## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chapter 1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>Chapter 2</td>
<td>The NETWORK Procedure</td>
<td>7</td>
</tr>
</tbody>
</table>

## Index

213
Credits and Acknowledgments

Documentation

Writing Matthew Galati, Yi Liao
Editing Anne Baxter, Ed Huddleston
Documentation Support Tim Arnold, Melanie Gratton
Technical Review Charles B. Kelly, Manoj Chari, Ed Hughes, Rob Pratt, Anatoli Melechko, Yu-Min Lin

Software

PROC NETWORK Matthew Galati, Yi Liao, Anatoli Melechko

Support Groups

Software Testing Charles B. Kelly, Rui Kang, Yu-Min Lin, Elham Sadeghi
Technical Support Tonya Chapman
Chapter 1
Introduction

Overview of the NETWORK Procedure

The NETWORK procedure includes a number of graph theory and network analysis algorithms that can augment data mining and machine learning approaches. In many practical applications of data mining and machine learning models, pairwise interaction between the entities of interest in the model often plays an important role. For example, when you are modeling churn in a telecommunications network to support a retention campaign, the influence of individual customers on the other customers—such as friends and acquaintances that they regularly interact with—might contribute to the propensity of other customers to churn. You could likewise imagine a customer being able to influence the propensity of his or her acquaintances to acquire new products. Social networks such as Facebook and Twitter are obvious examples of networks that represent such interactions between individuals.

About This Book

This book assumes that you are familiar with Base SAS software and with the books SAS Language Reference: Concepts and SAS Visual Data Management and Utility Procedures Guide. It also assumes that you are familiar with basic SAS System concepts, such as using the DATA step to create SAS data sets and using Base SAS procedures (such as the PRINT and SORT procedures) to manipulate SAS data sets.
Chapter Organization

This book is organized as follows:

Chapter 1, this chapter, provides an overview of the NETWORK procedure, describes typographical conventions, and tells you where you can find more information.

Chapter 2 describes the NETWORK procedure and is organized as follows:

- The “Overview” section briefly describes the analysis provided by the procedure.
- The “Getting Started” section provides a quick introduction to the procedure through a simple example.
- The “Syntax” section describes the SAS statements and options that control the procedure.
- The “Details” section discusses methodology and other topics, such as ODS tables.
- The “Examples” section contains examples that use the procedure.
- The “References” section contains references for the methodology.

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:
Loading a SAS Data Set onto a CAS Server

Procedures in this book require the input data to reside on a CAS server. To work with a SAS data set, you must first load the data set onto the CAS server. Data loaded on the CAS server are called data tables. This section lists three methods of loading a SAS data set onto a CAS server. In this section, mycas is the name of the caslib that is connected to the mysess CAS session.

- You can use a single DATA step to create a data table on the CAS server as follows:

```sas
data mycas.Sample;
  input from $ to $ @@;
datalines;
A B A C B C
; 
```

Note that DATA step operations might not work as intended when you perform them on the CAS server instead of the SAS client.

- You can create a SAS data set first, and when it contains exactly what you want, you can use another DATA step to load it onto the CAS server as follows:

```sas
data Sample;
  input from $ to $ @@;
datalines;
A B A C B C
; 
data mycas.Sample;
  set Sample;
  run;
```

- You can use the CASUTIL procedure as follows:

```sas
proc casutil sessref=mysess;
  load data=Sample casout="Sample";
quit;
```

The CASUTIL procedure can load data onto a CAS server more efficiently than the DATA step. For more information about the CASUTIL procedure, see SAS Cloud Analytic Services: Language Reference.
The mycas caslib stores the Sample data table, which can be distributed across many machine nodes. You must use a caslib reference in procedures in this book to enable the SAS client machine to communicate with the CAS session. For example, the following NETWORK procedure statements use a data table that resides in the mycas caslib:

```sas
proc network links = mycas.Sample;
   ...statements...;
run;
```

You can delete your data table by using the DELETE procedure as follows:

```sas
proc delete data = mycas.Sample;
run;
```

The Sample data table is accessible only in the mysess session. When you terminate the mysess session, the Sample data table is no longer accessible from the CAS server. If you want your Sample data table to be available to other CAS sessions, then you must promote your data table. For more information about data tables, see *SAS Cloud Analytic Services: User's Guide*.

---

**Typographical Conventions**

This book uses several type styles for presenting information. The following list explains the meaning of the typographical conventions used in this book:

- **roman** is the standard type style used for most text.
- **UPPERCASE ROMAN** is used for SAS statements, options, and other SAS language elements when they appear in text. However, you can enter these elements in your own SAS programs in lowercase, uppercase, or a mixture of the two.
- **UPPERCASE BOLD** is used in the “Syntax” sections’ initial lists of SAS statements and options.
- **oblique** is used in the syntax definitions and in text to represent arguments for which you supply a value.
- **VariableName** is used for the names of variables and data sets when they appear in text.
- **bold** is used for matrices and vectors.
- **italic** is used for terms that are defined in text, for emphasis, and for references to publications.
- **monospace** is used for example code. In most cases, this book uses lowercase type for SAS code.

---

**Options Used in Examples**

The HTMLBLUE style is used to create the graphs and the HTML tables that appear in the online documentation. The PEARLJ style is used to create the PDF tables that appear in the documentation. A style template controls stylistic elements such as colors, fonts, and presentation attributes. You can specify a style template in an ODS destination statement as follows:
Most of the PDF tables are produced by using the following SAS System option:

```sas
options papersize=(6.5in 9in);
```

If you run the examples, you might get slightly different output. This is a function of the SAS System options that are used and the precision that your computer uses for floating-point calculations.

