</td>
<td>0.20000</td>
<td>0.80000</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>3</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.16667</td>
<td>0.75000</td>
<td>1.75000</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>3</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.16667</td>
<td>0.75000</td>
<td>1.75000</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
<td>2</td>
<td>0.78078</td>
<td>0.75000</td>
<td>0.00000</td>
<td>0.50000</td>
<td>1.50000</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
<td>2</td>
<td>0.78078</td>
<td>0.75000</td>
<td>0.00000</td>
<td>0.50000</td>
<td>1.50000</td>
</tr>
<tr>
<td>J</td>
<td>2</td>
<td>2</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.00000</td>
<td>0.66667</td>
<td>1.33333</td>
</tr>
<tr>
<td>K</td>
<td>2</td>
<td>2</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.00000</td>
<td>0.66667</td>
<td>1.33333</td>
</tr>
<tr>
<td>L</td>
<td>2</td>
<td>2</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.00000</td>
<td>0.66667</td>
<td>1.33333</td>
</tr>
</tbody>
</table>

---

**Figure 1.44** Communities for a Simple Undirected Graph

<table>
<thead>
<tr>
<th>node</th>
<th>community_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>J</td>
<td>2</td>
</tr>
<tr>
<td>K</td>
<td>2</td>
</tr>
<tr>
<td>L</td>
<td>2</td>
</tr>
</tbody>
</table>

To calculate centrality by induced subgraph, you can simply use the communities output as the nodes data set input and use the DATA_NODES_VAR statement to define the cluster variable:
Chapter 1: The OPTGRAPH Procedure

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

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

**Centrality by Filtered Community for a Simple Undirected Graph**

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

Returning to the data in the section “Centrality by Community for a Simple Undirected Graph” on page 85, you can use the filtering option as follows:

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

The results, shown in Figure 1.45, now contain only those subgraphs with node size greater than 3.
A clique of a graph $G = (N, A)$ is an induced subgraph that is a complete graph. Every node in a clique is connected to every other node in that clique. A maximal clique is a clique that is not a subset of the nodes of any larger clique. That is, it is a set $C$ of nodes such that every pair of nodes in $C$ is connected by a link and every node not in $C$ is missing a link to at least one node in $C$. The number of maximal cliques in a particular graph can be very large and can grow exponentially with every node added. Finding cliques in graphs has applications in numerous industries including bioinformatics, social networks, electrical engineering, and chemistry.

You can find the maximal cliques of an input graph by invoking the CLIQUE statement. The options for this statement are described in the section “CLIQUE Statement” on page 25. The clique algorithm works only with undirected graphs.

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

The clique algorithm reports status information in a macro variable called _OPTGRAPH_CLIQUE_. For more information about this macro variable, see the section “Macro Variable _OPTGRAPH_CLIQUE_” on page 178.

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

This section illustrates the use of the clique algorithm on the simple undirected graph $G$ that is shown in Figure 1.46.

Figure 1.46  A Simple Undirected Graph $G$

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

```latex
\begin{verbatim}
data LinkSetIn;												
ingput 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;
\end{verbatim}
```

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:

```latex
\begin{verbatim}
proc optgraph
  data_links = LinkSetIn;
  clique
    out   = Cliques;
run;

proc sql;
  create table CliqueSizes as
  select clique, count(*) as size
  from Cliques
  group by clique
  order by size desc;
quit;
\end{verbatim}
```
The data set Clique now contains the maximal cliques of the input graph; it is shown in Figure 1.47.

**Figure 1.47** Maximal Cliques of a Simple Undirected Graph

<table>
<thead>
<tr>
<th>clique node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
</tr>
<tr>
<td>1 2</td>
</tr>
<tr>
<td>1 1</td>
</tr>
<tr>
<td>1 3</td>
</tr>
<tr>
<td>1 4</td>
</tr>
<tr>
<td>2 0</td>
</tr>
<tr>
<td>2 2</td>
</tr>
<tr>
<td>2 5</td>
</tr>
<tr>
<td>2 6</td>
</tr>
<tr>
<td>3 2</td>
</tr>
<tr>
<td>3 8</td>
</tr>
<tr>
<td>3 7</td>
</tr>
<tr>
<td>4 8</td>
</tr>
<tr>
<td>4 9</td>
</tr>
</tbody>
</table>

In addition, the data set CliqueSizes contains the number of nodes in each clique; it is shown in Figure 1.48.

**Figure 1.48** Sizes of Maximal Cliques of a Simple Undirected Graph

<table>
<thead>
<tr>
<th>clique size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 5</td>
</tr>
<tr>
<td>2 4</td>
</tr>
<tr>
<td>3 3</td>
</tr>
<tr>
<td>4 2</td>
</tr>
</tbody>
</table>

The maximal cliques are shown graphically in Figure 1.49 and Figure 1.50.
Chapter 1: The OPTGRAPH Procedure

Figure 1.49  Maximal Cliques $C^1$ and $C^2$

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

![Graph showing maximal cliques $C^1$ and $C^2$]

Figure 1.50  Maximal Cliques $C^3$ and $C^4$

$C^3 = \{2, 7, 8\}$  \hspace{1cm} $C^4 = \{8, 9\}$

![Graph showing maximal cliques $C^3$ and $C^4$]

Community Detection

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

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

The COMMUNITY statement reports status information in a macro variable called _OPTGRAPH_COMMUNITY_. For more information about this macro variable, see the section “Macro Variable _OPTGRAPH_COMMUNITY_” on page 179.
When you specify ALGORITHM=PARALLEL_LABEL_PROP in the COMMUNITY statement, community detection supports both undirected and directed graphs. When you specify ALGORITHM=LOUVAIN or ALGORITHM=LABEL_PROP in the COMMUNITY statement, community detection is supported only for undirected graphs. For directed graphs, you need to aggregate directed links into undirected links before you call the algorithm. For example, suppose there are two directed links: a link from \( i \) to \( j \) with a link weight of 4.3, and a link from \( j \) to \( i \) with a link weight of 3.2. One common aggregation strategy is to sum the link weights together. Using this strategy, the weight of the undirected link between \( i \) and \( j \) is 7.5.

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

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

Mathematically, modularity is defined as

\[
Q = \frac{1}{2w} \sum_{(u,v) \in A} \left( w_{uv} - \frac{w_u w_v}{2w} \right) \Delta(c_u, c_v)
\]

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

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

The following is a brief description of the Louvain algorithm:

1. Initialize each node as its own community.

2. Move each node from its current community to the neighboring community that increases modularity the most. Repeat this step until modularity cannot be improved.

3. Group the nodes in each community into a supernode. Construct a new graph based on supernodes. Repeat these steps until modularity cannot be further improved or the maximum number of iterations has been reached.

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

1. Initialize each node as its own community.

2. Move each node from its current community to the neighboring community that has the maximum sum of link weights to the current node; break ties randomly if necessary. Repeat this step until there are no more movements.

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

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

**Parallel Community Detection**

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

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

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

**Memory Requirement**

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

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

When you specify GRAPH_INTERNAL_FORMAT=THIN and ALGORITHM=PARALLEL_LABEL_PROP, the memory required for community detection is approximately twice this amount.
Assume that your machine architecture is such that an integer is 4 bytes and a double is 8 bytes. Then, a graph with 100 million nodes and 650 million links would require approximately 21 gigabytes (GB) of memory when you specify ALGORITHM=LOUVAIN or ALGORITHM=LABEL_PROP:

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

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

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

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

**Graph Direction**

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

**Large Community**

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

**Recursive**

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

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

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

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

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

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

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

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

“Example 1.7: Community Detection on Zachary’s Karate Club Data” on page 206 shows the use of the RESOLUTION_LIST= option in the calculation of communities.
**Large Graphs**

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

- Use `GRAPH_INTERNAL_FORMAT=THIN` instead of `GRAPH_INTERNAL_FORMAT=FULL`. This enables PROC OPTGRAPH to store the data in memory compactly.

- Use the `LINK_REMOVAL_RATIO=` option to remove unimportant links. This practice can often dramatically improve the running time of large graphs.

**Output Data Sets**

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

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

**OUT_NODES= Data Set**

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

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

**OUT_LEVEL= Data Set**

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

- `level`: resolution level number
- `resolution`: resolution value
- `communities`: number of communities at the current resolution level
- `modularity`: modularity value at the current resolution level
Chapter 1: The OPTGRAPH Procedure

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

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

OUT_OVERLAP= Data Set
This data set describes the intensity of each node. At the end of community detection, a node could have links that connect to multiple communities. The intensity of a node is computed as the sum of the link weights that connect to nodes in the specified community divided by the total link weights of the node. This data set is computationally expensive to produce, and it requires a large amount of disk space. Therefore, this data set is not produced if you specify multiple resolution values in the RESOLUTION_LIST= option and ALGORITHM=PARALLEL_LABEL_PROP. However, if ALGORITHM=LOUVAIN, the data set is produced and will only contain results corresponding to the smallest value of the RESOLUTION_LIST= option. The data set contains the following columns:

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

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

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

This data set is not produced if you specify ALGORITHM=PARALLEL_LABEL_PROP together with multiple resolution values in the RESOLUTION_LIST= option.

Community Detection on a Simple Graph
This section illustrates the use of the community detection algorithm on the simple undirected graph \( G \) that is shown in Figure 1.51.
The undirected graph $G$ can be represented by the following links data set LinkSetIn:

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

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

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

The data set NodeSetOut contains the community identifier of each node and is shown in Figure 1.52.
Chapter 1: The OPTGRAPH Procedure

Figure 1.52 Community Detection on a Simple Graph: Nodes Output

<table>
<thead>
<tr>
<th>node</th>
<th>community_1</th>
<th>community_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>H</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

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

Figure 1.53 Community Detection on a Simple Graph: Level Output

<table>
<thead>
<tr>
<th>level</th>
<th>resolution</th>
<th>communities</th>
<th>modularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>3</td>
<td>0.39256</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>2</td>
<td>0.34298</td>
</tr>
</tbody>
</table>

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

Figure 1.54 Community Detection on a Simple Graph: Community Summary

<table>
<thead>
<tr>
<th>level</th>
<th>resolution</th>
<th>community</th>
<th>nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

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

Figure 1.55 Community Detection on a Simple Graph: Community Overlap

<table>
<thead>
<tr>
<th>node</th>
<th>community intensity</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0.66667</td>
</tr>
<tr>
<td>A</td>
<td>0.33333</td>
</tr>
<tr>
<td>B</td>
<td>1.00000</td>
</tr>
<tr>
<td>F</td>
<td>1.00000</td>
</tr>
<tr>
<td>G</td>
<td>0.33333</td>
</tr>
<tr>
<td>G</td>
<td>0.66667</td>
</tr>
<tr>
<td>C</td>
<td>1.00000</td>
</tr>
<tr>
<td>D</td>
<td>1.00000</td>
</tr>
<tr>
<td>E</td>
<td>1.00000</td>
</tr>
<tr>
<td>I</td>
<td>1.00000</td>
</tr>
<tr>
<td>H</td>
<td>1.00000</td>
</tr>
</tbody>
</table>
The data set CommLinksOut describes how the communities are connected and is shown in Figure 1.56.

**Figure 1.56** Community Detection on a Simple Graph: Intercommunity Links

<table>
<thead>
<tr>
<th>level</th>
<th>resolution</th>
<th>from_community</th>
<th>to_community</th>
<th>link_weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Connected Components**

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

In PROC OPTGRAPH, you can invoke connected components by using the CONCOMP statement. The options for this statement are described in the section “CONCOMP Statement” on page 29.

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

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

The connected components algorithm reports status information in a macro variable called _OPTGRAPH_CONCOMP_. For more information about this macro variable, see the section “Macro Variable _OPTGRAPH_CONCOMP_” on page 179.

**Connected Components of a Simple Undirected Graph**

This section illustrates the use of the connected components algorithm on the simple undirected graph \( G \) that is shown in Figure 1.57.
The undirected graph $G$ can be represented by the following links data set, LinkSetIn:

```plaintext
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:

```plaintext
proc optgraph
  data_links = LinkSetIn
  out_nodes = NodeSetOut;
  concomp;
run;
```

The data set NodeSetOut contains the connected components of the input graph and is shown in Figure 1.58.

Notice that the graph is defined by using only the links data set. As seen in Figure 1.57, this graph also contains a singleton node labeled J, which has no associated links. By definition, this node defines its own component. But because the input graph was defined by using only the links data set, it did not show up in
the results data set. To define a graph by using nodes that have no associated links, you should also define the input nodes data set. In this case, define the nodes data set NodeSetIn as follows:

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

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

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

The resulting data set, NodeSetOut, includes the singleton node J as its own component, as shown in Figure 1.59.

![Figure 1.59 Connected Components of a Simple Undirected Graph](image)

<table>
<thead>
<tr>
<th>node</th>
<th>concomp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
</tr>
<tr>
<td>J</td>
<td>3</td>
</tr>
<tr>
<td>K</td>
<td>4</td>
</tr>
<tr>
<td>L</td>
<td>4</td>
</tr>
</tbody>
</table>

**Connected Components of a Simple Directed Graph**

This section illustrates the use of the connected components algorithm on the simple directed graph $G$ that is shown in Figure 1.60.
The directed graph $G$ can be represented by the following links data set, LinkSetIn:

```plaintext
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:

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

The data set NodeSetOut, shown in Figure 1.61, now contains the connected components of the input graph.
Figure 1.61  Connected Components of a Simple Directed Graph

<table>
<thead>
<tr>
<th>node</th>
<th>concomp</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>H</td>
<td>2</td>
</tr>
</tbody>
</table>

The connected components are represented graphically in Figure 1.62.

Figure 1.62  Strongly Connected Components of Graph $G$
Chapter 1: The OPTGRAPH Procedure

Core Decomposition

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

In PROC OPTGRAPH, you can invoke core decomposition by using the CORE statement. The options for this statement are described in the section “CORE Statement” on page 30.

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

The core decomposition algorithm reports status information in a macro variable called _OPTGRAPH_CORE_. For more information about this macro variable, see the section “Macro Variable _OPTGRAPH_CORE_” on page 180.

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

Core Decomposition of a Simple Undirected Graph

This section illustrates the use of the core decomposition algorithm on the simple undirected graph \( G \) that is shown in Figure 1.63.
The undirected graph $G$ can be represented using the following nodes data set NodeSetIn and links data set LinkSetIn:

```plaintext
data NodeSetIn;
  input node $ @@;
  datalines;
v1  v2  v3  v4  v5
v6  v7  v8  v9  v10
v11 v12 v13 v14 v15
v16 v17 v18 v19
;

data LinkSetIn;
  input from $ to $ @@;
  datalines;
```

---

**Figure 1.63** Simple Undirected Graph
The following statements calculate the core decomposition and output the results in the data set NodeSetOut:

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

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

**Figure 1.64** Core Decomposition of a Simple Undirected Graph

<table>
<thead>
<tr>
<th>node</th>
<th>core_out</th>
</tr>
</thead>
<tbody>
<tr>
<td>v19</td>
<td>0</td>
</tr>
<tr>
<td>v1</td>
<td>1</td>
</tr>
<tr>
<td>v5</td>
<td>1</td>
</tr>
<tr>
<td>v6</td>
<td>1</td>
</tr>
<tr>
<td>v7</td>
<td>1</td>
</tr>
<tr>
<td>v11</td>
<td>1</td>
</tr>
<tr>
<td>v2</td>
<td>2</td>
</tr>
<tr>
<td>v3</td>
<td>2</td>
</tr>
<tr>
<td>v4</td>
<td>2</td>
</tr>
<tr>
<td>v8</td>
<td>2</td>
</tr>
<tr>
<td>v9</td>
<td>2</td>
</tr>
<tr>
<td>v10</td>
<td>2</td>
</tr>
<tr>
<td>v12</td>
<td>3</td>
</tr>
<tr>
<td>v18</td>
<td>3</td>
</tr>
<tr>
<td>v13</td>
<td>4</td>
</tr>
<tr>
<td>v14</td>
<td>4</td>
</tr>
<tr>
<td>v15</td>
<td>4</td>
</tr>
<tr>
<td>v16</td>
<td>4</td>
</tr>
<tr>
<td>v17</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 1.65 shows the graph layered by its core number.
Figure 1.65 Core Decomposition
Chapter 1: The OPTGRAPH Procedure

Cycle

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

In PROC OPTGRAPH, you can find (or just count) the elementary cycles of an input graph by invoking the CYCLE statement. The options for this statement are described in the section “CYCLE Statement” on page 30. To find the cycles and report them in an output data set, use the OUT= option. To simply count the cycles, do not use the OUT= option.

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

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

The cycle detection algorithm reports status information in a macro variable called _OPTGRAPH_CYCLE_. For more information about this macro variable, see the section “Macro Variable _OPTGRAPH_CYCLE_” on page 180.

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

If MODE=ALL_CYCLES and there are many cycles, the OUT= data set can become very large. It might be beneficial to check the number of cycles before you try to create the OUT= data set. When you specify MODE=FIRST_CYCLE, the algorithm returns the first cycle that it finds and stops processing. This should run relatively quickly. For large-scale graphs, the MINLINKWEIGHT= and MAXLINKWEIGHT= options might increase the computation time. For more information about these options, see the section “CYCLE Statement” on page 30.

Cycle Detection of a Simple Directed Graph

This section provides a simple example of using the cycle detection algorithm on the simple directed graph $G$ that is shown in Figure 1.66. Two other examples are “Example 1.9: Cycle Detection for Kidney Donor Exchange” on page 212, which shows the use of cycle detection for optimizing a kidney donor exchange, and “Example 1.13: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System” on page 224, which shows another application of cycle detection.
Figure 1.66 A Simple Directed Graph $G$

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

```plaintext
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:

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

The result is written to the log of the OPTGRAPH procedure, as shown in Figure 1.67.
Figure 1.67 PROC OPTGRAPH Log: Check the Existence of a Cycle in a Simple Directed Graph

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

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

The result is written to the log of the OPTGRAPH procedure, as shown in Figure 1.68.
Figure 1.68  PROC OPTGRAPH Log: Count the Number of Cycles in a Simple Directed Graph

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

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

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

Figure 1.69  First Cycle Found in a Simple Directed Graph

<table>
<thead>
<tr>
<th>cycle</th>
<th>order</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>A</td>
</tr>
</tbody>
</table>

The first cycle that is found in the input graph is shown graphically in Figure 1.70.
The following statements return all the cycles in the graph:

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

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

**Figure 1.71**  All Cycles in a Simple Directed Graph

<table>
<thead>
<tr>
<th>cycle</th>
<th>order</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>E</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>E</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>E</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>E</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>B</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>E</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>E</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>C</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>D</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>F</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>E</td>
</tr>
</tbody>
</table>

The six additional cycles are shown graphically in Figure 1.72 through Figure 1.74.
Figure 1.72  Cycles

\[ A \rightarrow E \rightarrow B \rightarrow C \rightarrow A \]

\[ A \rightarrow E \rightarrow C \rightarrow A \]

Figure 1.73  Cycles

\[ B \rightarrow C \rightarrow D \rightarrow E \rightarrow B \]

\[ B \rightarrow C \rightarrow D \rightarrow F \rightarrow E \rightarrow B \]
**Eigenvector Problem**

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

In PROC OPTGRAPH, you can calculate eigenvectors of a given matrix by invoking the EIGENVECTOR statement. The options for this statement are described in the section “EIGENVECTOR Statement” on page 34.

The EIGENVECTOR statement reports status information in a macro variable called _OPTGRAPH_EIGEN_. For more information about this macro variable, see the section “Macro Variable _OPTGRAPH_EIGEN_” on page 180.

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

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

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

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

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

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

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

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

![Figure 1.75](image.png)

**Figure 1.75** Eigenvector Problem for a Small Matrix with Dense Input

<table>
<thead>
<tr>
<th>obs</th>
<th>eigen_1</th>
<th>eigen_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.65778</td>
<td>0.32280</td>
</tr>
<tr>
<td>2</td>
<td>0.26459</td>
<td>-0.64125</td>
</tr>
<tr>
<td>3</td>
<td>0.40078</td>
<td>-0.49082</td>
</tr>
<tr>
<td>4</td>
<td>0.53174</td>
<td>0.40988</td>
</tr>
<tr>
<td>5</td>
<td>0.23227</td>
<td>-0.27513</td>
</tr>
<tr>
<td></td>
<td>7.42209</td>
<td>4.72527</td>
</tr>
</tbody>
</table>

Eigenvector Problem for a Small Matrix with Sparse Input

This section shows the use of a sparse input format for the eigenvector problem. The following statements define the same matrix that is used in the section “Eigenvector Problem for a Small Matrix with Dense Input” on page 115, but they represent it sparsely in the form of graph links:
data LinkSetIn;
  input from to weight;
datalines;
  0 0 1
  0 2 2
  0 3 6
  0 4 1
  1 1 2
  1 2 3
  1 4 1
  2 2 1
  2 4 2;

Notice that there are self links $i \to i$. These correspond to the diagonal entries in the matrix that is defined in the data set MatrixSetIn. By default, PROC OPTGRAPH ignores self links. Therefore, in the sparse format, you must use the INCLUDE_SELFLINK option to match the dense matrix from the section “Eigenvector Problem for a Small Matrix with Dense Input” on page 115. Now you can calculate the same eigenvectors using sparse input as follows:

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

The output is shown in Figure 1.76.

**Figure 1.76** Eigenvector Problem for a Small Matrix with Sparse Input

<table>
<thead>
<tr>
<th>node</th>
<th>eigen_1</th>
<th>eigen_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.65778</td>
<td>0.32280</td>
</tr>
<tr>
<td>2</td>
<td>0.40078</td>
<td>-0.49082</td>
</tr>
<tr>
<td>3</td>
<td>0.53174</td>
<td>0.40988</td>
</tr>
<tr>
<td>4</td>
<td>0.23227</td>
<td>-0.27513</td>
</tr>
<tr>
<td>1</td>
<td>0.26459</td>
<td>-0.64125</td>
</tr>
<tr>
<td></td>
<td>7.42209</td>
<td>4.72527</td>
</tr>
</tbody>
</table>

**Linear Assignment (Matching)**

The *linear assignment problem* (LAP) is a fundamental problem in combinatorial optimization that involves assigning workers to tasks at minimal costs. In graph theoretic terms, LAP is equivalent to finding a minimum-weight matching in a weighted bipartite directed graph. In a *bipartite graph*, the nodes can be divided into two disjoint sets $S$ (workers) and $T$ (tasks) such that every link connects a node in $S$ to a node in $T$. That is, the node sets $S$ and $T$ are independent. The concept of assigning workers to tasks can be generalized to the assignment of any abstract object from one group to some abstract object from a second group.
The linear assignment problem can be formulated as an integer programming optimization problem. The form of the problem depends on the sizes of the two input sets, \( S \) and \( T \). Let \( A \) represent the set of possible assignments between sets \( S \) and \( T \). In the bipartite graph, these assignments are the links. If \( |S| \geq |T| \), then the following optimization problem is solved:

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

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:

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

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

In PROC OPTGRAPH, you can invoke the linear assignment problem solver by using the LINEAR_ASSIGNMENT statement. The options for this statement are described in the section “LINEAR_ASSIGNMENT Statement” on page 35. The algorithm that the PROC OPTGRAPH uses for solving a LAP is based on augmentation of shortest paths (Jonker and Volgenant 1987). This algorithm can be applied to either matrix data input (see the section “Matrix Input Data” on page 58) or graph data input (see the section “Graph Input Data” on page 50) as long as the graph is bipartite.

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

The linear assignment problem solver reports status information in a macro variable called _OPTGRAPH_LAP_. For more information about this macro variable, see the section “Macro Variable _OPTGRAPH_LAP_” on page 181.

For a detailed example, see “Example 1.10: Linear Assignment Problem for Minimizing Swim Times” on page 217.

---

**Minimum-Cost Network Flow**

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

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

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

In PROC OPTGRAPH, you can invoke the minimum-cost network flow solver by using the MINCOSTFLOW statement. The options for this statement are described in the section “MINCOSTFLOW Statement” on page 36.

The minimum-cost network flow solver reports status information in a macro variable called _OPTGRAPH_MCF_. For more information about this macro variable, see the section “Macro Variable _OPTGRAPH_MCF_” on page 181.

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

The input for the network is the standard graph input, which is described in the section “Graph Input Data” on page 50. The links data set, which is specified in the DATA_LINKS= option in the PROC OPTGRAPH statement, contains the following columns:

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

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

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

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

To explicitly define an upper bound of \( \infty \), use the special missing value, (.I). To explicitly define a lower bound of \(-\infty\), use the special missing value, (.M). Related to infinite bounds, the following scenarios are not supported:
• The flow on a link must be bounded from below ($l_{ij} = -\infty$ is not allowed).

• Flow balance constraints cannot be free ($b_i^L = -\infty$ and $b_i^U = \infty$ is not allowed).

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

**Minimum-Cost Network Flow for a Simple Directed Graph**

This example demonstrates how to use the network simplex algorithm to find a minimum-cost flow in a directed graph. Consider the directed graph in Figure 1.77, which appears in Ahuja, Magnanti, and Orlin (1993).

![Figure 1.77 Minimum-Cost Network Flow Problem: Data](image)

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

```plaintext
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
;```

The directed graph $G$ can be represented by the following links data set LinkSetIn and nodes data set NodeSetIn:
Chapter 1: The OPTGRAPH Procedure

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

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

    proc optgraph
        loglevel = moderate
        graph_direction = directed
        data_links = LinkSetIn
        data_nodes = NodeSetIn
        out_links = LinkSetOut;
        mincostflow
            logfreq = 1;
    run;

%put &_OPTGRAPH_;
%put &_OPTGRAPH_MCF_;

The progress of the procedure is shown in Figure 1.78.
NOTE: The optimal solution is displayed in Figure 1.79.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Primal Objective</th>
<th>Primal Infeasibility</th>
<th>Dual Infeasibility</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.0000000E+00</td>
<td>2.0000000E+01</td>
<td>8.9000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>0.0000000E+00</td>
<td>2.0000000E+01</td>
<td>8.9000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td>5.0000000E+00</td>
<td>1.5000000E+01</td>
<td>8.4000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>5.0000000E+00</td>
<td>1.5000000E+01</td>
<td>8.3000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>5.0000000E+00</td>
<td>1.5000000E+01</td>
<td>8.3000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>7.5000000E+01</td>
<td>1.5000000E+01</td>
<td>7.9000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>1.3000000E+02</td>
<td>1.0000000E+01</td>
<td>7.6000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>1.3000000E+02</td>
<td>1.0000000E+01</td>
<td>7.6000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>9</td>
<td>1.3000000E+02</td>
<td>1.0000000E+01</td>
<td>7.6000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>10</td>
<td>2.7000000E+02</td>
<td>0.0000000E+00</td>
<td>0.0000000E+00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: Objective = 270.
NOTE: Processing the minimum-cost network flow problem used 0.00 (cpu: 0.00) seconds.
NOTE: Creating links data set output.
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
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 represented graphically in Figure 1.80.

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

Using the same directed graph shown in Figure 1.77, this example demonstrates a network that has a flexible supply and demand. Consider the following adjustments to the node bounds:

- Node 1 has an infinite supply, but it still requires at least 10 units to be sent.
- Node 4 is a throughput node that can now handle an infinite amount of demand.
- Node 8 has a flexible demand. It requires between 6 and 10 units.
You use the special missing values (.I) to represent infinity and (.M) to represent minus infinity. The adjusted node bounds can be represented by the following nodes data set:

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

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

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

The progress of the procedure is shown in Figure 1.81.
Figure 1.81 PROC OPTGRAPH Log for Minimum-Cost Network Flow

NOTE: Running OPTGRAPH version 14.3.
NOTE: The OPTGRAPH procedure is executing in single-machine mode.
NOTE: Reading the nodes data set.
NOTE: There were 5 observations read from the data set WORK.NODESETIN.
NOTE: Reading the links data set.
NOTE: There were 11 observations read from the data set WORK.LINKSETIN.
NOTE: Data input used 0.00 (cpu: 0.01) seconds.
NOTE: Building the input graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The input graph storage is using 0.0 MBs (peak: 0.0 MBs) of memory.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 11.
NOTE: Processing the minimum-cost network flow problem.
NOTE: The network has 1 connected component.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Primal Objective</th>
<th>Primal Infeasibility</th>
<th>Dual Infeasibility</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0000000E+01</td>
<td>2.0000000E+01</td>
<td>1.7300000E+02</td>
<td>0.00</td>
</tr>
<tr>
<td>2</td>
<td>1.0000000E+01</td>
<td>2.0000000E+01</td>
<td>1.7300000E+02</td>
<td>0.00</td>
</tr>
<tr>
<td>3</td>
<td>4.5000000E+01</td>
<td>2.0000000E+01</td>
<td>1.7400000E+02</td>
<td>0.00</td>
</tr>
<tr>
<td>4</td>
<td>4.5000000E+01</td>
<td>2.0000000E+01</td>
<td>1.7400000E+02</td>
<td>0.00</td>
</tr>
<tr>
<td>5</td>
<td>7.5000000E+01</td>
<td>1.5000000E+01</td>
<td>1.6600000E+02</td>
<td>0.00</td>
</tr>
<tr>
<td>6</td>
<td>7.5000000E+01</td>
<td>1.5000000E+01</td>
<td>1.6600000E+02</td>
<td>0.00</td>
</tr>
<tr>
<td>7</td>
<td>1.5900000E+02</td>
<td>9.0000000E+00</td>
<td>8.7000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>8</td>
<td>1.5900000E+02</td>
<td>9.0000000E+00</td>
<td>1.6900000E+02</td>
<td>0.00</td>
</tr>
<tr>
<td>9</td>
<td>2.1400000E+02</td>
<td>4.0000000E+00</td>
<td>8.7000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>10</td>
<td>2.1400000E+02</td>
<td>4.0000000E+00</td>
<td>8.7000000E+01</td>
<td>0.00</td>
</tr>
<tr>
<td>11</td>
<td>2.2600000E+02</td>
<td>0.0000000E+00</td>
<td>0.0000000E+00</td>
<td>0.00</td>
</tr>
<tr>
<td>12</td>
<td>2.2600000E+02</td>
<td>0.0000000E+00</td>
<td>0.0000000E+00</td>
<td>0.00</td>
</tr>
<tr>
<td>13</td>
<td>2.2600000E+02</td>
<td>0.0000000E+00</td>
<td>0.0000000E+00</td>
<td>0.00</td>
</tr>
<tr>
<td>14</td>
<td>2.2600000E+02</td>
<td>0.0000000E+00</td>
<td>0.0000000E+00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

NOTE: The Network Simplex solve time is 0.00 seconds.
NOTE: Objective = 226.
NOTE: Processing the minimum-cost network flow problem used 0.00 (cpu: 0.00) seconds.
NOTE: Creating links data set output.
NOTE: Data output used 0.01 (cpu: 0.00) seconds.

STATUS=OK MCP=OPTIMAL
STATUS=OPTIMAL OBJECTIVE=226 CPU_TIME=0.00 REAL_TIME=0.00
The optimal solution is displayed in Figure 1.82.

**Figure 1.82** Minimum-Cost Network Flow Problem: Optimal Solution

<table>
<thead>
<tr>
<th>Obs</th>
<th>from</th>
<th>to</th>
<th>upper weight</th>
<th>mcf_flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>5</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>7</td>
<td>10</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>6</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>7</td>
<td>15</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>11</td>
<td>7</td>
<td>8</td>
<td>15</td>
<td>9</td>
</tr>
</tbody>
</table>

The optimal solution is represented graphically in Figure 1.83.

**Figure 1.83** Minimum-Cost Network Flow Problem: Optimal Solution

---

**Minimum Cut**

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

In PROC OPTGRAPH, you can invoke the minimum-cut algorithm by using the MINCUT statement. The options for this statement are described in the section “MINCUT Statement” on page 37. This algorithm can be used only on undirected graphs.

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

The minimum-cut algorithm reports status information in a macro variable called _OPTGRAPH_MINCUT_. For more information about this macro variable, see the section “Macro Variable _OPTGRAPH_MINCUT_” on page 182.

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

**Minimum Cut for a Simple Undirected Graph**

As a simple example, consider the weighted undirected graph in Figure 1.84.

![Figure 1.84 A Simple Undirected Graph](image)

The links data set can be represented as follows:

```sas
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:

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

The progress of the procedure is shown in Figure 1.85.

**Figure 1.85** PROC OPTGRAPH Log for Minimum Cut

```
NOTE: -------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
NOTE: Running OPTGRAPH version 14.3.
NOTE: -------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
NOTE: The OPTGRAPH procedure is executing in single-machine mode.
NOTE: -------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
NOTE: Reading the links data set.
NOTE: There were 12 observations read from the data set WORK.LINKSETIN.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: Building the input graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The input graph storage is using 0.0 MBs (peak: 0.0 MBs) of memory.
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 12.
NOTE: -------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
NOTE: Processing the minimum-cut problem.
NOTE: The minimum-cut algorithm found 3 cuts.
NOTE: The cut 1 has weight 4.
NOTE: The cut 2 has weight 5.
NOTE: The cut 3 has weight 5.
NOTE: Processing the minimum-cut problem used 0.00 (cpu: 0.00) seconds.
NOTE: -------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
NOTE: Creating nodes data set output.
NOTE: Creating minimum-cut data set output.
NOTE: Data output used 0.01 (cpu: 0.01) seconds.
NOTE: -------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------
NOTE: The data set WORK.NODESETOUT has 8 observations and 4 variables.
NOTE: The data set WORK.MINCUT has 6 observations and 4 variables.
STATUS=OK  MINCUT=OPTIMAL
STATUS=OPTIMAL  OBJECTIVE=4  CPU_TIME=0.00  REAL_TIME=0.00
```
The data set NodeSetOut now contains the partition of the nodes for each cut, shown in Figure 1.86.

**Figure 1.86 Minimum Cut Node Partition**

<table>
<thead>
<tr>
<th>node</th>
<th>mincut_1</th>
<th>mincut_2</th>
<th>mincut_3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

The data set MinCut contains the links in the cut-sets for each cut. This data set is shown in Figure 1.87, which also shows each cut separately.

**Figure 1.87 Minimum Cut Sets**

<table>
<thead>
<tr>
<th>mincut</th>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

mincut=1

from to weight

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

mincut=4

mincut=2

from to weight

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>3</td>
</tr>
</tbody>
</table>

mincut=5

mincut=3

from to weight

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
</tbody>
</table>

mincut=5

14
Minimum Spanning Tree

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

In PROC OPTGRAPH, you can invoke the minimum spanning tree algorithm by using the MINSPANTREE statement. The options for this statement are described in the section “MINSPANTREE Statement” on page 38. This algorithm can be used only on undirected graphs.

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

The minimum spanning tree algorithm reports status information in a macro variable called _OPTGRAPH_MST_. For more information about this macro variable, see the section “Macro Variable _OPTGRAPH_MST_” on page 182.

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

Minimum Spanning Tree for a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 1.88.

Figure 1.88 A Simple Undirected Graph

The links data set can be represented as follows:

```plaintext
data LinkSetIn;
  input from $ to $ weight @@;
datalines;
  A B 7  A D 5  B C 8  B D 9  B E 7  C E 5  D E 15  D F 6  E F 8  E G 9  F G 11  H I 1  I J 3  H J 2
;
```
The following statements calculate a minimum spanning forest and output the results in the data set MinSpanForest:

```sas
proc optgraph
  data_links = LinkSetIn;
  minspantree
    out = MinSpanForest;
run;
```

The data set MinSpanForest now contains the links that belong to a minimum spanning forest, which is shown in Figure 1.89.

**Figure 1.89** Minimum Spanning Forest

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>I</td>
<td>1</td>
</tr>
<tr>
<td>H</td>
<td>J</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
<td>5</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td>6</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>7</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>7</td>
</tr>
<tr>
<td>E</td>
<td>G</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td></td>
<td>42</td>
</tr>
</tbody>
</table>

The minimal cost links are shown in green in Figure 1.90.

**Figure 1.90** Minimum Spanning Forest

For a more detailed example, see “Example 1.12: Minimum Spanning Tree for Computer Network Topology” on page 223.
Reach (Ego) Network

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

In PROC OPTGRAPH, reach networks can be calculated by using the REACH statement. The options for this statement are described in the section “REACH Statement” on page 39.

The REACH statement reports status information in a macro variable called _OPTGRAPH_REACH_. For more information about this macro variable, see the section “Macro Variable _OPTGRAPH_REACH_” on page 182.

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

Output Data Sets

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

**OUT NODES= Data Set**

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

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

**OUT LINKS= Data Set**

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

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

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

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

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

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

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

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

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

This section illustrates the use of the reach networks algorithm on the simple directed graph $G$ that is shown in Figure 1.91.

**Figure 1.91** Simple Directed Graph $G$

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

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

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

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

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

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

``` SAS
proc optgraph
  graph_direction = directed
  data_links = LinkSetIn
  data_nodes_sub = NodeSubSetIn1;
reach
  out_nodes = ReachNodes1
```

out_links  = ReachLinks1
out_counts = ReachCounts1
maxreach   = 1;
run;

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

**Figure 1.92** Reach Network for $S_1 = \{A, G\}$ with Hop Limit of 1

**ReachNodes1**

<table>
<thead>
<tr>
<th>reach</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>G</td>
</tr>
<tr>
<td>1</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>I</td>
</tr>
</tbody>
</table>

**ReachLinks1**

<table>
<thead>
<tr>
<th>reach</th>
<th>from</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>1</td>
<td>H</td>
<td>G</td>
</tr>
<tr>
<td>1</td>
<td>G</td>
<td>I</td>
</tr>
<tr>
<td>1</td>
<td>H</td>
<td>I</td>
</tr>
</tbody>
</table>

**ReachCounts1**

<table>
<thead>
<tr>
<th>reach</th>
<th>node</th>
<th>count</th>
<th>count_not</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>7</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>G</td>
<td>7</td>
<td>2</td>
</tr>
</tbody>
</table>

The results are displayed graphically in **Figure 1.93**.
For the second set of source nodes, you can use the following statements to calculate the reach network with a hop limit of 2:

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

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

**Figure 1.94** Reach Network for $S_2 = \{ B \}$ with Hop Limit of 2

**ReachNodes2**

<table>
<thead>
<tr>
<th>reach node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 B</td>
</tr>
<tr>
<td>1 C</td>
</tr>
<tr>
<td>1 D</td>
</tr>
<tr>
<td>1 E</td>
</tr>
<tr>
<td>1 F</td>
</tr>
<tr>
<td>1 G</td>
</tr>
</tbody>
</table>
The results are displayed graphically in Figure 1.95.

**Figure 1.95** Reach Network for $S_1 = \{B\}$ with Hop Limit of 2

Processing Multiple Reach Networks in One Pass

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

Consider again the graph shown in Figure 1.91, now with source node sets $S_1 = \{C\}$ and $S_2 = \{A, H\}$. These source node sets can be defined together as follows:
data NodeSubSetIn;
  input node $ reach;
datalines;
  A 2
  C 1
  H 2;
;
You can use the following statements to process the two one-hop-limit reach networks in one pass:

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

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

**Figure 1.96** Reach Networks for $S_1 = \{C\}$ and $S_2 = \{A, H\}$ with Hop Limit of 1

<table>
<thead>
<tr>
<th>ReachNodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>reach</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ReachLinks</th>
</tr>
</thead>
<tbody>
<tr>
<td>reach</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>
Similar to the usage for centrality described in the section “Processing by Cluster” on page 79, you can use the BY_CLUSTER option in the REACH statement to process a number of induced subgraphs of a graph with only one call to PROC OPTGRAPH. In this section, you want to work on the subgraphs that are induced by node subsets \( N_0 = \{A, C, D, E\} \) and \( N_1 = \{B, F, G, H, I\} \) for the directed graph shown in Figure 1.91. The induced subgraphs are shown graphically in Figure 1.97 and Figure 1.98.

**Figure 1.97** Induced Subgraph for \( N^0 = \{A, C, D, E\} \)

**Figure 1.98** Induced Subgraph for \( N^1 = \{B, F, G, H, I\} \)
Define the subgraphs in the nodes data set by using the cluster variable as follows:

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

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

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

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

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

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

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

**Figure 1.99** Reach Networks for $S = \{B, C\}$ with Hop Limit of 2 for Induced Subgraphs

<table>
<thead>
<tr>
<th>reach</th>
<th>node</th>
<th>cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>E</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>G</td>
<td>1</td>
</tr>
</tbody>
</table>
Notice that since you are operating on the induced subgraphs (not the original graph), node B cannot reach nodes C and E because they are not in its induced subgraph.

**Processing Multiple Reach Networks in One Pass by Cluster**

You can also process several reach networks in one pass while looking over decomposed subgraphs. Consider the same original graph and subgraphs from the section “Processing Reach Networks by Cluster” on page 138. Now, suppose you want the one-hop-limit reach network where each original node is its own source node subset. Define nine source sets by using the node subset data set as follows:

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

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

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

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

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

The resulting data sets ReachNodes and ReachCounts are displayed in Figure 1.100.

Figure 1.100 Reach Networks for Each Source for Induced Subgraphs with a Node Hop Limit of 1

### ReachNodes

<table>
<thead>
<tr>
<th>reach</th>
<th>node</th>
<th>cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 A</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1 C</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1 D</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2 B</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>2 F</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>3 C</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>3 E</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4 D</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>4 E</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>5 D</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>5 E</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>6 F</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>6 G</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>7 G</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>7 H</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>7 I</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>8 G</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>8 H</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>8 I</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>9 I</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

### ReachCounts

<table>
<thead>
<tr>
<th>reach</th>
<th>node</th>
<th>count</th>
<th>count_not</th>
<th>count_in</th>
<th>count_out</th>
<th>count_in_or_out</th>
<th>count_in_and_out</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 A</td>
<td></td>
<td>0</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2 B</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3 C</td>
<td></td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4 D</td>
<td></td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>5 E</td>
<td></td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>6 F</td>
<td></td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>7 G</td>
<td></td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>8 H</td>
<td></td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>9 I</td>
<td></td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Processing Each Source Reach Network for Hop Limits of Both 1 and 2 in One Pass by Cluster

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

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

The resulting data sets ReachCounts1 and ReachCounts1 are displayed in Figure 1.101.

**Figure 1.101**  Reach Counts for Each Source Node for Induced Subgraphs with a Hop Limit of 1 and 2

<table>
<thead>
<tr>
<th>ReachCounts1</th>
<th>ReachCounts2</th>
</tr>
</thead>
<tbody>
<tr>
<td>reach node cluster count count_not</td>
<td>reach node cluster count count_not</td>
</tr>
<tr>
<td>1 A 0 3 1</td>
<td>1 A 0 4 0</td>
</tr>
<tr>
<td>3 C 0 2 2</td>
<td>3 C 0 3 1</td>
</tr>
<tr>
<td>4 D 0 2 2</td>
<td>4 D 0 2 2</td>
</tr>
<tr>
<td>5 E 0 2 2</td>
<td>5 E 0 2 2</td>
</tr>
<tr>
<td>2 B 1 2 3</td>
<td>2 B 1 3 2</td>
</tr>
<tr>
<td>6 F 1 2 3</td>
<td>6 F 1 4 1</td>
</tr>
<tr>
<td>7 G 1 3 2</td>
<td>7 G 1 3 2</td>
</tr>
<tr>
<td>8 H 1 3 2</td>
<td>8 H 1 3 2</td>
</tr>
<tr>
<td>9 I 1 1 4</td>
<td>9 I 1 1 4</td>
</tr>
</tbody>
</table>

For a more detailed example, see “Example 1.14: Reach Networks for Computation of Market Coverage of a Terrorist Network” on page 227.
**Shortest Path**

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

In PROC OPTGRAPH, you can calculate shortest paths by using the SHORTPATH statement. The options for this statement are described in the section “SHORTPATH Statement” on page 41.

The shortest path algorithm reports status information in a macro variable called _OPTGRAPH_SHORTPATH_. For more information about this macro variable, see the section “Macro Variable _OPTGRAPH_SHORTPATH_” on page 183.

By default, PROC OPTGRAPH finds shortest paths for all pairs. That is, it finds a shortest path for each possible combination of source and sink nodes. Alternatively, you can use the SOURCE= option to fix a particular source node and find shortest paths from the fixed source node to all possible sink nodes. Conversely, by using the SINK= option, you can fix a sink node and find shortest paths from all possible source nodes to the fixed sink node. By using both options together, you can request one particular shortest path for a specific source-sink pair. In addition, you can use the DATA NODES SUB= option to define a list of source-sink pairs to process, as described in the section “Node Subset Input Data” on page 55. The following sections show examples of these options.

Which algorithm PROC OPTGRAPH uses to find shortest paths depends on the data. The algorithm and run-time complexity for each link type are shown in Table 1.58.

<table>
<thead>
<tr>
<th>Link Type</th>
<th>Algorithm</th>
<th>Complexity (per Source Node)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unweighted</td>
<td>Breadth-first search</td>
<td>( O(</td>
</tr>
<tr>
<td>Weighted (nonnegative)</td>
<td>Dijkstra’s algorithm</td>
<td>( O(</td>
</tr>
<tr>
<td>Weighted (positive and negative allowed)</td>
<td>Bellman-Ford algorithm</td>
<td>( O(</td>
</tr>
</tbody>
</table>

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

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

**Output Data Sets**

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

**OUT_PATHS= Data Set**

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

The OUT_PATHS= data set contains the following columns:

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

**OUT_WEIGHTS= Data Set**

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

This data set contains the following columns:

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

**Shortest Paths for All Pairs**

This example illustrates the use of the shortest path algorithm for all source-sink pairs on the simple undirected graph $G$ shown in Figure 1.102.
The undirected graph $G$ can be represented by the following links data set, LinkSetIn:

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

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