---

**Where to Turn for More Information**

**Online Documentation**

You can access the documentation by going to [http://support.sas.com/documentation](http://support.sas.com/documentation).

**SAS Technical Support Services**

The SAS Technical Support staff is available to respond to problems and answer technical questions regarding the use of procedures in this book. Go to [http://support.sas.com/techsup](http://support.sas.com/techsup) for more information.
Chapter 2
The NETWORK Procedure

Contents

Overview: NETWORK Procedure ................................................................. 9
Getting Started: NETWORK Procedure ......................................................... 10
  Road Network Shortest Path ................................................................. 10
  Authority in US Supreme Court Precedent .............................................. 13
Syntax: NETWORK Procedure ................................................................. 16
  Functional Summary .............................................................................. 17
  PROC NETWORK Statement ................................................................... 25
  BICONNECTEDCOMPONENTS Statement ................................................. 27
  BY Statement ......................................................................................... 28
  CENTRALITY Statement ......................................................................... 28
  CLIQUE Statement ................................................................................. 32
  COMMUNITY Statement ......................................................................... 33
  CONNECTEDCOMPONENTS Statement .................................................... 36
  CORE Statement ...................................................................................... 36
  CYCLE Statement ................................................................................... 37
  DISPLAY Statement ................................................................................ 38
  DISPLAYOUT Statement ......................................................................... 39
  LINKSVAR Statement ............................................................................... 40
  LINKSQUERYVAR Statement ................................................................... 40
  NODESIMILARITY Statement ................................................................... 41
  NODESSUBSETVAR Statement ................................................................ 43
  NODESVAR Statement ........................................................................... 43
  NODESQUERYVAR Statement ................................................................... 44
  PATH Statement ...................................................................................... 44
  PATTERNMATCH Statement .................................................................... 46
  REACH Statement .................................................................................. 46
  SHORTESTPATH Statement ..................................................................... 47
  SUMMARY Statement ................................................................................ 48
  TRANSITIVECLOSURE Statement ............................................................ 49
Details: NETWORK Procedure ................................................................. 49
  Graph Input Data ..................................................................................... 49
  Execution Modes and Data Movement .................................................... 59
  Numeric Limitations ............................................................................... 60
  Size Limitations ...................................................................................... 61
  Common Notation and Assumptions ....................................................... 62
  Biconnected Components and Articulation Points .................................. 62
Chapter 2: The NETWORK Procedure

Centrality ................................................................. 66
Clique Enumeration .................................................. 81
Community Detection ............................................... 84
Connected Components ........................................... 92
Core Decomposition .................................................. 97
Cycle Enumeration .................................................. 101
Node Similarity ......................................................... 105
Path Enumeration .................................................... 110
Pattern Matching ...................................................... 114
Reach (Ego) Network ................................................ 122
Shortest Path .......................................................... 129
Summary Statistics ................................................... 140
Transitive Closure ..................................................... 147
Macro Variable _NETWORK_ ........................................ 149
ODS Table Names ..................................................... 151
Examples: NETWORK Procedure ................................. 153
  Example 2.1: Articulation Points in a Terrorist Network ..... 153
  Example 2.2: Influence Centrality for Project Groups in a Research Department ... 155
  Example 2.3: Betweenness and Closeness Centrality for Computer Network Topology 159
  Example 2.4: Betweenness and Closeness Centrality for Project Groups in a Research Department ........ 162
  Example 2.5: Eigenvector Centrality for Word Sense Disambiguation .......... 165
  Example 2.6: Community Detection on Zachary’s Karate Club Data .......... 168
  Example 2.7: Recursive Community Detection on Zachary’s Karate Club Data .... 172
  Example 2.8: Centrality Metrics for a Simple Undirected Graph by Community .... 174
  Example 2.9: Cycle Enumeration for Kidney Donor Exchange ..................... 180
  Example 2.10: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System 185
  Example 2.11: Reach Networks for Computing the Market Coverage of a Terrorist Network ...................... 188
  Example 2.12: Connected Components for US Patent Citations ..................... 190
  Example 2.13: Shortest Paths of the New York Road Network ...................... 192
  Example 2.14: Shortest Path in a Road Network by Date and Time .............. 193
  Example 2.15: Pattern Matching in a Social Network .............................. 197
  Example 2.16: Node Similarity for Link Prediction ................................. 205
References .................................................................. 210
Overview: NETWORK Procedure

The NETWORK procedure includes a number of graph theory and network analysis algorithms that can augment data mining and machine learning approaches. In many practical applications of data mining and machine learning models, pairwise interaction between the entities of interest in the model often plays an important role. For example, when you are modeling churn in a telecommunications network to support a retention campaign, the influence of individual customers on the other customers—such as friends and acquaintances that they regularly interact with—might contribute to the propensity of other customers to churn. You could likewise imagine a customer being able to influence the propensity of his or her acquaintances to acquire new products. Social networks such as Facebook and Twitter are obvious examples of networks that represent such interactions between individuals.

Networks also appear explicitly and implicitly in many other application contexts. Networks are often constructed from certain natural co-occurrence types of relationships—such as relationships among researchers who coauthor articles, actors who appear in the same movie, words or topics that occur in the same document, items that appear together in a shopping basket, terrorism suspects who travel together or are seen in the same location, and so on. In these types of relationship, the strength or frequency of interaction is modeled as weights on the links of the resulting network.