```
proc optgraph
  data_data_links = LinkSetIn;
  shortpath
    out_weights = ShortPathW
    out_paths = ShortPathP;
run;
```

The data set ShortPathP contains the shortest paths and is shown in Figure 1.103.
The data set ShortPathW contains the path weight for the shortest paths of each source-sink pair and is shown in Figure 1.104.

### Figure 1.103  All-Pairs Shortest Paths

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>order</th>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>1</td>
<td>A</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>2</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>3</td>
<td>E</td>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
<td>2</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>2</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>3</td>
<td>E</td>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>4</td>
<td>D</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>1</td>
<td>B</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>1</td>
<td>B</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>2</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>1</td>
<td>B</td>
<td>D</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>1</td>
<td>B</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>2</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>3</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>F</td>
<td>1</td>
<td>B</td>
<td>F</td>
<td>5</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>1</td>
<td>C</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>1</td>
<td>C</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>2</td>
<td>A</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>1</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>2</td>
<td>E</td>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
<td>1</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>1</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>2</td>
<td>E</td>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>3</td>
<td>D</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>1</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>2</td>
<td>E</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>3</td>
<td>C</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
<td>1</td>
<td>D</td>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
<td>1</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
<td>2</td>
<td>E</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
<td>1</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td>1</td>
<td>D</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>1</td>
<td>E</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>2</td>
<td>C</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>1</td>
<td>E</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>2</td>
<td>C</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>C</td>
<td>3</td>
<td>E</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
<td>1</td>
<td>E</td>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>1</td>
<td>E</td>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>2</td>
<td>D</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>A</td>
<td>1</td>
<td>F</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>A</td>
<td>2</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>A</td>
<td>3</td>
<td>E</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>A</td>
<td>4</td>
<td>C</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>B</td>
<td>1</td>
<td>F</td>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>F</td>
<td>C</td>
<td>1</td>
<td>F</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>C</td>
<td>2</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>C</td>
<td>3</td>
<td>E</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>1</td>
<td>F</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>E</td>
<td>1</td>
<td>F</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>E</td>
<td>2</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
</tbody>
</table>
When you are interested only in the source-sink pair that has the longest shortest path, you can use the PATHS= option. This option affects only the output processing; it does not affect the computation. All the designated source-sink shortest paths are calculated, but only the longest ones are written to the output data set.

The following statements display only the longest shortest paths:

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

The data set `ShortPathLong` now contains the longest shortest paths and is shown in Figure 1.105.

Figure 1.105 Longest Shortest Paths

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>order</th>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>F</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>2</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>3</td>
<td>E</td>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>4</td>
<td>D</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>1</td>
<td>B</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>2</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>3</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>1</td>
<td>E</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>2</td>
<td>C</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>3</td>
<td>A</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>A</td>
<td>1</td>
<td>F</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>A</td>
<td>2</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>A</td>
<td>3</td>
<td>E</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>A</td>
<td>4</td>
<td>C</td>
<td>A</td>
<td>2</td>
</tr>
</tbody>
</table>
Shortest Paths for a Subset of Source-Sink Pairs

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

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

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

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

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

The data set ShortPath contains the shortest paths and is shown in Figure 1.106.

![Figure 1.106](image)

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>order</th>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>1</td>
<td>A</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>2</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>3</td>
<td>E</td>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>4</td>
<td>D</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>1</td>
<td>C</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>2</td>
<td>A</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>1</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>2</td>
<td>E</td>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>3</td>
<td>D</td>
<td>F</td>
<td>1</td>
</tr>
</tbody>
</table>

Shortest Paths for a Subset of Source or Sink Pairs

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

In this case, you designate the subset of source (or sink) nodes in the node subset data set by specifying the source (or sink). By specifying only one of the variables, you indicate that you want PROC OPTGRAPH to calculate all pairs from a subset of source nodes (or to calculate all pairs to a subset of sink nodes).
For example, the following DATA step designates nodes $B$ and $E$ as source nodes:

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

You can use the same PROC OPTGRAPH call as is used in the section “Shortest Paths for a Subset of Source-Sink Pairs” on page 148 to calculate all the shortest paths from nodes $B$ and $E$. The data set ShortPath contains the shortest paths and is shown in Figure 1.107.

**Figure 1.107** Shortest Paths for a Subset of Source Pairs

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>order</th>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>A</td>
<td>1</td>
<td>B</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>1</td>
<td>B</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>2</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>1</td>
<td>B</td>
<td>D</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>1</td>
<td>B</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>2</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>3</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>F</td>
<td>1</td>
<td>B</td>
<td>F</td>
<td>5</td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>1</td>
<td>E</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>2</td>
<td>C</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>1</td>
<td>E</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>2</td>
<td>C</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>3</td>
<td>A</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>C</td>
<td>1</td>
<td>E</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
<td>1</td>
<td>E</td>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>1</td>
<td>E</td>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>2</td>
<td>D</td>
<td>F</td>
<td>1</td>
</tr>
</tbody>
</table>

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

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

You can use the same PROC OPTGRAPH call again to calculate all the shortest paths to nodes $B$ and $E$. The data set ShortPath contains the shortest paths and is shown in Figure 1.108.
Chapter 1: The OPTGRAPH Procedure

Figure 1.108  Shortest Paths for a Subset of Sink Pairs

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>order</th>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>1</td>
<td>A</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
<td>1</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
<td>2</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>1</td>
<td>B</td>
<td>A</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>2</td>
<td>A</td>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>3</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>1</td>
<td>C</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>2</td>
<td>A</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
<td>1</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
<td>1</td>
<td>D</td>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
<td>1</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>1</td>
<td>E</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>2</td>
<td>C</td>
<td>A</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>3</td>
<td>A</td>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>B</td>
<td>1</td>
<td>F</td>
<td>B</td>
<td>5</td>
</tr>
<tr>
<td>F</td>
<td>E</td>
<td>1</td>
<td>F</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>E</td>
<td>2</td>
<td>D</td>
<td>E</td>
<td>2</td>
</tr>
</tbody>
</table>

Shortest Paths for One Source-Sink Pair

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

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

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

The data set ShortPath contains this shortest path and is shown in Figure 1.109.

Figure 1.109  Shortest Paths for One Source-Sink Pair

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>order</th>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>F</td>
<td>1</td>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>2</td>
<td>E</td>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>3</td>
<td>D</td>
<td>F</td>
<td>1</td>
</tr>
</tbody>
</table>

The shortest path is shown graphically in Figure 1.110.
Shortest Paths with Auxiliary Weight Calculation

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

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

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

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

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

The data set ShortPathW contains the total path weight for shortest paths in each source-sink pair and is shown in Figure 1.111. Because the variable count in LinkSetIn has a value of 1 for all links, the value in the output data set variable path_weights2 contains the number of links in each shortest path.
The section “Road Network Shortest Path” on page 4 shows an example of using the shortest path algorithm to minimize travel to and from work based on traffic conditions.

**Shortest Paths with Negative Link Weights**

This section illustrates the use of the shortest path algorithm on a simple directed graph $G$ with negative link weights, shown in Figure 1.112.

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>path_weight</th>
<th>path_weight2</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>F</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>path_weight</th>
<th>path_weight2</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>A</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>C</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>A</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>F</td>
<td>B</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>C</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>E</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
You can represent the directed graph $G$ by using the following links data set `LinkSetIn`:

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

The following statements calculate the shortest paths between source node $E$ and sink node $B$:

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

The data set `ShortPathP` contains the shortest path from node $E$ to node $B$ and is shown in Figure 1.113.

### Figure 1.113 Shortest Paths with Negative Link Weights

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>order</th>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>B</td>
<td>1</td>
<td>E</td>
<td>D</td>
<td>-3</td>
</tr>
<tr>
<td>E</td>
<td>B</td>
<td>2</td>
<td>D</td>
<td>B</td>
<td>1</td>
</tr>
</tbody>
</table>
Now, consider the following adjustment to the weight of link $(B, E)$:

```sas
data LinkSetIn;
  set LinkSetIn;
  if(from="B" and to="E") then
    weight=1;
run;
```

In this case, there is a negative weight cycle $(E \to D \to B \to E)$. The Bellman-Ford algorithm catches this and produces an error, as shown in Figure 1.114.

![Figure 1.114 PROC OPTGRAPH Log: Negative Weight Cycle](image-url)

**Summary**

In PROC OPTGRAPH, you can calculate various summary statistics for a graph and its nodes by using the SUMMARY statement. The options for this statement are described in the section “SUMMARY Statement” on page 42.

The SUMMARY statement reports status information in a macro variable called _OPTGRAPH_SUMMARY_. For more information about this macro variable, see the section “Macro Variable _OPTGRAPH_SUMMARY_” on page 183.

**Output Data Sets**

The summary statistics that are produced are broken into two categories: statistics on the entire graph and statistics on the nodes and links of the graph. The latter statistics are appended to the output nodes and links data sets that you specify in the OUT_NODES= and OUT_LINKS= option in the PROC OPTGRAPH statement. The former statistics are contained in the data set that you specify in the OUT= option in the SUMMARY statement.
Let $\delta(u)$ represent the list of nodes that are connected to node $u$ in an undirected graph. In a directed graph, $\delta_{\text{out}}(u)$ represents the list of nodes that are connected from node $u$ (out-links), and $\delta_{\text{in}}(u)$ represents the list of nodes that are connected to node $u$ (in-links).

**OUT= Data Set**

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

- **nodes**: the number of nodes in the graph ($|N|$)
- **links**: the number of links in the graph ($|A|$)
- **avg_links_per_node**: the average number of links per node
- **density**: the number of links in the graph divided by the number of links in a complete graph $\left(\frac{|A|}{K(N)}\right)$
- **self_links_ignored**: the number of self-links ignored
- **dup_links_ignored**: the number of duplicate links ignored
- **leaf_nodes**: the number of leaf nodes
  - Undirected graph: $u \in N$ such that $\delta(u) = 1$
  - Directed graph: $u \in N$ such that $\delta_{\text{out}}(u) = 0$ and $\delta_{\text{out}}(u) + \delta_{\text{in}}(u) > 0$
- **singleton_nodes**: the number of singleton nodes
  - Undirected graph: $u \in N$ such that $\delta(u) = 0$
  - Directed graph: $u \in N$ such that $\delta_{\text{out}}(u) + \delta_{\text{in}}(u) = 0$

You can produce statistics about the connectedness of the graph by using the CONCOMP and BICONCOMP options. For more information about connected components and biconnected components, see the sections “Connected Components” on page 99 and “Biconnected Components and Articulation Points” on page 62, respectively. If you use the CONCOMP and BICONCOMP options, the following columns also appear in the summary output data set for undirected graphs:

- **concomp**: the number of connected components in the graph
- **biconcomp**: the number of biconnected components in the graph
- **artpoints**: the number of articulation points in the graph
- **isolated_pairs**: the number of isolated pairs of nodes (a connected component of size 2)
- **isolated_stars**: the number of isolated stars, (a connected component, $C$, of size greater than 2 with):
  - one node $i$ with $\delta(i) = |C| - 1$ and all other nodes $u \in C \setminus \{i\}$ with $\delta(u) = 1$

The following columns appear for directed graphs:

- **concomp**: the number of strongly connected components in the graph
• isolated_pairs: the number of isolated pairs of nodes (a weakly connected component of size 2)

• isolated_stars_out: the number of isolated outward stars (a weakly connected component, \( C \), of size greater than 2 with):
  – one node \( i \) with \( \delta^\text{out}(i) = |C| - 1 \) and all other nodes \( u \in C \setminus \{i\} \) with \( \delta^\text{in}(u) = 1 \)

• isolated_stars_in: the number of isolated inward stars (a weakly connected component, \( C \), of size greater than 2 with):
  – one node \( i \) with \( \delta^\text{in}(i) = |C| - 1 \) and all other nodes \( u \in C \setminus \{i\} \) with \( \delta^\text{out}(u) = 1 \)

You can produce statistics about the shortest paths in the graph by using the SHORTPATH= option. The \textit{diameter} of a graph is the longest shortest path distance of all possible source-sink pairs in the graph. Calculating the diameter of a graph is computationally expensive, because it involves calculating shortest paths for all pairs. For undirected graphs, an approximate method is available based on Boitmanis et al. (2006). You can invoke the algorithm by using the DIAMETER_APPROX= option. The exact method runs in time \( O(|N| \times (|N| \log |N| + |A|)) \); the approximate method runs in time \( O(|A| \sqrt{|N|}) \) with an additive error of \( O(\sqrt{|N|}) \). For more information about shortest paths, see the section “Shortest Path” on page 143.

If you use the SHORTPATH= option, the following columns also appear in the summary output data set:

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

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

\textit{OUT_NODES=} Data Set

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

• sum_in_and_out_wt: sum of the link weights from and to the node
• leaf_node: 1, if the node is a leaf node; otherwise, 0
• singleton_node: 1, if the node is a singleton node; otherwise, 0
• isolated_pair: the identifier, if the node is in an isolated pair; otherwise, missing (.)
• neighbor_leaf_nodes: the number of leaf nodes connected to the node
In addition, the following column is appended for undirected graphs:

- *isolated_star*: the identifier, if the node is in an isolated star; otherwise, missing (.)

The following columns are appended for directed graphs:

- *isolated_star_out*: the identifier, if the node is in an isolated outward star; otherwise, missing (.)
- *isolated_star_in*: the identifier, if the node is in an isolated inward star; otherwise, missing (.)

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

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

**OUT_LINKS= Data Set**

In addition, you can produce summary statistics about the links of the graph. By default, the following columns are appended to the data set that you specify in the OUT_LINKS= option in the PROC OPTGRAPH statement, for undirected graphs:

- *isolated_pair*: the identifier, if the link is in an isolated pair; otherwise, missing (.)
- *isolated_star*: the identifier, if the link is in an isolated star; otherwise, missing (.)

The following columns are appended for directed graphs:

- *isolated_star_out*: the identifier, if the link is in an isolated outward star; otherwise, missing (.)
- *isolated_star_in*: the identifier, if the link is in an isolated inward star; otherwise, missing (.)

**Summary Statistics of a Simple Directed Graph**

This section illustrates the calculation of summary statistics on the simple directed graph *G* that is shown in Figure 1.115.
You can represent the directed graph $G$ by using the following nodes data set, NodeSetIn, and links data set, LinkSetIn:

```
data NodeSetIn;
  input node $ @@;
datalines;
  A B C D E F G H I J K L M N O P
;
data LinkSetIn;
  input from $ to $ weight @@;
datalines;
  A B 1 A C 2 A D 2 B A 2 D E 2
  D F 1 E F 2 F D 2 F E 1 A A 2
  A B 2 I J 5 K L 3 K M 2 N O 1
  P O 1
;
```

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

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

The data set Summary contains the default summary statistics of the input graph and is shown in Figure 1.116.
The following statements calculate the default summary statistics and information about the connectedness of the graph, and they output the results in the data set `Summary`:

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

The data set `Summary` contains the summary statistics of the input graph and is shown in Figure 1.117.

**Figure 1.117** Graph Summary and Connectedness Statistics of a Simple Directed Graph

<table>
<thead>
<tr>
<th>nodes</th>
<th>links</th>
<th>avg_links_per_node</th>
<th>density</th>
<th>self_links_ignored</th>
<th>dup_links_ignored</th>
<th>leaf_nodes</th>
<th>singleton_nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>14</td>
<td>0.875</td>
<td>0.058333</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

**Summary Statistics of a Simple Directed Graph by Cluster**

Similar to how you can use the `BY_CLUSTER` option in the `CENTRALITY` statement, as described in the section “Processing by Cluster” on page 79, you can process a number of induced subgraphs of a graph with only one call to PROC OPTGRAPH by using the `BY_CLUSTER` option in the `SUMMARY` statement. In this section, you want to work on the subgraphs that are induced by node subsets $N_0 = \{A, B, C\}$, $N_1 = \{D, E, F\}$, and $N_3 = \{G, H, I, J, K, L, M, N, O, P\}$ for the directed graph shown in Figure 1.115. The induced subgraphs are shown graphically in Figure 1.118 (the dashed link is removed).
Define the subgraphs in the nodes data set by using the `cluster` variable as follows:

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

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

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

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

The data sets `Summary`, `NodeSetOut`, and `LinkSetOut` now contain the summary statistics for each induced subgraph; they are shown in Figure 1.119.
Figure 1.119 Summary Statistics for Induced Subgraphs of Graph G

### Summary

<table>
<thead>
<tr>
<th>cluster</th>
<th>nodes</th>
<th>links</th>
<th>avg_links_per_node</th>
<th>density</th>
<th>self_links_ignored</th>
<th>dup_links_ignored</th>
<th>leaf_nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>3</td>
<td>1.00000</td>
<td>0.50000</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>1.66667</td>
<td>0.83333</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>5</td>
<td>0.50000</td>
<td>0.05556</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>singleton_nodes</th>
<th>concomp</th>
<th>isolated_pairs</th>
<th>isolated_stars_out</th>
<th>isolated_stars_in</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### NodeSetOut

<table>
<thead>
<tr>
<th>node</th>
<th>cluster</th>
<th>sum_in_and_out_wt</th>
<th>leaf_node</th>
<th>singleton_node</th>
<th>isolated_pair</th>
<th>isolated_star_out</th>
<th>isolated_star_in</th>
<th>neighbor_leaf_nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>0</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>1</td>
</tr>
<tr>
<td>J</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>0</td>
</tr>
<tr>
<td>K</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>L</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>1</td>
</tr>
<tr>
<td>M</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>.</td>
<td>0</td>
</tr>
<tr>
<td>N</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>O</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>.</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>.</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### LinkSetOut

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>cluster</th>
<th>weight</th>
<th>isolated_pair</th>
<th>isolated_star_out</th>
<th>isolated_star_in</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>0</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>0</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>0</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
<td>1</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td>1</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>1</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>1</td>
<td>2</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>F</td>
<td>E</td>
<td>1</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>I</td>
<td>J</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>.</td>
<td>.</td>
</tr>
<tr>
<td>K</td>
<td>L</td>
<td>2</td>
<td>3</td>
<td>.</td>
<td>1</td>
<td>.</td>
</tr>
<tr>
<td>K</td>
<td>M</td>
<td>2</td>
<td>2</td>
<td>.</td>
<td>1</td>
<td>.</td>
</tr>
<tr>
<td>N</td>
<td>O</td>
<td>2</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>1</td>
</tr>
<tr>
<td>P</td>
<td>O</td>
<td>2</td>
<td>1</td>
<td>.</td>
<td>.</td>
<td>1</td>
</tr>
</tbody>
</table>
Summary Statistics of a Simple Undirected Graph

This section illustrates the calculation of summary and shortest path statistics on the simple undirected graph $G$ that is shown in Figure 1.120.

Figure 1.120  A Simple Undirected Graph $G$

You can represent the undirected graph $G$ by using the following links data set, LinkSetIn:

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

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

```plaintext
test proc optgraph
  data_links = LinkSetIn
  out_nodes = NodeSetOut;
  summary
    out = Summary
    shortpath = weight;
run;
```

The data sets Summary and NodeSetOut now contain the summary statistics of the input graph, which are shown in Figure 1.121.
The transitive closure of a graph $G$ is a graph $G^T = (N, A^T)$ such that for all $i, j \in N$ there is a link $(i, j) \in A^T$ if and only if there exists a path from $i$ to $j$ in $G$.

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

In PROC OPTGRAPH, you can invoke the transitive closure algorithm by using the TRANSITIVE_CLOSURE statement. The options for this statement are described in the section “TRANSITIVE_CLOSURE Statement” on page 44.

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

The transitive closure algorithm reports status information in a macro variable called _OPTGRAPH_TRANSCL_. For more information about this macro variable, see the section “Macro Variable _OPTGRAPH_TRANSCL_” on page 184.

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

**Transitive Closure of a Simple Directed Graph**

This example illustrates the use of the transitive closure algorithm on the simple directed graph $G$, which is shown in Figure 1.122.
The directed graph $G$ can be represented by the following links data set LinkSetIn:

```plaintext
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:

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

The data set TransClosure contains the transitive closure of $G$ and is shown in **Figure 1.123**.