To support the myriad ways in which networks appear in data mining, PROC NETWORK makes no assumptions about the context or application from which the network arises. It provides a number of network analysis algorithms (listed in Table 2.1) that take an abstract graph or network as input and help explain network structure and compute important network measures. Depending on the application, this type of network analysis can stand on its own and provide independent value, or it can support machine learning models—for example, by providing additional features that are derived from network measures such as node centrality.

### Table 2.1  Algorithm Classes in PROC NETWORK

<table>
<thead>
<tr>
<th>Algorithm Class</th>
<th>PROC NETWORK Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biconnected components</td>
<td>BICONNECTEDCOMPONENTS</td>
</tr>
<tr>
<td>Centrality</td>
<td>CENTRALITY</td>
</tr>
<tr>
<td>Clique enumeration</td>
<td>CLIQUE</td>
</tr>
<tr>
<td>Community detection</td>
<td>COMMUNITY</td>
</tr>
<tr>
<td>Connected components</td>
<td>CONNECTEDCOMPONENTS</td>
</tr>
<tr>
<td>Core decomposition</td>
<td>CORE</td>
</tr>
<tr>
<td>Cycle enumeration</td>
<td>CYCLE</td>
</tr>
<tr>
<td>Node similarity</td>
<td>NODESIMILARITY</td>
</tr>
<tr>
<td>Path enumeration</td>
<td>PATH</td>
</tr>
<tr>
<td>Pattern matching</td>
<td>PATTERNMATCH</td>
</tr>
<tr>
<td>Reach (ego) 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>TRANSITIVECLOSURE</td>
</tr>
</tbody>
</table>
Chapter 2: The NETWORK Procedure

As input, the NETWORK procedure expects graph \( G = (N, A) \), which 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 the relationship (or connection) between two nodes. The terms node and vertex are interchangeable in describing an entity. The term arc is interchangeable with the term edge or link in describing a connection. Similarly, the terms graph and network are interchangeable.

Getting Started: NETWORK Procedure

Because 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 chapter demonstrates a few potential applications through simple examples.

This section presents two introductory examples for getting started with the NETWORK procedure. For more information about the expected input formats and the available algorithms, see the sections “Details: NETWORK Procedure” on page 49 and “Examples: NETWORK Procedure” on page 153.

Road Network Shortest Path

Consider the following road network between a SAS employee’s home in Raleigh, North Carolina, and SAS headquarters nearby in Cary. In this road network (graph), the links are the roads and the nodes are intersections of the 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 travel between two nodes based on the length of the road and the typical speed during normal traffic patterns. These statements assume that the CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```r
data mycas.LinkSetInRoadNC10am;
  input start_inter $1-20 end_inter $21-40 miles miles_per_hour;
  time_to_travel = miles * 1/miles_per_hour * 60;
datalines;
614CapitalBlvd Capital/WadeAve 0.6 25
614CapitalBlvd Capital/US70W 0.6 25
614CapitalBlvd Capital/US440W 3.0 45
Capital/WadeAve WadeAve/RaleighExpy 3.0 40
Capital/US70W US70W/US440W 3.2 60
US70W/US440W US440W/RaleighExpy 2.7 60
Capital/US440W US440W/RaleighExpy 6.7 60
US440W/RaleighExpy RaleighExpy/US40W 3.0 60
WadeAve/RaleighExpy RaleighExpy/US40W 3.0 60
RaleighExpy/US40W US40W/HarrisonAve 1.3 55
US40W/HarrisonAve SASCampusDrive 0.5 25;
```

Using PROC NETWORK, you want to find the route that yields the shortest path between home (614 Capital Boulevard) and SAS headquarters (SAS Campus Drive). This can be done using the SHORTESTPATH statement as follows:
proc network
links = mycas.LinkSetInRoadNC10am;
linksVar
  from = start_inter
to = end_inter
weight = time_to_travel;
shortestPath
  outPaths = mycas.ShortPath
  source = "614CapitalBlvd"
sink = "SASCampusDrive";
run;

For more information about shortest path algorithms in PROC NETWORK, see the section “Shortest Path” on page 129. Figure 2.1 displays the output data table mycas.ShortPath, which shows the best route to take to minimize travel time at 10:00 a.m. on a workday. This route is also shown in Google Maps in Figure 2.2.

<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>
</tbody>
</table>

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

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

Now suppose that it is the evening rush hour (5:00–7:00 p.m.) and the time that it takes to travel this route 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 because of rush-hour traffic. The following data table lists approximate travel times and speeds for driving in the opposite direction:
data mycas.LinkSetInRoadNC5pm;
  input start_inter $1-20 end_inter $21-40 miles miles_per_hour;
  time_to_travel = miles * 1/miles_per_hour * 60;
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
;

The following statements are similar to those in the first PROC NETWORK run, except that they use the data table mycas.LinkSetInRoadNC5pm and the SOURCE= and SINK= option values are reversed:

    proc network
      links = mycas.LinkSetInRoadNC5pm;
      linksVar
        from = start_inter
        to = end_inter
        weight = time_to_travel;
      shortestPath
        outPaths = mycas.ShortPath
        source = "SASCampusDrive"
        sink = "614CapitalBlvd";
    run;

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

**Figure 2.3** Shortest Path for Road Network at 5:00 P.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>SASCampusDrive</td>
<td>US40W/HarrisonAve</td>
<td>1.2000</td>
</tr>
<tr>
<td>2</td>
<td>US40W/HarrisonAve</td>
<td>RaleighExpy/US40W</td>
<td>1.4182</td>
</tr>
<tr>
<td>3</td>
<td>RaleighExpy/US40W</td>
<td>US440W/RaleighExpy</td>
<td>3.0000</td>
</tr>
<tr>
<td>4</td>
<td>US440W/RaleighExpy</td>
<td>US70W/US440W</td>
<td>2.7000</td>
</tr>
<tr>
<td>6</td>
<td>Capital/US70W</td>
<td>614CapitalBlvd</td>
<td>1.4400</td>
</tr>
</tbody>
</table>

This new route is shown in Google Maps in Figure 2.4.
This example looks at the use of precedents in cases before the US Supreme Court. 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 precedents in order to take advantage of “the 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 2.5 shows a small representative subset of the citation network for landmark abortion decisions from the example in Fowler and Joen (2008).
The data table mycas.Cases stores a mapping between the case name and the case identifier:

```software
data mycas.Cases;
   length case_name $80;
   input case_id 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 table mycas.LinkSetInCourt provides the citation network between case identifiers:

```software
data mycas.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 ;```

The data table mycas.Cases stores a mapping between the case name and the case identifier:

```software
data mycas.Cases;
   length case_name $80;
   input case_id 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)
;```
You can calculate the authority scores of each case by using the CENTRALITY statement with the AUTH= option, as follows:

```sas
proc network
direction = directed
links = mycas.LinkSetInCourt
outNodes = mycas.NodeSetOut;
linksVar
    from = from_case
    to = to_case;
centrality
    auth = unweight;
run;
```

The output data table `mycas.NodeSetOut` contains the authority score for each case (node). Then, the following statements combine the case names and the case identifiers into a local data set called `NodeSetOut` and sort them by score in descending order:

```sas
data NodeSetOut(drop=rc);
    merge mycas.NodeSetOut(rename=(node=case_id)) mycas.Cases;
    by case_id;
run;
proc sort data=NodeSetOut;
    by descending centr_auth_unwt;
run;
```

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

**Figure 2.6** Authority Ranking of Landmark US Supreme Court Cases

<table>
<thead>
<tr>
<th>case_id</th>
<th>centr_auth_unwt</th>
<th>case_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>25347</td>
<td>1.000000</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 Ctr 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>29663</td>
<td>0.00000</td>
<td>Madsen v. Women's Health Ctr., 512 U.S. 753 (1994)</td>
</tr>
<tr>
<td>29933</td>
<td>0.00000</td>
<td>Wash. v. Glucksberg, 521 U.S. 702 (1997)</td>
</tr>
<tr>
<td>29155</td>
<td></td>
<td>Georgia v. South Carolina, 497 U.S. 376 (1990)</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 in examining all the citations for all 30,288 cases available in their data.
Syntax: NETWORK Procedure

PROC NETWORK statements are divided into four main categories:

**PROC Statement**

```
PROC NETWORK <options> ;
```

The PROC statement invokes the procedure and sets option values that are used across multiple algorithms.

**Data Input Statements**

```
LINKSVAR <options> ;
LINKSQUERYVAR <options> ;
NODESVAR <options> ;
NODESQUERYVAR <options> ;
NODESSUBSETVAR <options> ;
```

Data input statements control the names of the variables that PROC NETWORK expects in the data input.

**Algorithm Statements**

```
BICONNECTEDCOMPONENTS ;
CENTRALITY <options> ;
CLIQUE <options> ;
COMMUNITY <options> ;
CONNECTEDCOMPONENTS <options> ;
CORE <options> ;
CYCLE <options> ;
NODESIMILARITY <options> ;
PATH <options> ;
PATTERNMATCH <options> ;
REACH <options> ;
SHORTESTPATH <options> ;
SUMMARY <options> ;
TRANSITIVECLOSURE <options> ;
```

Algorithm statements control the names of the variables that PROC NETWORK expects in the data input.

**Standard Statements**

```
BY variables ;
DISPLAY <table-list> <options> ;
DISPLAYOUT table-spec-list <options> ;
```

Standard statements control BY-group processing and manage ODS tables.

The following section provides a quick summary of each statement and its options. Each statement is then described in more detail in its own section. The PROC NETWORK statement is described first, and sections that describe all the other statements are presented in alphabetical order (they are not ordered according to their category).
Table 2.2 summarizes the statements and options available in the NETWORK procedure.