**Figure 1.123**  Transitive Closure of a Simple Directed Graph

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
</tbody>
</table>

The transitive closure of $G$ is shown graphically in **Figure 1.124**.
Traveling Salesman Problem

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

\[
\begin{align*}
\text{minimize} & \quad \sum_{(i,j) \in A} c_{ij} x_{ij} \\
\text{subject to} & \quad \sum_{(i,j) \in \delta(i)} x_{ij} = 2 \quad i \in N \quad \text{(two_match)} \\
& \quad \sum_{(i,j) \in \delta(S)} x_{ij} \geq 2 \quad S \subseteq N, \ 2 \leq |S| \leq |N| - 1 \quad \text{(subtour elim)} \\
& \quad x_{ij} \in \{0, 1\} \quad (i,j) \in A
\end{align*}
\]

The equations (two_match) are the matching constraints, which ensure that each node has degree two in the subgraph. The inequalities (subtour elim) are the subtour elimination constraints (SECs), which enforce connectivity.
For a directed graph, $G$, the same formulation and solution approach is used on an expanded graph $G'$, as described in Kumar and Li (1994). PROC OPTGRAPH takes care of the construction of the expanded graph and returns the solution in terms of the original input graph.

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

In PROC OPTGRAPH, you can invoke the traveling salesman problem solver by using the TSP statement. The options for this statement are described in the section “TSP Statement” on page 45.

The traveling salesman problem solver reports status information in a macro variable called _OPTGRAPH_TSP_. For more information about this macro variable, see the section “Macro Variable _OPTGRAPH_TSP_” on page 184.

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

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

### Traveling Salesman Problem Applied to a Simple Undirected Graph

As a simple example, consider the weighted undirected graph in Figure 1.125.

**Figure 1.125** A Simple Undirected Graph
You can represent the links data set as follows:

```javascript
data LinkSetIn;
  input from $ to $ weight @0;
  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:

```javascript
proc optgraph
  loglevel = moderate
  data_links = LinkSetIn
  out_nodes = NodeSetOut;
  tsp
    out = TSPTour;
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_TSP_;
```

The progress of the OPTGRAPH procedure is shown in Figure 1.126.
### Figure 1.126 PROC OPTGRAPH Log: Optimal Traveling Salesman Tour of a Simple Undirected Graph

- Running OPTGRAPH version 14.3.
- The OPTGRAPH procedure is executing in single-machine mode.
- Reading the links data set.
- There were 22 observations read from the data set WORK.LINKSETIN.
- Data input used 0.01 (cpu: 0.00) seconds.
- Building the input graph storage used 0.00 (cpu: 0.00) seconds.
- The input graph storage is using 0.0 MBs (peak: 0.0 MBs) of memory.
- The number of nodes in the input graph is 10.
- The number of links in the input graph is 22.
- Processing the traveling salesman problem.
- The initial TSP heuristics found a tour with cost 16 using 0.01 (cpu: 0.00) seconds.
- The MILP presolver value NONE is applied.
- The MILP solver is called.
- The Branch and Cut algorithm is used.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>16.00000000</td>
<td>15.5005000</td>
<td>3.22%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>16.00000000</td>
<td>16.0000000</td>
<td>0.00%</td>
<td>0</td>
</tr>
</tbody>
</table>

- Objective = 16.
- Processing the traveling salesman problem used 0.02 (cpu: 0.00) seconds.
- Creating nodes data set output.
- Creating traveling salesman data set output.
- Data output used 0.01 (cpu: 0.00) seconds.
- The data set WORK.NODESETOUT has 10 observations and 2 variables.
- The data set WORK.TSPTOUR has 10 observations and 3 variables.

STATUS=OK  TSP=OPTIMAL
STATUS=OPTIMAL  OBJECTIVE=16  RELATIVE_GAP=0  ABSOLUTE_GAP=0  PRIMAL_INFEASIBILITY=0
BOUND_INFEASIBILITY=0  INTEGER_INFEASIBILITY=0  BEST_BOUND=16  NODES=1  ITERATIONS=16
CPU_TIME=0.00  REAL_TIME=0.02

The data set NodeSetOut now contains a sequence of nodes in the optimal tour and is shown in Figure 1.127.
The data set TSPTour now contains the links in the optimal tour and is shown in Figure 1.128.

The minimum-cost links are shown in green in Figure 1.129.
Traveling Salesman Problem Applied to a Simple Directed Graph

As another simple example, consider the weighted directed graph in Figure 1.130.

Figure 1.130  A Simple Directed Graph

You can represent the links data set as follows:

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

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

```plaintext
proc optgraph
    direction = directed
    loglevel = moderate
    data_links = LinkSetIn
    out_nodes = NodeSetOut;
    tsp
        out = TSPTour;
run;
%put &_OPTGRAPH_; %put &_OPTGRAPH_TSP_;```

The progress of the OPTGRAPH procedure is shown in Figure 1.131.
The data set NodeSetOut now contains a sequence of nodes in the optimal tour and is shown in Figure 1.132.
Figure 1.132  Nodes in the Optimal Traveling Salesman Tour

<table>
<thead>
<tr>
<th>node</th>
<th>tsp_order</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>3</td>
</tr>
<tr>
<td>E</td>
<td>4</td>
</tr>
<tr>
<td>D</td>
<td>5</td>
</tr>
</tbody>
</table>

The data set TSPTour now contains the links in the optimal tour and is shown in Figure 1.133.

Figure 1.133  Links in the Optimal Traveling Salesman Tour

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>1</td>
</tr>
</tbody>
</table>

The minimum-cost links are shown in green in Figure 1.134.

Figure 1.134  Optimal Traveling Salesman Tour
Macro Variables

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

Macro Variable _OPTGRAPH_

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

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

- **OK**  The procedure terminated normally.
- **OUT_OF_MEMORY**  Insufficient memory was allocated to the procedure.
- **INTERRUPTED**  The procedure was interrupted by the user during the input or setup phase.
- **ERROR**  The procedure encountered an error.

**BICONCOMP**
indicates the status of the biconnected components algorithm at termination. This algorithm is described in the section “Biconnected Components and Articulation Points” on page 62. The BICONCOMP term can take one of the following values:

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

**CENTRALITY**
indicates the status of the centrality algorithms at termination. These algorithms are described in the section “Centrality” on page 66. The CENTRALITY term can take one of the following values:

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

**CLIQUE**
indicates the status of the clique-finding algorithms at termination. These algorithms are described in the section “Clique” on page 87. The CLIQUE term can take one of the following values:

- **OK**  The algorithm terminated normally.
- **INTERRUPTED**  The algorithm was interrupted by the user.
- **TIMELIMIT**  The algorithm reached its execution time limit, which is specified in the MAXTIME= option in the CLIQUE statement.
<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOLUTION_LIM</td>
<td>The algorithm reached its limit on the number of cliques found, which is</td>
</tr>
<tr>
<td></td>
<td>specified in the MAXCLIQUES= option in the CLIQUE statement.</td>
</tr>
<tr>
<td>ERROR</td>
<td>The algorithm encountered an error.</td>
</tr>
</tbody>
</table>

**COMMUNITY**

indicates the status of the community algorithms at termination. These algorithms are described in the section “Community Detection” on page 90. The COMMUNITY term can take one of the following values:

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

**CONCOMP**

indicates the status of the connected components algorithm at termination. This algorithm is described in the section “Connected Components” on page 99. The CONCOMP term can take one of the following values:

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

**CORE**

indicates the status of the core decomposition algorithm at termination. This algorithm is described in the section “Core Decomposition” on page 104. The CORE term can take one of the following values:

- **OK** The algorithm terminated normally.
- **INTERRUPTED** The algorithm was interrupted by the user.
- **TIMELIMIT** The algorithm reached its execution time limit, which is specified in the MAXTIME= option in the CORE statement.
- **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 108. The CYCLE term can take one of the following values:

- **OK** The algorithm terminated normally.
- **TIMELIMIT** The algorithm reached its execution time limit, which is specified in the MAXTIME= option in the CYCLE statement.
- **SOLUTION_LIM** The algorithm reached its limit on the number of cycles found, which is specified in the MAXCYCLES= option in the CYCLE statement.
- **ERROR** The algorithm encountered an error.
EIGEN indicates the status of the eigenvector solver at termination. This solver is described in the section “Eigenvector Problem” on page 114. The EIGEN term can take one of the following values:

- **OK**: The solver terminated normally.
- **ITER_LIMIT_SOL**: The solver reached the maximum number of iterations that is specified in the MAXITER= option and found a solution.
- **ITER_LIMIT_NOSOL**: The solver reached the maximum number of iterations that is specified in the MAXITER= option and did not find a solution.
- **ERROR**: The solver encountered an error.

LAP indicates the status of the linear assignment solver at termination. This solver is described in the section “Linear Assignment (Matching)” on page 116. 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 117. The MCF term can take one of the following values:

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

MINCUT indicates the status of the minimum-cut solver at termination. This solver is described in the section “Minimum Cut” on page 125. The MINCUT term can take one of the following values:

- **OPTIMAL**: The solution is optimal.
- **INFEASIBLE**: The problem is infeasible.
- **INTERRUPTED**: The solver was interrupted by the user.
- **ERROR**: The solver encountered an error.
MST
indicates the status of the minimum spanning tree solver at termination. This solver is described in the section “Minimum Spanning Tree” on page 129. The MST term can take one of the following values:

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

REACH
indicates the status of the reach algorithms at termination. These algorithms are described in the section “Reach (Ego) Network” on page 131. The REACH term can take one of the following values:

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

SHORTPATH
indicates the status of the shortest path algorithms at termination. These algorithms are described in the section “Shortest Path” on page 143. The SHORTPATH term can take one of the following values:

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

SUMMARY
indicates the status of the summary algorithms at termination. These algorithms are described in the section “Summary” on page 154. The SUMMARY term can take one of the following values:

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

TRANSCL
indicates the status of the transitive closure algorithm at termination. This algorithm is described in the section “Transitive Closure” on page 163. The TRANSCL term can take one of the following values:

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

TSP
indicates the status of the traveling salesman problem solver at termination. This algorithm is described in the section “Traveling Salesman Problem” on page 165. The TSP term can take one of the following values:
<table>
<thead>
<tr>
<th>Macro Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPTIMAL</td>
<td>The solution is optimal.</td>
</tr>
<tr>
<td>OPTIMAL_AGAP</td>
<td>The solution is optimal within the absolute gap that is specified in the ABSOBJGAP= option.</td>
</tr>
<tr>
<td>OPTIMAL_RGAP</td>
<td>The solution is optimal within the relative gap that is specified in the RELOBJGAP= option.</td>
</tr>
<tr>
<td>OPTIMAL_COND</td>
<td>The solution is optimal, but some infeasibilities (primal, bound, or integer) exceed tolerances because of scaling.</td>
</tr>
<tr>
<td>TARGET</td>
<td>The solution is not worse than the target that is specified in the TARGET= option.</td>
</tr>
<tr>
<td>INFEASIBLE</td>
<td>The problem is infeasible.</td>
</tr>
<tr>
<td>UNBOUNDED</td>
<td>The problem is unbounded.</td>
</tr>
<tr>
<td>INFEASIBLE_OR_UNBOUNDED</td>
<td>The problem is infeasible or unbounded.</td>
</tr>
<tr>
<td>SOLUTION_LIM</td>
<td>The solver reached the maximum number of solutions specified in the MAXSOLS= option.</td>
</tr>
<tr>
<td>NODE_LIM_SOL</td>
<td>The solver reached the maximum number of nodes specified in the MAXNODES= option and found a solution.</td>
</tr>
<tr>
<td>NODE_LIM_NOSOL</td>
<td>The solver reached the maximum number of nodes specified in the MAXNODES= option and did not find a solution.</td>
</tr>
<tr>
<td>TIME_LIM_SOL</td>
<td>The solver reached the execution time limit specified in the MAXTIME= option and found a solution.</td>
</tr>
<tr>
<td>TIME_LIM_NOSOL</td>
<td>The solver reached the execution time limit specified in the MAXTIME= option and did not find a solution.</td>
</tr>
<tr>
<td>HEURISTIC_SOL</td>
<td>The solver used only heuristics and found a solution.</td>
</tr>
<tr>
<td>HEURISTIC_NOSOL</td>
<td>The solver used only heuristics and did not find a solution.</td>
</tr>
<tr>
<td>ABORT_SOL</td>
<td>The solver was stopped by the user but still found a solution.</td>
</tr>
<tr>
<td>ABORT_NOSOL</td>
<td>The solver was stopped by the user and did not find a solution.</td>
</tr>
<tr>
<td>OUTMEM_SOL</td>
<td>The solver ran out of memory but still found a solution.</td>
</tr>
<tr>
<td>OUTMEM_NOSOL</td>
<td>The solver ran out of memory and either did not find a solution or failed to output the solution due to insufficient memory.</td>
</tr>
<tr>
<td>FAIL_SOL</td>
<td>The solver stopped due to errors but still found a solution.yte</td>
</tr>
<tr>
<td>FAIL_NOSOL</td>
<td>The solver stopped due to errors and did not find a solution.yte</td>
</tr>
</tbody>
</table>

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

**Macro Variable _OPTGRAPH_BICONCOMP_**

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_BICONCOMP_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm that PROC OPTGRAPH uses to calculate biconnected components. The various terms of the variable are interpreted as follows:
Chapter 1: The OPTGRAPH Procedure

**STATUS**
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term BICONCOMP in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

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

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

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

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

**Macro Variable _OPTGRAPH_CENTRALITY_**

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

**STATUS**
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term CENTRALITY in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

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

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

**Macro Variable _OPTGRAPH_CLIQUE_**

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

**STATUS**
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term CLIQUE in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

**NUM_CLIQUES**
indicates the number of cliques found by the algorithm.
**CPU_TIME**
indicates the CPU time (in seconds) taken by the algorithm.

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

**Macro Variable _OPTGRAPH_COMMUNITY_**
The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_COMMUNITY_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm that PROC OPTGRAPH uses to calculate communities. The various terms of the variable are interpreted as follows:

**STATUS**
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term COMMUNITY in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

**RESOLUTION**
indicates the list of resolution levels specified in the RESOLUTION_LIST= option.

**NUM_COMMUNITIES**
indicates the number of communities found by the algorithm at each resolution level.

**MODULARITY**
indicates the final modularity found by the algorithm at each resolution level.

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

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

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

**STATUS**
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term CONCOMP in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

**NUM_COMPONENTS**
indicates the number of connected components found by the algorithm.
CPU_TIME
    indicates the CPU time (in seconds) taken by the algorithm.

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

Macro Variable _OPTGRAPH_CORE_

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

STATUS
    indicates the status of the algorithm at termination. The STATUS term takes the same value as the term CORE in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

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

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

Macro Variable _OPTGRAPH_CYCLE_

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

STATUS
    indicates the status of the algorithm at termination. The STATUS term takes the same value as the term CYCLE in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

NUM_CYCLES
    indicates the number of cycles found by the algorithm.

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

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

Macro Variable _OPTGRAPH_EIGEN_

The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_EIGEN_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm that PROC OPTGRAPH uses to calculate eigenvectors. The various terms of the variable are interpreted as follows:
**STATUS**
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term EIGEN in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

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

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

---

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

**STATUS**
indicates the status of the solver at termination. The STATUS term takes the same value as the term LAP in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

**OBJECTIVE**
indicates the total weight of the minimum linear assignment.

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

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

---

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

**STATUS**
indicates the status of the solver at termination. The STATUS term takes the same value as the term MCF in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

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

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

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

STATUS
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term MINCUT in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

OBJECTIVE
indicates the total link weight of the minimum cut.

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

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

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

STATUS
indicates the status of the solver at termination. The STATUS term takes the same value as the term MST in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

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

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

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

Macro Variable _OPTGRAPH_REACH_
The OPTGRAPH procedure defines a macro variable named _OPTGRAPH_REACH_. This variable contains a character string that indicates the status and some basic statistics about the results of the algorithm that PROC OPTGRAPH uses to calculate reach networks. The various terms of the variable are interpreted as follows:
**STATUS**
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term REACH in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

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

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

---

**Macro Variable _OPTGRAPH_SHORTPATH_**

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

**STATUS**
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term SHORTPATH in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

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

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

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

---

**Macro Variable _OPTGRAPH_SUMMARY_**

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

**STATUS**
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term SUMMARY in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

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

**REAL_TIME**
indicates the real time (in seconds) taken by the algorithm.
Macro Variable _OPTGRAPH_TRANSCL_

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

STATUS
indicates the status of the algorithm at termination. The STATUS term takes the same value as the term TRANSCL in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

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

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

Macro Variable _OPTGRAPH_TSP_

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

STATUS
indicates the status of the solver at termination. The STATUS term takes the same value as the term TSP in the _OPTGRAPH_ macro as defined in the section “Macro Variable _OPTGRAPH_” on page 173.

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

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

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

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

$$|BestInteger - BestBound|$$

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

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

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

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

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

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

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

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

ODS Table Names
Each table that the OPTGRAPH procedure creates has a name associated with it, and you must use this name to refer to the table when you use ODS statements. These names are listed in Table 1.59.

Table 1.59 ODS Tables Produced by PROC OPTGRAPH

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

The following code uses the example in the section “Traveling Salesman Problem Applied to a Simple Undirected Graph” on page 166 and calculates both an optimal traveling salesman tour and a minimum spanning tree. This code produces all four ODS output tables listed in Table 1.59.
data LinkSetIn;
  input from $ to $ weight @@;
datalines;
A B 1.0 A C 1.0 A D 1.5 B C 2.0 B D 4.0
B E 3.0 C D 3.0 C F 3.0 C H 4.0 D E 1.5
D F 3.0 D G 4.0 E F 1.0 E G 1.0 F G 2.0
F H 4.0 H I 3.0 I J 1.0 C J 5.0 F J 3.0
F I 1.0 H J 1.0
;
proc optgraph
  loglevel = moderate
  data_links = LinkSetIn
  out_nodes = NodeSetOut;
  mst
    out = MST;
  tsp
    out = TSP;
  performance details;
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_TSP_
%put &_OPTGRAPH_MST_

The performance information table in Figure 1.135 provides information about the computing environment. The value for Number of Threads is the maximum number of threads that are used in processing.

**Figure 1.135 Performance Information Table**

<table>
<thead>
<tr>
<th>Performance Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution Mode</td>
</tr>
<tr>
<td>Number of Threads</td>
</tr>
</tbody>
</table>

The procedure task timing table in Figure 1.136 provides a breakdown of each task and the amount of real time spent in processing.