<table>
<thead>
<tr>
<th>Description</th>
<th>Statement</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the links data table</td>
<td>PROC NETWORK</td>
<td>LINKS=</td>
</tr>
<tr>
<td>Specifies the links data table to query</td>
<td>PROC NETWORK</td>
<td>LINKSQUERY=</td>
</tr>
<tr>
<td>Specifies the nodes data table</td>
<td>PROC NETWORK</td>
<td>NODES=</td>
</tr>
<tr>
<td>Specifies the nodes data table to query</td>
<td>PROC NETWORK</td>
<td>NODESQUERY=</td>
</tr>
<tr>
<td>Specifies the nodes subset data table</td>
<td>PROC NETWORK</td>
<td>NODESSETSUBSET=</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the links output data table</td>
<td>PROC NETWORK</td>
<td>OUTLINKS=</td>
</tr>
<tr>
<td>Specifies the nodes output data table</td>
<td>PROC NETWORK</td>
<td>OUTNODES=</td>
</tr>
<tr>
<td><strong>Options</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the graph direction</td>
<td>PROC NETWORK</td>
<td>DIRECTION=</td>
</tr>
<tr>
<td>Specifies whether to use a distributed graph</td>
<td>PROC NETWORK</td>
<td>DISTRIBUTED=</td>
</tr>
<tr>
<td>Includes self-links</td>
<td>PROC NETWORK</td>
<td>INCLUDESELFINK=</td>
</tr>
<tr>
<td>Specifies the index offset for identifiers</td>
<td>PROC NETWORK</td>
<td>INDEXOFFSET=</td>
</tr>
<tr>
<td>Specifies the desired frequency (in number of seconds) between log entries</td>
<td>PROC NETWORK</td>
<td>LOGFREQTIME=</td>
</tr>
<tr>
<td>Specifies the overall log level</td>
<td>PROC NETWORK</td>
<td>LOGLEVEL=</td>
</tr>
<tr>
<td>Specifies the maximum number of threads to use for multithreaded processing</td>
<td>PROC NETWORK</td>
<td>NTHREADS=</td>
</tr>
<tr>
<td>Specifies that the input graph data are in a standardized format</td>
<td>PROC NETWORK</td>
<td>STANDARDIZEDLABELS=</td>
</tr>
<tr>
<td>Specifies whether time units are in CPU time or real time</td>
<td>PROC NETWORK</td>
<td>TIMETYPE=</td>
</tr>
<tr>
<td><strong>Data Input Statements</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the data variable name for the auxiliary link weights</td>
<td>LINKSVAR</td>
<td>AUXWEIGHT=</td>
</tr>
<tr>
<td>Specifies the data variable name for the from nodes</td>
<td>LINKSVAR</td>
<td>FROM=</td>
</tr>
<tr>
<td>Specifies the data variable name for the to nodes</td>
<td>LINKSVAR</td>
<td>TO=</td>
</tr>
<tr>
<td>Specifies the data variable(s) for the additional link attributes to carry over to the output results</td>
<td>LINKSVAR</td>
<td>VARS=</td>
</tr>
<tr>
<td>Specifies the data variable name for the link weights</td>
<td>LINKSVAR</td>
<td>WEIGHT=</td>
</tr>
<tr>
<td>Description</td>
<td>Statement</td>
<td>Option</td>
</tr>
<tr>
<td>----------------------------------------------------------------------------</td>
<td>--------------------</td>
<td>--------------</td>
</tr>
<tr>
<td>Specifies the data variable name for the <em>from</em> nodes in the query graph</td>
<td>LINKSQUERYVAR FROM=</td>
<td></td>
</tr>
<tr>
<td>Specifies the data variable name for the <em>to</em> nodes in the query graph</td>
<td>LINKSQUERYVAR TO=</td>
<td></td>
</tr>
<tr>
<td>Specifies the data variable(s) for the link attributes to consider in the query graph</td>
<td>LINKSQUERYVAR VARS=</td>
<td></td>
</tr>
<tr>
<td>Specifies the data variable name for the nodes</td>
<td>NODESVAR NODE=</td>
<td></td>
</tr>
<tr>
<td>Specifies the data variable(s) for the additional node attributes to carry over to the output results</td>
<td>NODESVAR VARS=</td>
<td></td>
</tr>
<tr>
<td>Specifies the data variable name for the node weights</td>
<td>NODESVAR WEIGHT=</td>
<td></td>
</tr>
<tr>
<td>Specifies the data variable name for the nodes in the query graph</td>
<td>NODESQUERYVAR NODE=</td>
<td></td>
</tr>
<tr>
<td>Specifies the data variable(s) for the node attributes to consider in the query graph</td>
<td>NODESQUERYVAR VARS=</td>
<td></td>
</tr>
<tr>
<td>Specifies the data variable name for the nodes</td>
<td>NODESSUBSETVAR NODE=</td>
<td></td>
</tr>
<tr>
<td>Specifies the data variable name for the reach identifier</td>
<td>NODESSUBSETVAR REACH=</td>
<td></td>
</tr>
<tr>
<td>Specifies the data variable name for the sink indicator</td>
<td>NODESSUBSETVAR SINK=</td>
<td></td>
</tr>
<tr>
<td>Specifies the data variable name for the source indicator</td>
<td>NODESSUBSETVAR SOURCE=</td>
<td></td>
</tr>
</tbody>
</table>

**Algorithm Statements**

**CENTRALITY Statement**

- Specifies which type of authority centrality to calculate: CENTRALITY AUTH=
- Specifies which type of betweenness centrality to calculate: CENTRALITY BETWEEN=
- Specifies whether to normalize the betweenness calculation: CENTRALITY BETWEENNORM=
- Specifies which type of closeness centrality to calculate: CENTRALITY CLOSE=
- Specifies a method for accounting for the shortest path distance between two nodes when a path does not exist (disconnected nodes): CENTRALITY CLOSENOPATH=
- Calculates the node clustering coefficient: CENTRALITY CLUSTERINGCOEF=
- Calculates degree centrality: CENTRALITY DEGREE=
## Table 2.2 (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Statement</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies which type of eigenvector centrality to calculate</td>
<td>CENTRALITY</td>
<td>EIGEN=</td>
</tr>
<tr>
<td>Specifies the algorithm to use for eigenvector calculation</td>
<td>CENTRALITY</td>
<td>EIGENALGORITHM=</td>
</tr>
<tr>
<td>Specifies the maximum number of iterations for eigenvector calculation</td>
<td>CENTRALITY</td>
<td>EIGENMAXITERS=</td>
</tr>
<tr>
<td>Specifies which type of hub centrality to calculate</td>
<td>CENTRALITY</td>
<td>HUB=</td>
</tr>
<tr>
<td>Specifies which type of influence centrality to calculate</td>
<td>CENTRALITY</td>
<td>INFLUENCE=</td>
</tr>
<tr>
<td>Specifies the internal graph format</td>
<td>CENTRALITY</td>
<td>INTERNALFORMAT=</td>
</tr>
<tr>
<td>Specifies which type of PageRank centrality metric to calculate</td>
<td>CENTRALITY</td>
<td>PAGERANK=</td>
</tr>
<tr>
<td>Specifies the damping factor for the PageRank algorithm</td>
<td>CENTRALITY</td>
<td>PAGERANKALPHA=</td>
</tr>
<tr>
<td>Specifies the convergence tolerance for the PageRank algorithm</td>
<td>CENTRALITY</td>
<td>PAGERANKTOLERANCE=</td>
</tr>
</tbody>
</table>

### CLIQUE Statement

| Specifies the maximum number of cliques to return during clique enumeration | CLIQUE   | MAXCLIQUES=                |
| Specifies the maximum link weight for the cliques found                    | CLIQUE   | MAXLINKWEIGHT=             |
| Specifies the maximum node weight for the cliques found                    | CLIQUE   | MAXNODEWEIGHT=             |
| Specifies the maximum size for the cliques found                           | CLIQUE   | MAXSIZE=                   |
| Specifies the maximum amount of time to spend finding cliques              | CLIQUE   | MAXTIME=                   |
| Specifies the minimum link weight for the cliques found                    | CLIQUE   | MINLINKWEIGHT=             |
| Specifies the minimum node weight for the cliques found                    | CLIQUE   | MINNODEWEIGHT=             |
| Specifies the minimum size for the cliques found                           | CLIQUE   | MINSIZE=                   |
| Specifies the output data table for cliques                                | CLIQUE   | OUT=                       |

### COMMUNITY Statement

| Specifies the community detection algorithm                                | COMMUNITY| ALGORITHM=                 |
| Specifies the internal graph format                                       | COMMUNITY| INTERNALFORMAT=           |
| Specifies the percentage of small-weight links to be removed               | COMMUNITY| LINKREMOVALRATIO=         |
### Table 2.2 (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Statement</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies the maximum number of iterations for community detection</td>
<td>COMMUNITY</td>
<td>MAXITERS=</td>
</tr>
<tr>
<td>Specifies the output data table for intercommunity links</td>
<td>COMMUNITY</td>
<td>OUTCOMMLINKS=</td>
</tr>
<tr>
<td>Specifies the output data table for the community summary</td>
<td>COMMUNITY</td>
<td>OUTCOMMUNITY=</td>
</tr>
<tr>
<td>Specifies the output data table for the community level summary</td>
<td>COMMUNITY</td>
<td>OUTLEVEL=</td>
</tr>
<tr>
<td>Specifies the output data table for the community overlap</td>
<td>COMMUNITY</td>
<td>OUTOVERLAP=</td>
</tr>
<tr>
<td>Specifies the random factor in the parallel label propagation algorithm</td>
<td>COMMUNITY</td>
<td>RANDOMFACTOR=</td>
</tr>
<tr>
<td>Specifies the random seed for the parallel label propagation algorithm</td>
<td>COMMUNITY</td>
<td>RANDOMSEED=</td>
</tr>
<tr>
<td>Applies the recursive option to break large communities</td>
<td>COMMUNITY</td>
<td>RECURSIVE</td>
</tr>
<tr>
<td>Specifies the resolution list for community detection</td>
<td>COMMUNITY</td>
<td>RESOLUTIONLIST=</td>
</tr>
<tr>
<td>Specifies the modularity tolerance value for community detection</td>
<td>COMMUNITY</td>
<td>TOLERANCE=</td>
</tr>
<tr>
<td>CONNECTEDCOMPONENTS Statement</td>
<td>CONNECTEDCOMPONENTS</td>
<td>ALGORITHM=</td>
</tr>
<tr>
<td>Specifies the algorithm to use for connected components</td>
<td>CONNECTEDCOMPONENTS</td>
<td>INTERNALFORMAT=</td>
</tr>
<tr>
<td>CORE Statement</td>
<td>CORE</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Specifies the maximum amount of time to spend calculating the core decomposition</td>
<td>CORE</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>CYCLE Statement</td>
<td>CYCLE</td>
<td>ALGORITHM=</td>
</tr>
<tr>
<td>Specifies the maximum number of cycles to return during cycle enumeration</td>
<td>CYCLE</td>
<td>MAXCYCLES=</td>
</tr>
<tr>
<td>Specifies the maximum length for the cycles found</td>
<td>CYCLE</td>
<td>MAXLENGTH=</td>
</tr>
<tr>
<td>Specifies the maximum link weight for the cycles found</td>
<td>CYCLE</td>
<td>MAXLINKWEIGHT=</td>
</tr>
<tr>
<td>Specifies the maximum node weight for the cycles found</td>
<td>CYCLE</td>
<td>MAXNODEWEIGHT=</td>
</tr>
<tr>
<td>Specifies the maximum amount of time to spend finding cycles</td>
<td>CYCLE</td>
<td>MAXTIME=</td>
</tr>
<tr>
<td>Description</td>
<td>Statement</td>
<td>Option</td>
</tr>
<tr>
<td>-----------------------------------------------------------------------------</td>
<td>-----------</td>
<td>-------------------------------</td>
</tr>
<tr>
<td>Specifies the minimum length for the cycles found</td>
<td>CYCLE</td>
<td>MINLENGTH=</td>
</tr>
<tr>
<td>Specifies the minimum link weight for the cycles found</td>
<td>CYCLE</td>
<td>MINLINKWEIGHT=</td>
</tr>
<tr>
<td>Specifies the minimum node weight for the cycles found</td>
<td>CYCLE</td>
<td>MINNODEWEIGHT=</td>
</tr>
<tr>
<td>Specifies the output data table for cycles</td>
<td>CYCLE</td>
<td>OUT=</td>
</tr>
</tbody>
</table>