**Figure 1.136 Task Timing Table**

<table>
<thead>
<tr>
<th>Procedure Task Timing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task</td>
</tr>
<tr>
<td>Input</td>
</tr>
<tr>
<td>Setup</td>
</tr>
<tr>
<td>Minimum Spanning Tree</td>
</tr>
<tr>
<td>Traveling Salesman Problem</td>
</tr>
<tr>
<td>Output</td>
</tr>
</tbody>
</table>

The problem summary table in Figure 1.137 provides a basic summary of the graph (or matrix) input.
The solution summary tables in Figure 1.138 and Figure 1.139 provide a basic solution summary for each algorithm that is processed. The information in these tables matches the information that is provided in the macro variables for each algorithm, described in the section “Macro Variables” on page 173.

**Figure 1.138** Solution Summary Table for MST

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Type</td>
<td>Minimum Spanning Tree</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Optimal</td>
</tr>
<tr>
<td>Objective Value</td>
<td>10</td>
</tr>
<tr>
<td>CPU Time</td>
<td>0.00</td>
</tr>
<tr>
<td>Real Time</td>
<td>0.00</td>
</tr>
</tbody>
</table>

**Figure 1.139** Solution Summary Table for TSP

<table>
<thead>
<tr>
<th>Solution Summary</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Type</td>
<td>Traveling Salesman Problem</td>
</tr>
<tr>
<td>Solution Status</td>
<td>Optimal</td>
</tr>
<tr>
<td>Objective Value</td>
<td>16</td>
</tr>
<tr>
<td>Relative Gap</td>
<td>0</td>
</tr>
<tr>
<td>Absolute Gap</td>
<td>0</td>
</tr>
<tr>
<td>Primal Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Bound Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Integer Infeasibility</td>
<td>0</td>
</tr>
<tr>
<td>Best Bound</td>
<td>16</td>
</tr>
<tr>
<td>Nodes</td>
<td>1</td>
</tr>
<tr>
<td>Iterations</td>
<td>16</td>
</tr>
<tr>
<td>CPU Time</td>
<td>0.01</td>
</tr>
<tr>
<td>Real Time</td>
<td>0.01</td>
</tr>
</tbody>
</table>
Example 1.1: Articulation Points in a Terrorist Network

This example considers the terrorist communications network from the attacks on the United States on September 11, 2001, described in Krebs 2002. Figure 1.140 shows this network, which was constructed after the attacks, based on collected intelligence information.

Figure 1.140  Terrorist Communications Network from 9/11

The full network data include 153 links. The following statements show a small subset to illustrate the use of the BICONCOMP statement in this context:
data LinkSetInTerror911;
  input from & $32. to & $32.;
datalines;
Abu Zubeida    Djamal Beghal
Jean-Marc Grandvisir Djamal Beghal
Nizar Trabelsi  Djamal Beghal
Abu Walid      Djamal Beghal
Abu Qatada     Djamal Beghal
Zacarias Moussaoui Djamal Beghal
Jerome Courtaillier Djamal Beghal
Kamel Daoudi    Djamal Beghal
Abu Walid      Kamel Daoudi
Abu Walid      Abu Qatada
Kamel Daoudi    Zacarias Moussaoui
Kamel Daoudi    Jerome Courtaillier
Jerome Courtaillier Zacarias Moussaoui
Jerome Courtaillier David Courtaillier
Zacarias Moussaoui David Courtaillier
Zacarias Moussaoui Ahmed Ressam
Zacarias Moussaoui Abu Qatada
Zacarias Moussaoui Ramzi Bin al-Shibh
Zacarias Moussaoui Mahamed Atta
Ahmed Ressam    Haydar Abu Doha
Mehdi Khammoun  Haydar Abu Doha
Essid Sami Ben Khemais Haydar Abu Doha
Mehdi Khammoun  Essid Sami Ben Khemais
Mehdi Khammoun  Mohamed Bensakhria
...
;  
Suppose that this communications network had been discovered before the attack on 9/11. If the investigators’ goal was to disrupt the flow of communication between different groups within the organization, then they would want to focus on the people who are articulation points in the network. 

To find the articulation points, use the following statements:

    proc optgraph
      data_links = LinkSetInTerror911
      out_nodes = NodeSetOut;
      biconcomp;
    run;

data ArtPoints;
  set NodeSetOut;
  where artpoint=1;
run;
Chapter 1: The OPTGRAPH Procedure

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

Output 1.1.1 Articulation Points of Terrorist Communications Network from 9/11

<table>
<thead>
<tr>
<th>node</th>
<th>artpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>Djamil Beghal</td>
<td>1</td>
</tr>
<tr>
<td>Zacarias Moussaoui</td>
<td>1</td>
</tr>
<tr>
<td>Essid Sami Ben Khemais</td>
<td>1</td>
</tr>
<tr>
<td>Mohamed Atta</td>
<td>1</td>
</tr>
<tr>
<td>Mamoun Darkazanli</td>
<td>1</td>
</tr>
<tr>
<td>Nawaf Alhazmi</td>
<td>1</td>
</tr>
</tbody>
</table>

Example 1.2: Influence Centrality for Project Groups in a Research Department

This example looks at an undirected graph that represents a few of the project groups in a hypothetical research department. A link between nodes A and B means that person A and B work together or that person A reports to person B. This graph represents six main projects.

- Department 1 (D1) consists of Snopp, Gukrishnan, Leon, and Kabutz. Snopp reports to Chapman.
- Department 2 (D2) consists of Oliver, Gotti, Patrick, and Zhuo. Oliver reports to Chapman.
- Department 3 (D3) consists of Gotti, Leon, and Kabutz. Gotti reports to Chapman.
- Department 4 (D4) consists of the following project teams who report to Yu. Yu reports to Chapman on this project.
  - Department 4a (D4a) consists of Polark, Chang, Weng, and Angel. Polark reports to Yu.
  - Department 4b (D4b) consists of Christoph, Nardo, Gotti, and Zhuo. Christoph reports to Yu.
  - Department 4c (D4c) consists of Graffe, Zhuo, and Hund. Graffe reports to Yu.

The links are shown in Figure 1.141.
The link weights measure the reporting magnitude. In general the higher the weight, the higher the contribution to the influence metric. Chapman is the director of the overall department, and Yu is the manager of a subgroup. The leads for the projects D1, D2, and D3 report to Chapman, and the leads for D4a, D4b, and D4c report to Yu. Reporting links to the director, Chapman, are given a link weight of 3, and reporting links to Yu are given a weight of 2. Links that represent people working together on a project all receive equal weight of 1. The node weights also represent some level of reporting: directors (4), managers (3), leads (2), and all others (1).

The project graph can be represented in the following link and node data sets:

```r
data LinkSetInDept;
    input from $1-12 to $13-24 weight;
datalines;
Yu       Chapman  3
Gotti    Chapman  3
Oliver   Chapman  3
Snopp    Chapman  3
Gukrishnan Leon  1
Snopp    Gukrishnan 1
Kabutz   Gukrishnan 1
Kabutz   Snopp    1
Snopp    Leon     1
Kabutz   Leon     1
Gotti    Oliver   1
Gotti    Patrick  1
Oliver   Patrick  1
```

The link weights measure the reporting magnitude. In general the higher the weight, the higher the contribution to the influence metric. Chapman is the director of the overall department, and Yu is the manager of a subgroup. The leads for the projects D1, D2, and D3 report to Chapman, and the leads for D4a, D4b, and D4c report to Yu. Reporting links to the director, Chapman, are given a link weight of 3, and reporting links to Yu are given a weight of 2. Links that represent people working together on a project all receive equal weight of 1. The node weights also represent some level of reporting: directors (4), managers (3), leads (2), and all others (1).
data NodeSetInDept;
  input node $1-12 weight;
  datalines;
  Chapman 4
  Yu 3
  Gotti 2
  Polark 2
  Christoph 2
  Oliver 2
  Snopp 2
  Zhuo 1
  Nardo 1
  Weng 1
  Chang 1
  Hund 1
  Graffe 1
  Leon 1
  Gukrishnan 1
  Kabutz 1
  Patrick 1
  Angel 1
;

The following statements calculate influence centrality (in addition to degree centrality):

proc optgraph
  loglevel = moderate
  data_links = LinkSetInDept
  data_nodes = NodeSetInDept
  out_nodes = NodeSetOut;
Example 1.2: Influence Centrality for Project Groups in a Research Department

```sas
centrality
degree = out
influence = weight;
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_CENTRALITY_;
```

The progress of the procedure is shown in Output 1.2.1.

**Output 1.2.1** PROC OPTGRAPH Log: Influence Centrality for Project Groups in a Research Department

```
NOTE: --------------------------------------------------------------
NOTE: Running OPTGRAPH version 14.3.
NOTE: --------------------------------------------------------------
NOTE: The OPTGRAPH procedure is executing in single-machine mode.
NOTE: Reading the nodes data set.
NOTE: There were 18 observations read from the data set WORK.NODESETINDEPT.
NOTE: Reading the links data set.
NOTE: There were 35 observations read from the data set WORK.LINKSETINDEPT.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: Building the input graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The input graph storage is using 0.0 MBs (peak: 0.0 MBs) of memory.
NOTE: The number of nodes in the input graph is 18.
NOTE: The number of links in the input graph is 35.
NOTE: --------------------------------------------------------------
NOTE: Processing centrality metrics.
NOTE: --------------------------------------------------------------
NOTE: Processing degree centrality metrics.
NOTE: Processing centrality metrics used 0.0 MBs of memory.
NOTE: Processing degree centrality metrics used 0.00 (cpu: 0.00) seconds.
NOTE: --------------------------------------------------------------
NOTE: Processing influence centrality metrics.
NOTE: Processing centrality metrics used 0.0 MBs of memory.
NOTE: Processing influence centrality metrics used 0.00 (cpu: 0.00) seconds.
NOTE: --------------------------------------------------------------
NOTE: Processing centrality metrics used 0.00 (cpu: 0.00) seconds.
NOTE: --------------------------------------------------------------
NOTE: Creating nodes data set output.
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: --------------------------------------------------------------
NOTE: The data set WORK.NODESETOUT has 18 observations and 5 variables.
STATUS=OK  CENTRALITY=OK
STATUS=OK  CPU_TIME=0.00  REAL_TIME=0.00
```
The node data set NodeSetOut now contains the weighted influence centrality of the department’s graph, including $C_1$ (variable centr_influence1_wt) and $C_2$ (variable centr_influence2_wt). This data set is shown in Output 1.2.2.

**Output 1.2.2** Influence Centrality for Project Groups in a Research Department

<table>
<thead>
<tr>
<th>node</th>
<th>weight</th>
<th>centr_degree_out</th>
<th>centr_influence1_wt</th>
<th>centr_influence2_wt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gotti</td>
<td>2</td>
<td>8</td>
<td>0.35714</td>
<td>1.57143</td>
</tr>
<tr>
<td>Zhuo</td>
<td>1</td>
<td>7</td>
<td>0.25000</td>
<td>1.17857</td>
</tr>
<tr>
<td>Oliver</td>
<td>2</td>
<td>4</td>
<td>0.21429</td>
<td>1.14286</td>
</tr>
<tr>
<td>Chapman</td>
<td>4</td>
<td>4</td>
<td>0.42857</td>
<td>1.10714</td>
</tr>
<tr>
<td>Christoph</td>
<td>2</td>
<td>4</td>
<td>0.17857</td>
<td>1.03571</td>
</tr>
<tr>
<td>Yu</td>
<td>3</td>
<td>4</td>
<td>0.32143</td>
<td>0.92857</td>
</tr>
<tr>
<td>Snopp</td>
<td>2</td>
<td>4</td>
<td>0.21429</td>
<td>0.82143</td>
</tr>
<tr>
<td>Leon</td>
<td>1</td>
<td>4</td>
<td>0.14286</td>
<td>0.82143</td>
</tr>
<tr>
<td>Patrick</td>
<td>1</td>
<td>3</td>
<td>0.10714</td>
<td>0.82143</td>
</tr>
<tr>
<td>Kabutz</td>
<td>1</td>
<td>4</td>
<td>0.14286</td>
<td>0.82143</td>
</tr>
<tr>
<td>Nardo</td>
<td>1</td>
<td>3</td>
<td>0.10714</td>
<td>0.78571</td>
</tr>
<tr>
<td>Polark</td>
<td>2</td>
<td>4</td>
<td>0.17857</td>
<td>0.64286</td>
</tr>
<tr>
<td>Graffe</td>
<td>1</td>
<td>3</td>
<td>0.14286</td>
<td>0.64286</td>
</tr>
<tr>
<td>Gukrishnan</td>
<td>1</td>
<td>3</td>
<td>0.10714</td>
<td>0.50000</td>
</tr>
<tr>
<td>Weng</td>
<td>1</td>
<td>3</td>
<td>0.10714</td>
<td>0.39286</td>
</tr>
<tr>
<td>Chang</td>
<td>1</td>
<td>3</td>
<td>0.10714</td>
<td>0.39286</td>
</tr>
<tr>
<td>Hund</td>
<td>1</td>
<td>2</td>
<td>0.07143</td>
<td>0.39286</td>
</tr>
<tr>
<td>Angel</td>
<td>1</td>
<td>3</td>
<td>0.10714</td>
<td>0.39286</td>
</tr>
</tbody>
</table>

As expected, the director Chapman has the highest first-order influence, since the weights of the reporting links to him are high. The highest second-order influence is Gotti, who reports to the director but is also involved in three different projects and therefore has a large sphere of influence. This example is revisited with other centrality metrics in other examples.

**Example 1.3: Betweenness and Closeness Centrality for Computer Network Topology**

Consider a small network of 10 computers spread out across an office. Let a node represent a computer, and let a link represent a direct connection between the machines. For this example, consider the links as Ethernet connections that enable data to transfer between computers. If two computers are not connected directly, then the information must flow through other connected machines. Consider a topology as shown in Figure 1.142. This is an example of the well-known kite network, which was popularized by David Krackhardt (1990) for better understanding of social networks in the workplace.
Define the link data set as follows:

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

To better understand the topology of the computer network, calculate the degree, closeness, and betweenness centrality. It is also interesting to look for articulation points in the computer network to identify places of vulnerability. All of these calculations can be done in one call to PROC OPTGRAPH as follows:

```
proc optgraph
  data_links = LinkSetInCompNet
  out_links = LinkSetOut
  out_nodes = NodeSetOut;
  centrality
    degree = out
    close = unweight
    between = unweight;
  biconcomp;
run;
```

**Output 1.3.1** shows the resulting node data set NodeSetOut sorted by closeness.
Output 1.3.1 Node Closeness and Betweenness Centrality, Sorted by Closeness

<table>
<thead>
<tr>
<th>node</th>
<th>centr_degree_out</th>
<th>centr_close_unwt</th>
<th>centr_between_unwt</th>
<th>arpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>5</td>
<td>0.64286</td>
<td>0.23148</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>5</td>
<td>0.64286</td>
<td>0.23148</td>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>0.60000</td>
<td>0.10185</td>
<td>0</td>
</tr>
<tr>
<td>H</td>
<td>3</td>
<td>0.60000</td>
<td>0.38889</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>4</td>
<td>0.52941</td>
<td>0.02315</td>
<td>0</td>
</tr>
<tr>
<td>E</td>
<td>4</td>
<td>0.52941</td>
<td>0.02315</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>3</td>
<td>0.50000</td>
<td>0.00000</td>
<td>0</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
<td>0.50000</td>
<td>0.00000</td>
<td>0</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
<td>0.42857</td>
<td>0.22222</td>
<td>1</td>
</tr>
<tr>
<td>J</td>
<td>1</td>
<td>0.31034</td>
<td>0.00000</td>
<td>0</td>
</tr>
</tbody>
</table>

Output 1.3.2 shows the resulting node (NodeSetOut) and link data sets (LinkSetOut) sorted by betweenness.

Output 1.3.2 Node Closeness and Betweenness Centrality, Sorted by Betweenness

<table>
<thead>
<tr>
<th>Obs</th>
<th>node</th>
<th>centr_degree_out</th>
<th>centr_close_unwt</th>
<th>centr_between_unwt</th>
<th>arpoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H</td>
<td>3</td>
<td>0.60000</td>
<td>0.38889</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>5</td>
<td>0.64286</td>
<td>0.23148</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>5</td>
<td>0.64286</td>
<td>0.23148</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>I</td>
<td>2</td>
<td>0.42857</td>
<td>0.22222</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>6</td>
<td>0.60000</td>
<td>0.10185</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>B</td>
<td>4</td>
<td>0.52941</td>
<td>0.02315</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>E</td>
<td>4</td>
<td>0.52941</td>
<td>0.02315</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>A</td>
<td>3</td>
<td>0.50000</td>
<td>0.00000</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>G</td>
<td>3</td>
<td>0.50000</td>
<td>0.00000</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>J</td>
<td>1</td>
<td>0.31034</td>
<td>0.00000</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Obs</th>
<th>from</th>
<th>to</th>
<th>biconcomp</th>
<th>centr_between_unwt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>H</td>
<td>I</td>
<td>2</td>
<td>0.44444</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>H</td>
<td>3</td>
<td>0.29167</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>H</td>
<td>3</td>
<td>0.29167</td>
</tr>
<tr>
<td>4</td>
<td>I</td>
<td>J</td>
<td>1</td>
<td>0.25000</td>
</tr>
<tr>
<td>5</td>
<td>B</td>
<td>C</td>
<td>3</td>
<td>0.12963</td>
</tr>
<tr>
<td>6</td>
<td>E</td>
<td>F</td>
<td>3</td>
<td>0.12963</td>
</tr>
<tr>
<td>7</td>
<td>A</td>
<td>C</td>
<td>3</td>
<td>0.12500</td>
</tr>
<tr>
<td>8</td>
<td>F</td>
<td>G</td>
<td>3</td>
<td>0.12500</td>
</tr>
<tr>
<td>9</td>
<td>C</td>
<td>D</td>
<td>3</td>
<td>0.09259</td>
</tr>
<tr>
<td>10</td>
<td>D</td>
<td>F</td>
<td>3</td>
<td>0.09259</td>
</tr>
<tr>
<td>11</td>
<td>A</td>
<td>D</td>
<td>3</td>
<td>0.08333</td>
</tr>
<tr>
<td>12</td>
<td>D</td>
<td>G</td>
<td>3</td>
<td>0.08333</td>
</tr>
<tr>
<td>13</td>
<td>B</td>
<td>E</td>
<td>3</td>
<td>0.07407</td>
</tr>
<tr>
<td>14</td>
<td>C</td>
<td>F</td>
<td>3</td>
<td>0.07407</td>
</tr>
<tr>
<td>15</td>
<td>B</td>
<td>D</td>
<td>3</td>
<td>0.05093</td>
</tr>
<tr>
<td>16</td>
<td>D</td>
<td>E</td>
<td>3</td>
<td>0.05093</td>
</tr>
<tr>
<td>17</td>
<td>A</td>
<td>B</td>
<td>3</td>
<td>0.04167</td>
</tr>
<tr>
<td>18</td>
<td>E</td>
<td>G</td>
<td>3</td>
<td>0.04167</td>
</tr>
</tbody>
</table>
The computers with the highest closeness centrality are $C$ and $F$, because they have the shortest paths to all other nodes. These computers are key to the efficient distribution of information across the network. Assuming that the entire office has some centralized data that should be shared with all computers, machines $C$ and $F$ would be the best candidates for storing the data on their local hard drives. The computer with the highest betweenness centrality is $H$. Although machine $H$ has only three connections, it is one of the most important machines in the office because it serves as the only way to reach computers $I$ and $J$ from the other machines in the office. Notice also that machine $H$ is an articulation point because removing it would disconnect the office network. In this setting, computers with high betweenness should be carefully maintained and secured with UPS (uninterruptible power supply) systems to ensure they are always online.

Example 1.4: Betweenness and Closeness Centrality for Project Groups in a Research Department

This example uses the same data as are used in the section “Example 1.2: Influence Centrality for Project Groups in a Research Department” on page 190, which illustrates influence centrality by considering the link weights that represent some measure of reporting magnitude. In “Example 1.2: Influence Centrality for Project Groups in a Research Department” on page 190, links between managers (or leads) and direct reports had higher link weights than links between non-managers. This interpretation makes sense in the context of influence centrality because weight and the metric are directly related. However, weight and the metric are inversely related for closeness and betweenness centrality.