**NODESIMILARITY Statement**

| Specifies the internal graph format                                        | NODESIMILARITY | INTERNALFORMAT=               |
| Specifies the Jaccard similarity algorithm                                  | NODESIMILARITY | JACCARD=                      |
| Specifies the maximum similarity value to output in the OUTSIMILARITY= table | NODESIMILARITY | MAXSCORE=                     |
| Specifies the minimum similarity value to output in the OUTSIMILARITY= table | NODESIMILARITY | MINSCORE=                     |
| Specifies the number of dimensions for vector embeddings                   | NODESIMILARITY | NDIMENSIONS=                  |
| Specifies the number of negative samples per sample used in training the model | NODESIMILARITY | NEGATIVESAMPLEFACTOR=         |
| Specifies the number of training samples for the vector algorithm           | NODESIMILARITY | NSAMPLES=                     |
| Specifies the output data table similarity between pairs of nodes           | NODESIMILARITY | OUTSIMILARITY=                |
| Specifies the proximity order for the vector similarity algorithm           | NODESIMILARITY | PROXIMITYORDER=               |
| Specifies the sink node for similarity calculations                        | NODESIMILARITY | SINK=                         |
| Specifies the source node for similarity calculations                      | NODESIMILARITY | SOURCE=                       |
| Specifies the vector similarity algorithm                                   | NODESIMILARITY | VECTOR=                       |

**PATH Statement**

| Specifies the internal graph format                                        | PATH     | INTERNALFORMAT=               |
| Specifies the maximum length for the paths found                           | PATH     | MAXLENGTH=                    |
| Specifies the maximum link weight for the paths found                       | PATH     | MAXLINKWEIGHT=                 |
| Specifies the maximum node weight for the paths found                       | PATH     | MAXNODEWEIGHT=                 |
| Specifies the maximum amount of time to spend finding paths                 | PATH     | MAXTIME=                      |
| Specifies the minimum length for the paths found                           | PATH     | MINLENGTH=                    |
### Table 2.2 (continued)

<table>
<thead>
<tr>
<th>Description</th>
<th>Statement</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Specifies the minimum link weight for the paths found</td>
<td>PATH</td>
<td>MINLINKWEIGHT=</td>
</tr>
<tr>
<td>Specifies the minimum node weight for the paths found</td>
<td>PATH</td>
<td>MINNODEWEIGHT=</td>
</tr>
<tr>
<td>Specifies the output data table for path links</td>
<td>PATH</td>
<td>OUTPATHSLINKS=</td>
</tr>
<tr>
<td>Specifies the output data table for path nodes</td>
<td>PATH</td>
<td>OUTPATHSNODES=</td>
</tr>
<tr>
<td>Specifies the sink node for path calculations</td>
<td>PATH</td>
<td>SINK=</td>
</tr>
<tr>
<td>Specifies the source node for path calculations</td>
<td>PATH</td>
<td>SOURCE=</td>
</tr>
<tr>
<td><strong>PATTERNMATCH Statement</strong></td>
<td>PATTERNMATCH</td>
<td>OUTMATCHLINKS=</td>
</tr>
<tr>
<td>Specifies the links output data table for matching subgraphs</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Specifies the nodes output data table for matching subgraph mappings</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>REACH Statement</strong></td>
<td>REACH</td>
<td>DIGRAPHT</td>
</tr>
<tr>
<td>Calculates the directed reach counts</td>
<td>REACH</td>
<td>EACHSOURCE</td>
</tr>
<tr>
<td>Treats each node as a source in reach calculations</td>
<td>REACH</td>
<td>MAXREACH=</td>
</tr>
<tr>
<td>Specifies the maximum number of links in the reach calculations</td>
<td>REACH</td>
<td>OUTCOUNTS=</td>
</tr>
<tr>
<td>Specifies the output data table for reach counts</td>
<td>REACH</td>
<td>OUTREACHLINKS=</td>
</tr>
<tr>
<td>Specifies the output data table for reach links</td>
<td>REACH</td>
<td>OUTREACHNODES=</td>
</tr>
<tr>
<td><strong>SHORTESTPATH Statement</strong></td>
<td>SHORTESTPATH</td>
<td>MAXPATHWEIGHT=</td>
</tr>
<tr>
<td>Specifies the maximum path weight</td>
<td>SHORTESTPATH</td>
<td>OUTPATHS=</td>
</tr>
<tr>
<td>Specifies the output data table for shortest paths</td>
<td>SHORTESTPATH</td>
<td>OUTWEIGHTS=</td>
</tr>
<tr>
<td>Specifies the output data table for shortest path summaries</td>
<td>SHORTESTPATH</td>
<td>SINK=</td>
</tr>
<tr>
<td>Specifies the sink node for shortest path calculations</td>
<td>SHORTESTPATH</td>
<td>SOURCE=</td>
</tr>
</tbody>
</table>
Table 2.2  (continued)