This example considers the speed of the flow of information between people. In this sense, connections between managers and direct reports have smaller values, which cost less in the shortest path calculations. The following DATA step produces a new links data set, based on LinkSetInDept, which uses the inverse of the weight:

```plaintext
data LinkSetInDeptInv;
    set LinkSetInDept;
    weight = 1 / weight;
run;
```

The following statements calculate weighted (and unweighted) closeness and betweenness centrality. Notice that this example also uses the NTHREADS= option in the PERFORMANCE statement to specify two threads to allow the computation to be run in parallel. Since these data have 18 nodes, each thread can process 9 nodes simultaneously.

```plaintext
class proc optgraph
    loglevel = moderate
    data_links = LinkSetInDeptInv
    out_links = LinkSetOut
    out_nodes = NodeSetOut;
    performance
        nthreads = 2;
        centrality
            close = both
            between = both;
run;
%put &_OPTGRAPH_;%put &_OPTGRAPH_CENTRALITY_;```
The progress of the procedure is shown in Output 1.4.1.

**Output 1.4.1** PROC OPTGRAPH Log: Closeness and Node Betweenness Centrality for Project Groups in a Research Department

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cpu</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>betwNL/close(wt)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nodes Complete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Processing weighted between/close centrality metrics using 2 threads.
NOTE: Processing centrality metrics used 0.1 MBs of memory.
NOTE: Processing weighted between/close centrality metrics used 0.00 (cpu: 0.00) seconds.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Cpu</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td>betwNL/close(unwt)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Nodes Complete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

NOTE: Processing unweighted between/close centrality metrics using 2 threads.
NOTE: Processing centrality metrics used 0.1 MBs of memory.
NOTE: Processing unweighted between/close centrality metrics used 0.00 (cpu: 0.00) seconds.
The node data set `NodeSetOut` shows the weighted and unweighted closeness and node betweenness centrality, as shown in Output 1.4.2.

**Output 1.4.2** Closeness and Betweenness Centrality for Project Groups in a Research Department

<table>
<thead>
<tr>
<th>node</th>
<th>centr_close_wt</th>
<th>centr_close_unwt</th>
<th>centr_between_wt</th>
<th>centr_between_unwt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yu</td>
<td>0.87179</td>
<td>0.50000</td>
<td>0.50000</td>
<td>0.41262</td>
</tr>
<tr>
<td>Chapman</td>
<td>0.88696</td>
<td>0.50000</td>
<td>0.44118</td>
<td>0.23235</td>
</tr>
<tr>
<td>Gotti</td>
<td>0.81600</td>
<td>0.51515</td>
<td>0.20956</td>
<td>0.28444</td>
</tr>
<tr>
<td>Oliver</td>
<td>0.73913</td>
<td>0.44737</td>
<td>0.04044</td>
<td>0.02230</td>
</tr>
<tr>
<td>Snopp</td>
<td>0.75556</td>
<td>0.38636</td>
<td>0.16176</td>
<td>0.08088</td>
</tr>
<tr>
<td>Gukrishnan</td>
<td>0.46575</td>
<td>0.32692</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>Leon</td>
<td>0.50746</td>
<td>0.38636</td>
<td>0.00000</td>
<td>0.03885</td>
</tr>
<tr>
<td>Kabutz</td>
<td>0.50746</td>
<td>0.38636</td>
<td>0.00000</td>
<td>0.03885</td>
</tr>
<tr>
<td>Patrick</td>
<td>0.50000</td>
<td>0.37778</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>Zhuo</td>
<td>0.58286</td>
<td>0.47222</td>
<td>0.06618</td>
<td>0.15172</td>
</tr>
<tr>
<td>Polark</td>
<td>0.69388</td>
<td>0.38636</td>
<td>0.30882</td>
<td>0.30882</td>
</tr>
<tr>
<td>Chang</td>
<td>0.44156</td>
<td>0.29310</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>Angel</td>
<td>0.44156</td>
<td>0.29310</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>Weng</td>
<td>0.44156</td>
<td>0.29310</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>Christoph</td>
<td>0.68456</td>
<td>0.48571</td>
<td>0.05882</td>
<td>0.11275</td>
</tr>
<tr>
<td>Nardo</td>
<td>0.51777</td>
<td>0.42500</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>Graffe</td>
<td>0.67105</td>
<td>0.43590</td>
<td>0.08088</td>
<td>0.06642</td>
</tr>
<tr>
<td>Hund</td>
<td>0.45133</td>
<td>0.36957</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

The link data set `LinkSetOut` shows the weighted and unweighted link betweenness centrality, as shown in Output 1.4.3.
Output 1.4.3  Link Betweenness Centrality for Project Groups in a Research Department

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>weight</th>
<th>centr_between_wt</th>
<th>centr_between_unwt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yu</td>
<td>Chapman</td>
<td>0.33333</td>
<td>0.39706</td>
<td>0.25576</td>
</tr>
<tr>
<td>Gotti</td>
<td>Chapman</td>
<td>0.33333</td>
<td>0.20221</td>
<td>0.09767</td>
</tr>
<tr>
<td>Oliver</td>
<td>Chapman</td>
<td>0.33333</td>
<td>0.14338</td>
<td>0.07623</td>
</tr>
<tr>
<td>Snopp</td>
<td>Chapman</td>
<td>0.33333</td>
<td>0.26471</td>
<td>0.16005</td>
</tr>
<tr>
<td>Gukrishnan</td>
<td>Leon</td>
<td>1.00000</td>
<td>0.00735</td>
<td>0.03431</td>
</tr>
<tr>
<td>Snopp</td>
<td>Gukrishnan</td>
<td>1.00000</td>
<td>0.11029</td>
<td>0.05637</td>
</tr>
<tr>
<td>Kabutz</td>
<td>Gukrishnan</td>
<td>1.00000</td>
<td>0.00735</td>
<td>0.03431</td>
</tr>
<tr>
<td>Kabutz</td>
<td>Snopp</td>
<td>1.00000</td>
<td>0.03676</td>
<td>0.03517</td>
</tr>
<tr>
<td>Snopp</td>
<td>Leon</td>
<td>1.00000</td>
<td>0.03676</td>
<td>0.03517</td>
</tr>
<tr>
<td>Kabutz</td>
<td>Leon</td>
<td>1.00000</td>
<td>0.00735</td>
<td>0.00735</td>
</tr>
<tr>
<td>Gotti</td>
<td>Oliver</td>
<td>1.00000</td>
<td>0.00000</td>
<td>0.03431</td>
</tr>
<tr>
<td>Gotti</td>
<td>Patrick</td>
<td>1.00000</td>
<td>0.05882</td>
<td>0.06066</td>
</tr>
<tr>
<td>Oliver</td>
<td>Patrick</td>
<td>1.00000</td>
<td>0.03676</td>
<td>0.02022</td>
</tr>
<tr>
<td>Zhuo</td>
<td>Oliver</td>
<td>1.00000</td>
<td>0.02574</td>
<td>0.03885</td>
</tr>
<tr>
<td>Zhuo</td>
<td>Gotti</td>
<td>1.00000</td>
<td>0.05515</td>
<td>0.10184</td>
</tr>
<tr>
<td>Zhuo</td>
<td>Patrick</td>
<td>1.00000</td>
<td>0.02941</td>
<td>0.04412</td>
</tr>
<tr>
<td>Kabutz</td>
<td>Gotti</td>
<td>1.00000</td>
<td>0.07353</td>
<td>0.12586</td>
</tr>
<tr>
<td>Leon</td>
<td>Gotti</td>
<td>1.00000</td>
<td>0.07353</td>
<td>0.12586</td>
</tr>
<tr>
<td>Polark</td>
<td>Yu</td>
<td>0.50000</td>
<td>0.41176</td>
<td>0.41176</td>
</tr>
<tr>
<td>Polark</td>
<td>Chang</td>
<td>1.00000</td>
<td>0.11029</td>
<td>0.11029</td>
</tr>
<tr>
<td>Chang</td>
<td>Angel</td>
<td>1.00000</td>
<td>0.00735</td>
<td>0.00735</td>
</tr>
<tr>
<td>Polark</td>
<td>Angel</td>
<td>1.00000</td>
<td>0.11029</td>
<td>0.11029</td>
</tr>
<tr>
<td>Weng</td>
<td>Polark</td>
<td>1.00000</td>
<td>0.11029</td>
<td>0.11029</td>
</tr>
<tr>
<td>Weng</td>
<td>Chang</td>
<td>1.00000</td>
<td>0.00735</td>
<td>0.00735</td>
</tr>
<tr>
<td>Weng</td>
<td>Angel</td>
<td>1.00000</td>
<td>0.00735</td>
<td>0.00735</td>
</tr>
<tr>
<td>Christoph</td>
<td>Yu</td>
<td>0.50000</td>
<td>0.13603</td>
<td>0.15870</td>
</tr>
<tr>
<td>Christoph</td>
<td>Nardo</td>
<td>1.00000</td>
<td>0.04779</td>
<td>0.04412</td>
</tr>
<tr>
<td>Christoph</td>
<td>Gotti</td>
<td>1.00000</td>
<td>0.02574</td>
<td>0.09620</td>
</tr>
<tr>
<td>Christoph</td>
<td>Zhuo</td>
<td>1.00000</td>
<td>0.03309</td>
<td>0.05147</td>
</tr>
<tr>
<td>Nardo</td>
<td>Gotti</td>
<td>1.00000</td>
<td>0.05515</td>
<td>0.05147</td>
</tr>
<tr>
<td>Nardo</td>
<td>Zhuo</td>
<td>1.00000</td>
<td>0.02206</td>
<td>0.02941</td>
</tr>
<tr>
<td>Graffe</td>
<td>Yu</td>
<td>0.50000</td>
<td>0.18015</td>
<td>0.12402</td>
</tr>
<tr>
<td>Graffe</td>
<td>Hund</td>
<td>1.00000</td>
<td>0.06985</td>
<td>0.04804</td>
</tr>
<tr>
<td>Graffe</td>
<td>Zhuo</td>
<td>1.00000</td>
<td>0.03676</td>
<td>0.08578</td>
</tr>
<tr>
<td>Zhuo</td>
<td>Hund</td>
<td>1.00000</td>
<td>0.05515</td>
<td>0.07696</td>
</tr>
</tbody>
</table>

Note that Chapman (director) and Yu (manager, reporting to Chapman) both have the highest weighted closeness centrality. However, Yu’s weighted betweenness centrality is highest because he serves as more of a *gatekeeper* between his three groups (D4a, D4b, and D4c) and the rest of the department.

Example 1.5: Eigenvector Centrality for Word Sense Disambiguation

In many languages, numerous words are polysemous (they carry more than one meaning). A common task in information retrieval is to assign the correct meaning to a polysemous word within a given context. Take the
word “bass” as an example. It can mean either a type of fish (as in the sentence “I went fishing for some sea bass”) or tones of low frequency (as in the sentence “The bass part of the song is very moving”).

The following example from Mihalcea 2005 shows how eigenvector centrality can be used to disambiguate the word sense in the sentence “The church bells no longer ring on Sundays.” The following senses of words can be drawn from a dictionary:

- **church**
  1. one of the groups of Christians who have their own beliefs and forms of worship
  2. a place for public (especially Christian) worship
  3. a service conducted in a church

- **bell**
  1. a hollow device made of metal that makes a ringing sound when struck
  2. a push button at an outer door that gives a ringing or buzzing signal when pushed
  3. the sound of a bell

- **ring**
  1. make a ringing sound
  2. ring or echo with sound
  3. make (bells) ring, often for the purposes of musical edification

- **Sunday**
  1. first day of the week; observed as a day of rest and worship by most Christians

Using one of the similarity metrics defined in Sinha and Mihalcea 2007, you can generate a graph in which the nodes correspond to the word senses given above and the weights are determined by the similarity metric. The resulting graph is shown in Figure 1.143.
To identify the correct senses, you run eigenvector centrality on the graph and select the highest ranking sense for each word:

```sas
data LinkSetIn;
  input from $ to $ weight;
datalines;
  bell_1 ring_1 0.85
  bell_1 ring_2 0.55
  bell_1 ring_3 1.01
  bell_2 ring_1 0.40
  bell_2 ring_2 0.35
  bell_2 ring_3 0.80
  bell_3 ring_1 0.23
  bell_3 ring_2 0.19
  bell_3 ring_3 1.06
  ring_3 church_1 0.30
  ring_3 church_2 0.34
  ring_3 church_3 0.50
  church_1 sunday_1 0.31
  church_2 sunday_1 0.35
;
proc optgraph
  data_links = LinkSetIn
  out_nodes = NodeSetOut;
centrality
  eigen = weight;
run;

data NodeSetOut;
  length word $8 sense $1;
  set NodeSetOut;
  word = scan(node,1,'_');
  sense = scan(node,2,'_');
run;

proc sort
  data = NodeSetOut
  out = WordSenses;
  by word descending centr_eigen_wt;
run;

data WordSenses;
  set WordSenses(drop=centr_eigen_wt);
  by word;
  if first.word then output;
run;
```

The eigenvector scores and the implied word sense are shown in Output 1.5.1.
Output 1.5.1 Eigenvector Centrality for Word Sense Disambiguation

<table>
<thead>
<tr>
<th>node</th>
<th>centr_eigen_wt</th>
</tr>
</thead>
<tbody>
<tr>
<td>ring_3</td>
<td>1.00000</td>
</tr>
<tr>
<td>bell_1</td>
<td>0.77997</td>
</tr>
<tr>
<td>bell_3</td>
<td>0.59692</td>
</tr>
<tr>
<td>bell_2</td>
<td>0.53889</td>
</tr>
<tr>
<td>ring_1</td>
<td>0.48924</td>
</tr>
<tr>
<td>ring_2</td>
<td>0.35207</td>
</tr>
<tr>
<td>church_3</td>
<td>0.24081</td>
</tr>
<tr>
<td>church_2</td>
<td>0.17248</td>
</tr>
<tr>
<td>church_1</td>
<td>0.15222</td>
</tr>
<tr>
<td>sunday_1</td>
<td>0.05180</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>word</th>
<th>sense</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>bell</td>
<td>1</td>
<td>bell_1</td>
</tr>
<tr>
<td>church</td>
<td>3</td>
<td>church_3</td>
</tr>
<tr>
<td>ring</td>
<td>3</td>
<td>ring_3</td>
</tr>
<tr>
<td>sunday</td>
<td>1</td>
<td>sunday_1</td>
</tr>
</tbody>
</table>

Example 1.6: Centrality Metrics for Project Groups in a Research Department

The following statements use the WEIGHT2= option, and the project groups in a research department as depicted in Figure 1.141 on page 191. The data set contains the original weight and its inverse, which is used in the calculations of closeness and betweenness.

```plaintext
data LinkSetInDept2;
  input from $1-12$ to $13-24$ weight weightInv;
datalines;
  Yu     Chapman 3 0.33
  Gotti  Chapman 3 0.33
  Oliver Chapman 3 0.33
  Snopp  Chapman 3 0.33
  Gukrishnan Leon 1 1
  Snopp  Gukrishnan 1 1
  Kabutz Gukrishnan 1 1
  Kabutz Snopp 1 1
  Snopp  Leon 1 1
  Kabutz Leon 1 1
  Gotti  Oliver 1 1
  Gotti  Patrick 1 1
  Oliver Patrick 1 1
  Zhuo  Oliver 1 1
  Zhuo  Gotti 1 1
  Zhuo  Patrick 1 1
  Kabutz Gotti 1 1
  Leon  Gotti 1 1
  Polark Yu 2 0.50
  Polark Chang 1 1
  Chang  Angel 1 1
  Polark Angel 1 1
```
The node data set NodeSetOut now shows the resulting centrality metrics given both weight interpretations.
Example 1.6: Centrality Metrics for Project Groups in a Research Department

Output 1.6.1 Centrality for Project Groups in a Research Department

<table>
<thead>
<tr>
<th>node</th>
<th>weight</th>
<th>centr_degree_out</th>
<th>centr_eigen_wt</th>
<th>centr_close_wt</th>
<th>centr_between_wt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapman</td>
<td>4</td>
<td>4</td>
<td>1.00000</td>
<td>0.88959</td>
<td>0.44118</td>
</tr>
<tr>
<td>Yu</td>
<td>3</td>
<td>4</td>
<td>0.62475</td>
<td>0.87404</td>
<td>0.50000</td>
</tr>
<tr>
<td>Gotti</td>
<td>2</td>
<td>8</td>
<td>0.70480</td>
<td>0.81849</td>
<td>0.20956</td>
</tr>
<tr>
<td>Polark</td>
<td>2</td>
<td>4</td>
<td>0.18777</td>
<td>0.69530</td>
<td>0.30882</td>
</tr>
<tr>
<td>Christoph</td>
<td>2</td>
<td>4</td>
<td>0.34168</td>
<td>0.68521</td>
<td>0.05882</td>
</tr>
<tr>
<td>Oliver</td>
<td>2</td>
<td>4</td>
<td>0.58858</td>
<td>0.74203</td>
<td>0.04044</td>
</tr>
<tr>
<td>Snopp</td>
<td>2</td>
<td>4</td>
<td>0.49133</td>
<td>0.75859</td>
<td>0.16176</td>
</tr>
<tr>
<td>Zhuo</td>
<td>1</td>
<td>7</td>
<td>0.32567</td>
<td>0.58319</td>
<td>0.06618</td>
</tr>
<tr>
<td>Nardo</td>
<td>1</td>
<td>3</td>
<td>0.18983</td>
<td>0.51813</td>
<td>0.00000</td>
</tr>
<tr>
<td>Weng</td>
<td>1</td>
<td>3</td>
<td>0.03591</td>
<td>0.44213</td>
<td>0.00000</td>
</tr>
<tr>
<td>Chang</td>
<td>1</td>
<td>3</td>
<td>0.03591</td>
<td>0.44213</td>
<td>0.00000</td>
</tr>
<tr>
<td>Hund</td>
<td>1</td>
<td>2</td>
<td>0.07667</td>
<td>0.45153</td>
<td>0.00000</td>
</tr>
<tr>
<td>Graffe</td>
<td>1</td>
<td>3</td>
<td>0.22852</td>
<td>0.67220</td>
<td>0.08088</td>
</tr>
<tr>
<td>Leon</td>
<td>1</td>
<td>4</td>
<td>0.21239</td>
<td>0.50822</td>
<td>0.00000</td>
</tr>
<tr>
<td>Gukrishnan</td>
<td>1</td>
<td>3</td>
<td>0.12674</td>
<td>0.46690</td>
<td>0.00000</td>
</tr>
<tr>
<td>Kabutz</td>
<td>1</td>
<td>4</td>
<td>0.21239</td>
<td>0.50822</td>
<td>0.00000</td>
</tr>
<tr>
<td>Patrick</td>
<td>1</td>
<td>3</td>
<td>0.22398</td>
<td>0.50074</td>
<td>0.00000</td>
</tr>
<tr>
<td>Angel</td>
<td>1</td>
<td>3</td>
<td>0.03591</td>
<td>0.44213</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>centr_influence1_wt</th>
<th>centr_influence2_wt</th>
<th>centr_cluster</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.42857</td>
<td>1.10714</td>
<td>0.16667</td>
</tr>
<tr>
<td>0.32143</td>
<td>0.92857</td>
<td>0.00000</td>
</tr>
<tr>
<td>0.35714</td>
<td>1.57143</td>
<td>0.28571</td>
</tr>
<tr>
<td>0.17857</td>
<td>0.64286</td>
<td>0.50000</td>
</tr>
<tr>
<td>0.17857</td>
<td>1.03571</td>
<td>0.50000</td>
</tr>
<tr>
<td>0.21429</td>
<td>1.14286</td>
<td>0.66667</td>
</tr>
<tr>
<td>0.21429</td>
<td>0.82143</td>
<td>0.50000</td>
</tr>
<tr>
<td>0.25000</td>
<td>1.17857</td>
<td>0.33333</td>
</tr>
<tr>
<td>0.10714</td>
<td>0.78571</td>
<td>1.00000</td>
</tr>
<tr>
<td>0.10714</td>
<td>0.39286</td>
<td>1.00000</td>
</tr>
<tr>
<td>0.10714</td>
<td>0.39286</td>
<td>1.00000</td>
</tr>
<tr>
<td>0.07143</td>
<td>0.39286</td>
<td>1.00000</td>
</tr>
<tr>
<td>0.14286</td>
<td>0.64286</td>
<td>0.33333</td>
</tr>
<tr>
<td>0.14286</td>
<td>0.82143</td>
<td>0.66667</td>
</tr>
<tr>
<td>0.10714</td>
<td>0.50000</td>
<td>1.00000</td>
</tr>
<tr>
<td>0.14286</td>
<td>0.82143</td>
<td>0.66667</td>
</tr>
<tr>
<td>0.10714</td>
<td>0.82143</td>
<td>1.00000</td>
</tr>
<tr>
<td>0.10714</td>
<td>0.39286</td>
<td>1.00000</td>
</tr>
</tbody>
</table>
Example 1.7: Community Detection on Zachary's Karate Club Data

This example uses Zachary’s Karate Club data (Zachary 1977), which describes social network friendships between 34 members of a karate club at a U.S. university in the 1970s. This is one of the standard publicly available data sets for testing community detection algorithms. It contains 34 nodes and 78 links. The graph is shown in Figure 1.144.

![Zachary’s Karate Club Graph](image)

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

```plaintext
data LinkSetIn;
  input from to weight @@;
  datalines;
  0   9   1 0   10   1 0   14   1 0   15   1 0   16   1 0   19   1 0   20   1 0   21   1
  0   23  1 0   24   1 0   27   1 0   28   1 0   29   1 0   30   1 0   31   1 0   32   1
  0   33  1 2   1   1 3   1   1 3   2   1 4   1   1 4   2   1 4   3   1 5   1   1
  6   1   1   7   1   1   7   5   1   7   6   1   8   1   1   8   2   1   8   3   1   8   4   1
  9   1   1   9   3   1   10  3   1   11  1   1   11  5   1   11  6   1   12  1   1   13  1   1
  13  4   1 14   1   1   14  2   1   14  3   1   14  4   1   17  6   1   17  7   1   18  1   1
  18  2   1 20   1   1   20  2   1   22  1   1   22  2   1   26  24  1   26  25  1   28  3   1
  28  24  1   28  25  1   29  3   1   30  24  1   30  27  1   31  2   1   31  9   1   32  1   1
  32  25  1   32  26  1   32  29  1   33  3   1   33  9   1   33  15  1   33  16  1   33  19  1
  33  21  1   33  23  1   33  24  1   33  30  1   33  31  1   33  32  1
;```


The following statements use the RESOLUTION_LIST= option to represent resolution levels (1, 0.5) in community detection on the Karate Club data. For more information about resolution levels, see the section “Resolution List” on page 94.

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

The data set `NodeSetOut` contains the community identifier of each node. It is shown in **Output 1.7.1**.

**Output 1.7.1**  Community Nodes Output

<table>
<thead>
<tr>
<th>node</th>
<th>community_1</th>
<th>community_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>24</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>29</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node</th>
<th>community_1</th>
<th>community_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>33</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>25</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Column **community_1** contains the community identifier of each node when the resolution value is 1.0; column **community_2** contains the community identifier of each node when the resolution value is 0.5. Different node colors are used to represent different communities in Figure 1.145 and Figure 1.146. As you can see from the figures, four communities at resolution 1.0 are merged to two communities at resolution 0.5.
Figure 1.145  Karate Club Communities (Resolution = 1.0)

The data set CommLevelOut contains the number of communities and the corresponding modularity values found at each resolution level. It is shown in Output 1.7.2.
Example 1.7: Community Detection on Zachary’s Karate Club Data

**Output 1.7.2** Community Level Summary Output

<table>
<thead>
<tr>
<th>level</th>
<th>resolution</th>
<th>communities</th>
<th>modularity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>4</td>
<td>0.41880</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>2</td>
<td>0.37179</td>
</tr>
</tbody>
</table>

The data set CommOut contains the number of nodes contained in each community. It is shown in Output 1.7.3.

**Output 1.7.3** Community Number of Nodes Output

<table>
<thead>
<tr>
<th>level</th>
<th>resolution</th>
<th>community</th>
<th>nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>0</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>0</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>1</td>
<td>17</td>
</tr>
</tbody>
</table>

The data set CommOverlapOut contains the intensity of each node that belongs to multiple communities. It is shown in Output 1.7.4. Note that only the communities in the last resolution level (the smallest resolution value) are output in this data set. In this example, Node 0 belongs to two communities, with 82.3% of its links connecting to Community 0, and 17.6% of its links connecting to Community 1.

**Output 1.7.4** Community Overlap Output

<table>
<thead>
<tr>
<th>node</th>
<th>community</th>
<th>intensity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0.82353</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0.17647</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0.60000</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>0.40000</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0.50000</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0.50000</td>
</tr>
<tr>
<td>14</td>
<td>0</td>
<td>0.20000</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>0.80000</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>1.00000</td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td>1.00000</td>
</tr>
<tr>
<td>19</td>
<td>0</td>
<td>1.00000</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>0.33333</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>0.66667</td>
</tr>
<tr>
<td>21</td>
<td>0</td>
<td>1.00000</td>
</tr>
<tr>
<td>23</td>
<td>0</td>
<td>1.00000</td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td>1.00000</td>
</tr>
<tr>
<td>27</td>
<td>0</td>
<td>1.00000</td>
</tr>
<tr>
<td>28</td>
<td>0</td>
<td>0.75000</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>0.25000</td>
</tr>
<tr>
<td>29</td>
<td>0</td>
<td>0.66667</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>0.33333</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>node</th>
<th>community</th>
<th>intensity</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>0</td>
<td>1.00000</td>
</tr>
<tr>
<td>31</td>
<td>0</td>
<td>0.75000</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>0.25000</td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td>0.83333</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>0.16667</td>
</tr>
<tr>
<td>33</td>
<td>0</td>
<td>0.91667</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>0.08333</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0.11111</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.88889</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0.12500</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.87500</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0.40000</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0.60000</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>1.00000</td>
</tr>
</tbody>
</table>
Chapter 1: The OPTGRAPH Procedure

Output 1.7.4  continued

<table>
<thead>
<tr>
<th>node</th>
<th>community</th>
<th>intensity</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>22</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>26</td>
<td>0</td>
<td>1.00000</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
<td>1.00000</td>
</tr>
</tbody>
</table>

The data set CommLinksOut shows how the communities are interconnected. It is shown in Output 1.7.5. In this example, when the resolution value is 1, the link weight between Communities 0 and 1 is 7, and the link weight between Communities 1 and 2 is 4.

Output 1.7.5  Community Links Output

<table>
<thead>
<tr>
<th>level</th>
<th>resolution</th>
<th>from_community</th>
<th>to_community</th>
<th>link_weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.0</td>
<td>0</td>
<td>1</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>0</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>0</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

Example 1.8: Recursive Community Detection on Zachary’s Karate Club Data

This example illustrates the use of the RECURSIVE option in community detection on Zachary’s Karate Club data. The data set appears in “Example 1.7: Community Detection on Zachary’s Karate Club Data” on page 206. This example forces each community to contain no more than five nodes and the number of links between any pair of nodes within any community to be no greater than 2.

```
proc optgraph
  data_links = LinkSetIn
  out_nodes = NodeSetOut
  graph_internal_format = thin;
  community
    resolution_list = 1.0
    recursive (max_comm_size = 5 max_diameter = 2 relation = AND)
    out_community = CommOut;
run;
```

The data set NodeSetOut contains the community identifier of each node. It is shown in Output 1.8.1.
Example 1.8: Recursive Community Detection on Zachary’s Karate Club Data

Output 1.8.1 Community Nodes Output

<table>
<thead>
<tr>
<th>node</th>
<th>community_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>6</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>5</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>23</td>
<td>3</td>
</tr>
<tr>
<td>24</td>
<td>9</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>9</td>
</tr>
<tr>
<td>29</td>
<td>8</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>8</td>
</tr>
</tbody>
</table>

Output 1.8.2 Community Number of Nodes Output

<table>
<thead>
<tr>
<th>level resolution</th>
<th>community nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1</td>
<td>0 5</td>
</tr>
<tr>
<td>1 1</td>
<td>1 2</td>
</tr>
<tr>
<td>1 1</td>
<td>2 2</td>
</tr>
<tr>
<td>1 1</td>
<td>3 7</td>
</tr>
<tr>
<td>1 1</td>
<td>4 3</td>
</tr>
<tr>
<td>1 1</td>
<td>5 3</td>
</tr>
<tr>
<td>1 1</td>
<td>6 3</td>
</tr>
<tr>
<td>1 1</td>
<td>7 3</td>
</tr>
<tr>
<td>1 1</td>
<td>8 2</td>
</tr>
<tr>
<td>1 1</td>
<td>9 4</td>
</tr>
</tbody>
</table>

The data set CommOut contains the number of nodes contained in each community. It is shown in Output 1.8.2.

The community graph is shown in Figure 1.147, with different node shapes and colors representing different communities.
As you can see from Output 1.8.2, Community 3, whose nodes are drawn as black ellipses in Figure 1.147, contains seven nodes even though the maximum number of nodes in any community is set to be 5. This is because Community 3 has a symmetric shape: Nodes 0 and 33 are in the center, and they symmetrically connect to Nodes 21, 15, 19, 16, and 23. Therefore, this community cannot be further split.

**Example 1.9: Cycle Detection for Kidney Donor Exchange**

This example looks at an application of cycle detection to help create a kidney donor exchange. Suppose someone needs a kidney transplant and a family member is willing to donate one. If the donor and recipient are incompatible (because of blood types, tissue mismatch, and so on), the transplant cannot happen. Now suppose two donor-recipient pairs A and B are in this situation, but donor A is compatible with recipient B and donor B is compatible with recipient A. Then two transplants can take place in a two-way swap, shown graphically in Figure 1.148. More generally, an $n$-way swap can be performed involving $n$ donors and $n$ recipients (Willingham 2009).
To model this problem, define a directed graph as follows. Each node is an incompatible donor-recipient pair. Link \((i, j)\) exists if the donor from node \(i\) is compatible with the recipient from node \(j\). The link weight is a measure of the quality of the match. By introducing dummy links whose weight is 0, you can also include altruistic donors who have no recipients or recipients who have no donors. The idea is to find a maximum-weight node-disjoint union of directed cycles. You want the union to be node-disjoint so that no kidney is donated more than once, and you want cycles so that the donor from node \(i\) gives up a kidney if and only if the recipient from node \(i\) receives a kidney.

Without any other constraints, the problem could be solved as a linear assignment problem, as described in the section “Linear Assignment (Matching)” on page 116. But doing so would allow arbitrarily long cycles in the solution. Because of practical considerations (such as travel) and to mitigate risk, each cycle must have no more than \(L\) links. The kidney exchange problem is to find a maximum-weight node-disjoint union of short directed cycles.

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

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

```sas
/* create random graph on n nodes with arc probability p and uniform(0,1) weight */
%let n = 100;
%let p = 0.02;
data LinkSetIn;
do from = 0 to &n - 1;
do to = 0 to &n - 1;
   if from eq to then continue;
   else if ranuni(1) < &p then do;
      weight = ranuni(2);
      output;
   end;
end;
end;
run;
```
The following statements use PROC OPTGRAPH to generate all cycles whose length is greater than or equal to 2 and less than or equal to 10:

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

PROC OPTGRAPH finds 224 cycles of the appropriate length, as shown in Output 1.9.1.

**Output 1.9.1** Cycles for Kidney Donor Exchange PROC OPTGRAPH Log

```
NOTE: __________________________________________________________
NOTE: __________________________________________________________
NOTE: Running OPTGRAPH version 14.3.
NOTE: __________________________________________________________
NOTE: The OPTGRAPH procedure is executing in single-machine mode.
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 0.1 MBs (peak: 0.1 MBs) of memory.
NOTE: The number of nodes in the input graph is 97.
NOTE: The number of links in the input graph is 194.
NOTE: __________________________________________________________
NOTE: Processing cycle detection.
NOTE: The graph has 224 cycles.
NOTE: Processing cycle detection used 0.74 (cpu: 0.74) seconds.
NOTE: __________________________________________________________
NOTE: Creating cycle data set output.
NOTE: Data output used 0.01 (cpu: 0.00) seconds.
NOTE: __________________________________________________________
NOTE: The data set WORK.CYCLES has 2124 observations and 3 variables.
STATUS=OK  CYCLE=OK
STATUS=OK  NUM_CYCLES=224  CPU_TIME=0.74  REAL_TIME=0.74
```
From the resulting data set Cycles, use the following DATA step to convert the cycles into one observation per arc:

```plaintext
/* 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:

maximize \[ \sum_{c \in C} \left( \sum_{(i,j) \in A_c} w_{ij} \right) x_c \]

subject to \[ \sum_{c \in C, i \in N_c} x_c \leq 1 \quad i \in N \]
\[(\text{incomp\_pair})\]
\[ x_c \in \{0, 1\} \quad c \in C \]

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:

```plaintext
/* 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];
```
 Chapter 1: The OPTGRAPH Procedure

/* 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;

PROC OPTMODEL solves the problem by using the mixed integer linear programming solver. As shown in Output 1.9.2, it was able to find a total weight (quality level) of 26.02.

Output 1.9.2  Cycles for Kidney Donor Exchange PROC OPTMODEL Log

NOTE: There were 194 observations read from the data set WORK.LINKSETIN.
NOTE: There were 1900 observations read from the data set WORK.CYCLES0.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 224 variables (0 free, 0 fixed).
NOTE: The problem has 224 binary and 0 integer variables.
NOTE: The problem has 63 linear constraints (63 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 1900 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 46 variables and 35 constraints.
NOTE: The MILP presolver removed 901 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 178 variables, 28 constraints, and 999 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Sols</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>22.7780692</td>
<td>868.9019355</td>
<td>97.38%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>22.7780692</td>
<td>26.7803921</td>
<td>14.94%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>23.2747070</td>
<td>26.0966379</td>
<td>10.81%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>26.0202871</td>
<td>26.0202871</td>
<td>0.00%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
<td>26.0202871</td>
<td>26.0202871</td>
<td>0.00%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The MILP solver added 12 cuts with 673 cut coefficients at the root.
NOTE: Objective = 26.020287142.
NOTE: The data set WORK.SOLUTION has 6 observations and 2 variables.
STATUS=OK ALGORITHM=BC Solution_STATUS=OPTIMAL OBJECTIVE=26.020287142 RELATIVE_GAP=0
ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=2.220446E-16 BOUND_INFEASIBILITY=2.220446E-16
INTEGER_INFEASIBILITY=1.44329E-15 BEST_BOUND=26.020287142 NODES=1 ITERATIONS=98
PREsolve_TIME=0.02 SOLUTION_TIME=0.04

The data set Solution, shown in Output 1.9.3, now contains the cycles that define the best exchange and their associated weight (quality).
Example 1.10: Linear Assignment Problem for Minimizing Swim Times

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

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

```sas
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
;
```

```sas
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:

```sas
proc optgraph
  data_matrix = RelayTimesF;
  linear_assignment
    out = LinearAssignF
    id = (name sex);
run;
%put &_OPTGRAPH_
%put &_OPTGRAPH_LAP_
```

### Output 1.9.3: Maximum Quality Solution for Kidney Donor Exchange

<table>
<thead>
<tr>
<th>c</th>
<th>cycle_weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>5.84985</td>
</tr>
<tr>
<td>43</td>
<td>3.90015</td>
</tr>
<tr>
<td>71</td>
<td>5.44467</td>
</tr>
<tr>
<td>124</td>
<td>7.42574</td>
</tr>
<tr>
<td>222</td>
<td>2.28231</td>
</tr>
<tr>
<td>224</td>
<td>1.11757</td>
</tr>
</tbody>
</table>
proc optgraph
  data_matrix = RelayTimesM;
  linear_assignment
    out = LinearAssignM
    id = (name sex);
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_LAP_;

The progress of the two PROC OPTGRAPH calls is shown in Output 1.10.1 and Output 1.10.2.

Output 1.10.1  PROC OPTGRAPH Log: Linear Assignment for Female Swim Times

NOTE: _______________________________________________________________
NOTE: Running OPTGRAPH version 14.3.
NOTE: _______________________________________________________________
NOTE: The OPTGRAPH procedure is executing in single-machine mode.
NOTE: _______________________________________________________________
NOTE: The number of columns in the input matrix is 4.
NOTE: The number of rows in the input matrix is 6.
NOTE: The number of nonzeros in the input matrix is 24.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: _______________________________________________________________
NOTE: Processing the linear assignment problem.
NOTE: Objective = 111.5.
NOTE: Processing the linear assignment problem used 0.00 (cpu: 0.00) seconds.
NOTE: _______________________________________________________________
NOTE: Data output used 0.00 (cpu: 0.00) seconds.
NOTE: _______________________________________________________________
NOTE: The data set WORK.LINEARASSIGNF has 4 observations and 4 variables.
STATUS=OK  LAP=OPTIMAL
STATUS=OPTIMAL  OBJECTIVE=111.5  CPU_TIME=0.00  REAL_TIME=0.00
Example 1.10: Linear Assignment Problem for Minimizing Swim Times

Output 1.10.2 PROC OPTGRAPH Log: Linear Assignment for Male Swim Times

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

Output 1.10.3 Optimal Assignments for Best Female Swim Times

<table>
<thead>
<tr>
<th>name</th>
<th>sex</th>
<th>assign</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Karen</td>
<td>F</td>
<td>breast</td>
<td>32.6</td>
</tr>
<tr>
<td>Jan</td>
<td>F</td>
<td>fly</td>
<td>27.1</td>
</tr>
<tr>
<td>Carol</td>
<td>F</td>
<td>free</td>
<td>24.0</td>
</tr>
<tr>
<td>Ellen</td>
<td>F</td>
<td>back</td>
<td>27.8</td>
</tr>
</tbody>
</table>

111.5

Output 1.10.4 Optimal Assignments for Best Male Swim Times

<table>
<thead>
<tr>
<th>name</th>
<th>sex</th>
<th>assign</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jim</td>
<td>M</td>
<td>free</td>
<td>22.4</td>
</tr>
<tr>
<td>Mike</td>
<td>M</td>
<td>breast</td>
<td>24.0</td>
</tr>
<tr>
<td>Sam</td>
<td>M</td>
<td>back</td>
<td>27.2</td>
</tr>
<tr>
<td>Clayton</td>
<td>M</td>
<td>fly</td>
<td>23.0</td>
</tr>
</tbody>
</table>

96.6
Example 1.11: Linear Assignment Problem, Sparse Format versus Dense Format

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

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

Recall that the linear assignment problem can also be interpreted as the minimum-weight matching in a bipartite graph. The eligible assignments define links between the rows (swimmers) and the columns (strokes), as in Figure 1.149.

**Figure 1.149** Bipartite Graph for Linear Assignment Problem

You can represent the same data in RelayTimesMatrix by using a links data set as follows:
Example 1.11: Linear Assignment Problem, Sparse Format versus Dense Format

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

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

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

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

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

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

The data sets LinearAssignMatrix and LinearAssignLinks now contain the optimal assignments, as shown in Output 1.11.1 and Output 1.11.2.

**Output 1.11.1** Optimal Assignments for Swim Times (Dense Input)

<table>
<thead>
<tr>
<th>name</th>
<th>sex</th>
<th>assign</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>F</td>
<td>breast</td>
<td>36.7</td>
</tr>
<tr>
<td>Karen</td>
<td>F</td>
<td>free</td>
<td>26.2</td>
</tr>
<tr>
<td>Andrea</td>
<td>F</td>
<td>back</td>
<td>28.6</td>
</tr>
<tr>
<td>Carol</td>
<td>F</td>
<td>fly</td>
<td>26.6</td>
</tr>
</tbody>
</table>

118.1
Output 1.11.2  Optimal Assignments for Swim Times (Sparse Input)

<table>
<thead>
<tr>
<th>name</th>
<th>attr</th>
<th>cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sue</td>
<td>breast</td>
<td>36.7</td>
</tr>
<tr>
<td>Karen</td>
<td>free</td>
<td>26.2</td>
</tr>
<tr>
<td>Andrea</td>
<td>back</td>
<td>28.6</td>
</tr>
<tr>
<td>Carol</td>
<td>fly</td>
<td>26.6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>118.1</td>
</tr>
</tbody>
</table>

The optimal assignments are shown graphically in Figure 1.150.

Figure 1.150  Optimal Assignments for Swim Times

For large problems where a number of links are forbidden, the sparse format can be faster and can save a great deal of memory. Consider an example that uses the format of the DATA_MATRIX= option with 15,000 columns (|S| = 15, 000) and 4,000 rows (|T| = 4, 000). To store the dense matrix in memory, PROC OPTGRAPH needs to allocate approximately |S| · |T| · 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 · 4,000 · 0.05 = 3,000,000), then the dense storage would still need 457 MB. The sparse storage for the same example needs approximately |S| · |T| · 0.05 · 12/1024/1024 = 34 MB. If the problem is fully dense (all links are eligible), then the dense format that uses the DATA_MATRIX= option is the most efficient.
Example 1.12: Minimum Spanning Tree for Computer Network Topology

Consider again the small network of computers described in the section “Example 1.3: Betweenness and Closeness Centrality for Computer Network Topology” on page 194. Suppose that this network has not yet been formed, but for structural reasons the connections between the machines shown in Figure 1.142 are the only possible links. In designing the network, the goal is to make sure that each machine in the office can reach every other machine. To accomplish this goal, Ethernet lines must be constructed and run between the machines. The construction costs for each possible link are based approximately on distance and are shown in Figure 1.151. Besides distance, the costs also reflect some restrictions due to physical boundaries. To connect all the machines in the office at minimal cost, you need to find a minimum spanning tree on the network of possible links.

Figure 1.151 Potential Office Computer Network

 Define the link data set as follows:

```plaintext
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:

```plaintext
proc optgraph
   data_links = LinkSetInCompNet;
   minspantree
      out     = MinSpanTree;
run;
```
Output 1.12.1 shows the resulting data set MinSpanTree, which is displayed graphically in Figure 1.152 with the minimal cost links shown in green.

**Figure 1.152** Minimum Spanning Tree for Office Computer Network

Output 1.12.1 Minimum Spanning Tree of a Computer Network Topology

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>J</td>
<td>1.0</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>1.0</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>1.0</td>
</tr>
<tr>
<td>E</td>
<td>G</td>
<td>1.0</td>
</tr>
<tr>
<td>H</td>
<td>I</td>
<td>1.0</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>1.0</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>1.5</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
<td>1.5</td>
</tr>
<tr>
<td>C</td>
<td>H</td>
<td>4.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13.0</td>
</tr>
</tbody>
</table>

**Example 1.13: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System**

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

The bug tracking system needs to check for two situations when users declare a bug to be a DUP. The first situation is called a circular dependence. Consider bugs A, B, C, and D in the tracking system. The first user declares that A is a DUP of B and that C is a DUP of D. Then, a second user declares that B is a DUP of C, and a third user declares that D is a DUP of A. You now have a circular dependence, and no primary bug is
Example 1.13: Transitive Closure for Identification of Circular Dependencies

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 108. 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:

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

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

```plaintext
proc optgraph
  loglevel = moderate
  graph_direction = directed
  data_links = DefectLinks;
  data_links_var
    from = defectId
    to = linkedDefect;
  cycle
    out = Cycles
    mode = all_cycles;
  transitive_closure
    out = TransClosure;
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_CYCLE_;
%put &_OPTGRAPH_TRANSCL_;
```
proc sql;
create table Chains as
    select defectId, linkedDefect from TransClosure
    except
    select defectId, linkedDefect from DefectLinks;
quit;

The progress of the procedure is shown in Output 1.13.1.

Output 1.13.1 PROC OPTGRAPH Log: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System

NOTE: The OPTGRAPH procedure is executing in single-machine mode.
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 (peak: 0.0 MBs) of memory.
NOTE: The number of nodes in the input graph is 16.
NOTE: The number of links in the input graph is 12.
NOTE: Processing cycle detection.
NOTE: The graph has 1 cycle.
NOTE: Processing cycle detection used 0.00 (cpu: 0.00) seconds.
NOTE: Processing the transitive closure.
NOTE: Processing the transitive closure used 0.00 (cpu: 0.00) seconds.
NOTE: Creating cycle data set output.
NOTE: Creating transitive closure data set output.
NOTE: Data output used 0.01 (cpu: 0.00) seconds.
NOTE: The data set WORK.CYCLES has 4 observations and 3 variables.
NOTE: The data set WORK.TRANSCLOSE has 20 observations and 2 variables.
STATUS=OK CYCLE=OK TRANSCL=OK
STATUS=OK NUM_CYCLES=1 CPU_TIME=0.00 REAL_TIME=0.00
STATUS=OK CPU_TIME=0.00 REAL_TIME=0.00
NOTE: Table WORK.CHAINS created, with 8 rows and 2 columns.

The data set Cycles contains one case of a circular dependence in which the DUPs start and end at S0152674.
Example 1.14: Reach Networks for Computation of Market Coverage of a Terrorist Network

Output 1.13.2 Cycle in Bug Tracking System

<table>
<thead>
<tr>
<th>cycle</th>
<th>order</th>
<th>node</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>S0152674</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>S0153280</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>S0153307</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>S0152674</td>
</tr>
</tbody>
</table>

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

Output 1.13.3 Chains in Bug Tracking System

<table>
<thead>
<tr>
<th>defectId</th>
<th>linkedDefect</th>
</tr>
</thead>
<tbody>
<tr>
<td>S0152674</td>
<td>S0152674</td>
</tr>
<tr>
<td>S0152674</td>
<td>S0153307</td>
</tr>
<tr>
<td>S0153280</td>
<td>S0152674</td>
</tr>
<tr>
<td>S0153280</td>
<td>S0153280</td>
</tr>
<tr>
<td>S0153307</td>
<td>S0153280</td>
</tr>
<tr>
<td>S0153307</td>
<td>S0153307</td>
</tr>
<tr>
<td>S0162973</td>
<td>S0165405</td>
</tr>
<tr>
<td>S0325026</td>
<td>S0643230</td>
</tr>
</tbody>
</table>

Example 1.14: Reach Networks for Computation of Market Coverage of a Terrorist Network

The problem of finding an efficient method for covering a market (a set of entities) is important in numerous industries. For example, consider that you are an advertising company with access to data that are collected from your customers’ social networks. To keep costs at a minimum in some new promotion, you want to find a minimal set of customers to whom you need to advertise in order to reach the entire market. To solve this, you could first generate all the reach networks for each customer using PROC OPTGRAPH. These networks can then be used in a set-covering problem, which can be solved as an integer linear program using PROC OPTMODEL. Let $N$ be the set of customers that you want to reach, and let the links $A$ define the social network between those customers. If you use a one-hop reach network, you assume that if an advertisement is sent to customer $i$, then customer $i$ will promote the advertisement to all his friends (those he is connected to in $A$). If you use two-hop reach networks, you assume that customer $i$’s friends will also promote to their friends. So the question is: to which subset of customers should you advertise to reach all customers through the promotion mechanism?

This problem can be generalized as follows:

Given a graph $G = (N, A)$, choose a node set $N^*$ of minimal size such that there is a path of length less than or equal to $L$ to every node in $N$ from a node in $N^*$.

To illustrate an application of this problem, consider again the terrorist communications network from the section “Example 1.1: Articulation Points in a Terrorist Network” on page 188. In this case, customers are alleged terrorists. Solving the covering problem here can tell you a subset of people to focus on in an investigation in order to cover all members of the network.
The following `%GenerateReach` macro runs PROC OPTGRAPH to generate the reach network for each person in the terrorist network for a variable hop limit:

```sas
%macro GenerateReach(limit=);
proc optgraph;
  out_nodes = NodeSetOut
  data_links = LinkSetInTerror911;
  reach
    each_source
    out_nodes = ReachNode
    maxreach = &limit;
run;
%mend GenerateReach;
```

The following macro `%SolverCover` runs PROC OPTMODEL to solve the set-covering problem:

```sas
%macro SolverCover();
proc optmodel;
  string tmpLabel;
  set<num> NODE_ID;
  set<string> NODE_LABEL init {};
  string nodeIdToLabel(NODE_ID);
  num nodeLabelToId(NODE_LABEL);
  set<num> REACH_SET(NODE_ID) init {};
  set<string,num> PAIRS;
  /* read data */
  read data NodeSetOut into NODE_ID=[_n_] nodeIdToLabel=node;
  read data ReachNode into PAIRS=[node reach];
  for{i in NODE_ID} do;
    tmpLabel = nodeIdToLabel[i];
    NODE_LABEL = NODE_LABEL union {tmpLabel};
    nodeLabelToId[tmpLabel] = i;
  end;
  for{<label,i> in PAIRS} do;
    REACH_SET[i] = REACH_SET[i] union {nodeLabelToId[label]};
  end;
  /* declare decision variables */
  var x (NODE_ID) binary;
  /* declare objective */
  minimize numNodes = sum{j in NODE_ID} x[j];
  /* cover constraint */
  con cover {i in NODE_ID}:
    sum{j in REACH_SET[i]} x[j] >= 1;
  /* solve */
  solve;
  create data Solution from [label]=
    (setof{j in NODE_ID : round(x[j].sol)=1}nodeIdToLabel[j]);
```
Example 1.15: Traveling Salesman Tour through US Capital Cities

The following statements calculate the minimal cover for the one-hop limit:

%GenerateReach(limit=1);
%SolverCover();

In order to cover the network, assuming a one-hop limit, the investigators would need to investigate the people listed in the data set Solution, shown in Output 1.14.1.

**Output 1.14.1** Minimal One-Hop Cover for Terrorist Communications Network

<table>
<thead>
<tr>
<th>Obs</th>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Djamil Beghal</td>
</tr>
<tr>
<td>2</td>
<td>Zacarias Moussaoui</td>
</tr>
<tr>
<td>3</td>
<td>Essid Sami Ben Khemais</td>
</tr>
<tr>
<td>4</td>
<td>Mohamed Atta</td>
</tr>
<tr>
<td>5</td>
<td>Agus Budiman</td>
</tr>
<tr>
<td>6</td>
<td>Mamduh Mahmud Salim</td>
</tr>
<tr>
<td>7</td>
<td>Fayez Ahmed</td>
</tr>
<tr>
<td>8</td>
<td>Nawaf Alhazmi</td>
</tr>
<tr>
<td>9</td>
<td>Hani Hanjour</td>
</tr>
<tr>
<td>10</td>
<td>Nabil al-Marab</td>
</tr>
</tbody>
</table>

The following statements calculate the minimal cover for the two-hop limit:

%GenerateReach(limit=2);
%SolverCover();

If investigators assume a two-hop limit, they could focus their attention to the two people shown in Output 1.14.2. Then by following their links (and their links’ links) they could cover the entire network.

**Output 1.14.2** Minimal Two-Hop Cover for Terrorist Communications Network

<table>
<thead>
<tr>
<th>Obs</th>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Zacarias Moussaoui</td>
</tr>
<tr>
<td>2</td>
<td>Mohamed Atta</td>
</tr>
</tbody>
</table>

Example 1.15: Traveling Salesman Tour through US Capital Cities

Consider a cross-country trip where you want to travel the fewest miles to visit all of the capital cities in all US states except Alaska and Hawaii. Finding the optimal route is an instance of the traveling salesman problem, which is described in section “Traveling Salesman Problem” on page 165.

The following PROC SQL statements use the built-in data set maps.uscity to generate a list of the capital cities and their latitude and longitude:
/* Get a list of the state capital cities (with lat and long) */
proc sql;
create table Cities as
    select unique statecode as state, city, lat, long
    from maps.uscity
    where capital='Y' and statecode not in ('AK', 'PR', 'HI');
quit;

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

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

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

/* Find optimal tour using OPTGRAPH */
proc optgraph
    loglevel = moderate
    data_links = CitiesDist
    out_nodes = TSPTourNodes;
data_links_var
    from = city1
to = city2
    weight = distance;
tsp
    out = TSPTourLinks;
run;
%put &_OPTGRAPH_;
%put &_OPTGRAPH_TSP_;

The progress of the procedure is shown in Output 1.15.1. The total mileage needed to optimally traverse the capital cities is 10,627.75 miles.
Example 1.15: Traveling Salesman Tour through US Capital Cities

Output 1.15.1  PROC OPTGRAPH Log: Traveling Salesman Tour through US Capital Cities

NOTE: ---------------------------------------------------------------
NOTE: Running OPTGRAPH version 14.3.
NOTE: ---------------------------------------------------------------
NOTE: The OPTGRAPH procedure is executing in single-machine mode.
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 0.2 MBs (peak: 0.2 MBs) of memory.
NOTE: The number of nodes in the input graph is 49.
NOTE: The number of links in the input graph is 1176.
NOTE: ---------------------------------------------------------------
NOTE: Processing the traveling salesman problem.
NOTE: The initial TSP heuristics found a tour with cost 10645.918753 using 0.09 (cpu: 0.08)
NOTE: The MILP presolver value NONE is applied.
NOTE: The MILP solver is called.
NOTE: The Branch and Cut algorithm is used.

<table>
<thead>
<tr>
<th>Node</th>
<th>Active</th>
<th>Soln</th>
<th>BestInteger</th>
<th>BestBound</th>
<th>Gap</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10645.918753</td>
<td>10040.5139714</td>
<td>6.03%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10645.918753</td>
<td>10241.6970024</td>
<td>3.95%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10645.918753</td>
<td>10262.9074205</td>
<td>3.73%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10645.918753</td>
<td>10267.6251909</td>
<td>3.68%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10645.918753</td>
<td>10275.1505973</td>
<td>3.61%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10645.918753</td>
<td>10283.2134412</td>
<td>3.53%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10645.918753</td>
<td>10345.3151313</td>
<td>2.91%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10645.918753</td>
<td>10350.0790852</td>
<td>2.86%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10645.918753</td>
<td>10355.3956630</td>
<td>2.81%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10645.918753</td>
<td>10381.4538838</td>
<td>2.55%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10645.918753</td>
<td>10481.2454442</td>
<td>1.57%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10645.918753</td>
<td>10560.1837745</td>
<td>0.81%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10645.918753</td>
<td>10576.0823291</td>
<td>0.66%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10645.918753</td>
<td>10612.1627809</td>
<td>0.32%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>10645.918753</td>
<td>10619.6942572</td>
<td>0.25%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>10627.7543183</td>
<td>10627.7543183</td>
<td>0.00%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>10627.7543183</td>
<td>10627.7543183</td>
<td>0.00%</td>
<td>0</td>
</tr>
</tbody>
</table>

NOTE: The MILP solver added 16 cuts with 4818 cut coefficients at the root.
NOTE: Objective = 10627.754318.
NOTE: Processing the traveling salesman problem used 0.12 (cpu: 0.10) seconds.
NOTE: ---------------------------------------------------------------
NOTE: Creating nodes data set output.
NOTE: Creating traveling salesman data set output.
NOTE: Data output used 0.01 (cpu: 0.00) seconds.
NOTE: ---------------------------------------------------------------
The following PROC GPROJECT and PROC GMAP statements produce a graphical display of the solution:

```sas
/* 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 */
(data 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;
```
Example 1.15: Traveling Salesman Tour through US Capital Cities

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

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

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

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

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

The minimal cost tour through the capital cities is shown on the US map in Figure 1.15.2.
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:

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

The data set TSPTourLinksDirected is shown in Figure 1.153.
References


ABSOBJGAP= option
TSP statement, 45
ALGORITHM= option
COMMUNITY statement, 26
CONCOMP statement, 29
AUTH= option
CENTRALITY statement, 20
BETWEEN= option
CENTRALITY statement, 21
BETWEEN_NORM= option
CENTRALITY statement, 21
BICONCOMP option
SUMMARY statement, 42
BICONCOMP statement
statement options, 20
BY_CLUSTER option
CENTRALITY statement, 21
REACH statement, 39
SUMMARY statement, 43
CENTRALITY statement
statement options, 20
CLIQUE statement
statement options, 25
CLOSE= option
CENTRALITY statement, 21
CLOSE_NOPATH= option
CENTRALITY statement, 22
CLUSTER= option
DATA_NODES_VAR statement, 33
CLUSTERING_COEF option
CENTRALITY statement, 22
COMMUNITY statement
statement options, 26
CONCOMP option
SUMMARY statement, 43
CONCOMP statement
statement options, 29
CONFLICTSEARCH= option
TSP statement, 45
CORE statement
statement options, 30
CUTOFF= option
TSP statement, 46
CUTSTRATEGY= option
TSP statement, 46
CYCLE statement
statement options, 31
DATA_LINKS= option
PROC OPTGRAPH statement, 17
DATA_LINKS_VAR statement
statement options, 32
DATA_MATRIX= option
PROC OPTGRAPH statement, 17
DATA_NODERS= option
PROC OPTGRAPH statement, 17
DATA_NODERS_SUB= option
PROC OPTGRAPH statement, 18
DATA_NODERS_VAR statement
statement options, 33
DEGREE= option
CENTRALITY statement, 22
DETAILS option
PERFORMANCE statement, 38, 39
DIAMETER_APPROX= option
SUMMARY statement, 43
DIGRAPH option
REACH statement, 39
EACH_SOURCE option
REACH statement, 39
EIGEN= option
CENTRALITY statement, 23
EIGEN_ALGORITHM= option
CENTRALITY statement, 23
EIGEN_MAXITER= option
CENTRALITY statement, 23
EIGENVALUES= option
EIGENVECTOR statement, 34
EIGENVECTOR statement
statement options, 34
EMPHASIS= option
TSP statement, 46
FILTER_SUBGRAPH= option
PROC OPTGRAPH statement, 18
FROM= option
DATA_LINKS_VAR statement, 32
GRAPH_DIRECTION= option
PROC OPTGRAPH statement, 18
GRAPH_INTERNAL_FORMAT= option
PROC OPTGRAPH statement, 18
HEURISTICS= option
TSP statement, 46
HUB= option
CENTRALITY statement, 23
ID= option
LINEAR_ASSIGNMENT statement, 35
IGNORE_SELF option
REACH statement, 40
INCLUDE_SELFLINK option
PROC OPTGRAPH statement, 19
INFLUENCE= option
CENTRALITY statement, 24
LINEAR_ASSIGNMENT statement
statement options, 35
LINK_REMOVAL_RATIO= option
COMMUNITY statement, 26
LINKS= option
CORE statement, 30
LOGFREQ= option
MINCOSTFLOW statement, 36
SHORTPATH statement, 41
TSP statement, 47
LOGFREQNODE= option
CENTRALITY statement, 24
SUMMARY statement, 43
LOGFREQTIME= option
CENTRALITY statement, 24
REACH statement, 40
SUMMARY statement, 43
LOGLEVEL= option
BICONCOMP statement, 20
CENTRALITY statement, 24
CLIQUE statement, 25
COMMUNITY statement, 26
CONCOMP statement, 29
CORE statement, 30
CYCLE statement, 31
EIGENVECTOR statement, 34
LINEAR_ASSIGNMENT statement, 35
MINCOSTFLOW statement, 36
MINCUT statement, 37
MINSPANTREE statement, 38
PROC OPTGRAPH statement, 19
REACH statement, 40
SHORTPATH statement, 41
SUMMARY statement, 43
TRANSITIVE_CLOSURE statement, 45
TSP statement, 47
LOWER= option
DATA_LINKS_VAR statement, 33
MAXCLIQUES= option
CLIQUE statement, 26
MAXCYCLES= option
CYCLE statement, 31
MAXITER= option
COMMUNITY statement, 27
EIGENVECTOR statement, 35
MAXLENGTH= option
CYCLE statement, 31
MAXLINKWEIGHT= option
CYCLE statement, 31
MAXNODES= option
TSP statement, 47
MAXNODEWEIGHT= option
CYCLE statement, 31
MAXNUMCUTS= option
MINCUT statement, 37
MAXREACH= option
REACH statement, 40
MAXSOLS= option
TSP statement, 47
MAXTIME= number
MINCOSTFLOW statement, 37
MAXTIME= option
CLIQUE statement, 26
CORE statement, 30
CYCLE statement, 31
TSP statement, 47
MAXWEIGHT= option
MINCUT statement, 37
MILP= option
TSP statement, 48
MINCOSTFLOW statement
statement options, 36
MINCUT statement
statement options, 37
MINLENGTH= option
CYCLE statement, 32
MINLINKWEIGHT= option
CYCLE statement, 32
MINNODEWEIGHT= option
CYCLE statement, 32
MINSPANTREE statement
statement options, 38
MODULARITY= option
COMMUNITY statement, 28
NEIGEN= option
EIGENVECTOR statement, 35
NODE= option
DATA_NODES_VAR statement, 33
NODESEL= option
TSP statement, 48
OPTGRAPH Procedure, 10
PERFORMANCE statement, 38
OUT= option
CLIQUE statement, 26
CYCLE statement, 32
Index

- **EIGENVECTOR statement**, 35
- **LINEAR_ASSIGNMENT statement**, 36
- **MINCUT statement**, 37
- **MINSPANTREE statement**, 38
- **SHORTPATH statement**, 41
- **SUMMARY statement**, 44
- **TRANSITIVE_CLOSURE statement**, 45
- **TSP statement**, 48

**OUT_COMM_LINKS= option**
- **COMMUNITY statement**, 27

**OUT_COMMUNITY= option**
- **COMMUNITY statement**, 27

**OUT_COUNTS1= option**
- **REACH statement**, 40

**OUT_COUNTS2= option**
- **REACH statement**, 40

**OUT_COUNTS= option**
- **REACH statement**, 40

**OUT_LEVEL= option**
- **COMMUNITY statement**, 27

**OUT_LINKS= option**
- **PROC OPTGRAPH statement**, 19
- **REACH statement**, 40

**OUT_NODES= option**
- **PROC OPTGRAPH statement**, 19
- **REACH statement**, 40

**OUT_OVERLAP= option**
- **COMMUNITY statement**, 27

**OUT_PATHS= option**
- **SHORTPATH statement**, 41

**OUT_WEIGHTS= option**
- **SHORTPATH statement**, 41

**PATHS= option**
- **SHORTPATH statement**, 41

**PERFORMANCE statement**, 38

**DETAILS option**, 38, 39

**PROBE= option**
- **TSP statement**, 48

**PROC OPTGRAPH statement**
- **statement options**, 17

**RANDOM_FACTOR= option**
- **COMMUNITY statement**, 27

**RANDOM_SEED= option**
- **COMMUNITY statement**, 27

**REACH statement**
- **statement options**, 39

**RECURSIVE (options)**
- **COMMUNITY statement**, 27

**RELOBJGAP= option**
- **TSP statement**, 49

**RESOLUTION_LIST= option**
- **COMMUNITY statement**, 28

**SHORTPATH statement**
- **statement options**, 41

**SHORTPATH= option**
- **SUMMARY statement**, 44

**SINK= option**
- **SHORTPATH statement**, 42

**SOURCE= option**
- **SHORTPATH statement**, 42

**STANDARDIZED_LABELS option**
- **PROC OPTGRAPH statement**, 19

**STRONGITER= option**
- **TSP statement**, 49

**STRONGLEN= option**
- **TSP statement**, 49

**SUBSIZESWITCH= option**
- **CENTRALITY statement**, 25
- **SUMMARY statement**, 44

**SUMMARY statement**
- **statement options**, 42

**TARGET= option**
- **TSP statement**, 49

**TIMETYPE= option**
- **PROC OPTGRAPH statement**, 19

**TO= option**
- **DATA_LINKS_VAR statement**, 33

**TOLERANCE= option**
- **COMMUNITY statement**, 28

**TRANSITIVE_CLOSURE statement**
- **statement options**, 44

**TSP statement**
- **statement options**, 45

**UPPER= option**
- **DATA_LINKS_VAR statement**, 33

**USEWEIGHT= option**
- **SHORTPATH statement**, 42

**VARSEL= option**
- **TSP statement**, 49

**WEIGHT2= option**
- **CENTRALITY statement**, 25
- **DATA_NODES_VAR statement**, 34
- **SHORTPATH statement**, 42

**WEIGHT= option**
- **DATA_LINKS_VAR statement**, 33
- **DATA_NODES_VAR statement**, 34
- **LINEAR_ASSIGNMENT statement**, 36
Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore
for additional books and resources.