<table>
<thead>
<tr>
<th>Description Statement</th>
<th>Statement</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUMMARY Statement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Calculates information about biconnected components</td>
<td>SUMMARY</td>
<td>BICONNECTEDCOMPONENTS</td>
</tr>
<tr>
<td>Calculates information about connected components</td>
<td>SUMMARY</td>
<td>CONNECTEDCOMPONENTS</td>
</tr>
<tr>
<td>Calculates the approximate diameter and chooses the weight type</td>
<td>SUMMARY</td>
<td>DIAMETERAPPROX=</td>
</tr>
<tr>
<td>Specifies the output data table for summary results</td>
<td>SUMMARY</td>
<td>OUT=</td>
</tr>
<tr>
<td>Calculates information about shortest paths and chooses the weight type</td>
<td>SUMMARY</td>
<td>SHORTESTPATH=</td>
</tr>
</tbody>
</table>

TRANSITIVECLOSURE Statement

Specifies the output data table for transitive closure results
TRANSITIVECLOSURE OUT=

Table 2.3 lists the supported DIRECTION= option values in the PROC NETWORK statement.

Table 2.3  Supported Input Formats by Statement

<table>
<thead>
<tr>
<th>Statement</th>
<th>DIRECTION=</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UNDIRECTED</td>
</tr>
<tr>
<td>BICONNECTEDCOMPONENTS</td>
<td>X</td>
</tr>
<tr>
<td>CENTRALITY</td>
<td></td>
</tr>
<tr>
<td>AUTH=, HUB=</td>
<td></td>
</tr>
<tr>
<td>BETWEEN=, CLOSE=,</td>
<td>X</td>
</tr>
<tr>
<td>CLUSTERINGCOEF,</td>
<td>X</td>
</tr>
<tr>
<td>DEGREE, EIGEN=,</td>
<td>X</td>
</tr>
<tr>
<td>INFLUENCE=, PAGERANK=</td>
<td></td>
</tr>
<tr>
<td>CLIQUE</td>
<td>X</td>
</tr>
<tr>
<td>COMMUNITY</td>
<td></td>
</tr>
<tr>
<td>ALGORITHM=</td>
<td></td>
</tr>
<tr>
<td>LOUVAIN, LABELPROP</td>
<td>X</td>
</tr>
<tr>
<td>PARALLELLABELPROP</td>
<td></td>
</tr>
<tr>
<td>CONNECTEDCOMPONENTS</td>
<td></td>
</tr>
<tr>
<td>ALGORITHM=</td>
<td></td>
</tr>
<tr>
<td>DFS</td>
<td>X</td>
</tr>
<tr>
<td>PARALLEL, UNIONFIND</td>
<td></td>
</tr>
<tr>
<td>CORE</td>
<td>X</td>
</tr>
<tr>
<td>CYCLE</td>
<td></td>
</tr>
<tr>
<td>NODESIMILARITY</td>
<td>X</td>
</tr>
<tr>
<td>PATH</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>
Table 2.3  (continued)

<table>
<thead>
<tr>
<th>Statement</th>
<th>DIRECTION=</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UNDIRECTED</td>
</tr>
<tr>
<td>PATTERNMATCH</td>
<td>X</td>
</tr>
<tr>
<td>REACH</td>
<td>X</td>
</tr>
<tr>
<td>SHORTESTPATH</td>
<td>X</td>
</tr>
<tr>
<td>SUMMARY</td>
<td></td>
</tr>
<tr>
<td>BICONNECTEDCOMPONENTS</td>
<td>X</td>
</tr>
<tr>
<td>DIAMETERAPPROX=</td>
<td></td>
</tr>
<tr>
<td>otherwise</td>
<td>X</td>
</tr>
<tr>
<td>TRANSITIVECLOSURE</td>
<td>X</td>
</tr>
</tbody>
</table>

For each algorithm statement in the NETWORK procedure, Table 2.4 indicates which output data table options you can specify and whether the algorithm populates the data tables that are specified in the OUTNODES= and OUTLINKS= options in the PROC NETWORK statement.

Table 2.4  Output Options by Statement

<table>
<thead>
<tr>
<th>Statement</th>
<th>OUTNODES=</th>
<th>OUTLINKS=</th>
<th>Algorithm Statement Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>BICONNECTEDCOMPONENTS</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CENTRALITY</td>
<td></td>
<td>X</td>
<td>OUT=</td>
</tr>
<tr>
<td></td>
<td>AUTH=, CLOSE=, CLUSTERINGCOEF, DEGREE, EIGEN=, HUB=, INFLUENCE=, PAGERANK= BETWEEN=</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>CLIQUE</td>
<td>X</td>
<td>X</td>
<td>OUTCOMMLINKS=, OUTCOMMUNITY=, OUTLEVEL=, OUTOVERLAP=</td>
</tr>
<tr>
<td>COMMUNITY</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CONNECTEDCOMPONENTS</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CORE</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CYCLE</td>
<td></td>
<td>OUT=</td>
<td></td>
</tr>
<tr>
<td>NODESIMILARITY</td>
<td>X</td>
<td></td>
<td>OUTSIMILARITY=</td>
</tr>
<tr>
<td>VECTOR=TRUE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PATH</td>
<td></td>
<td></td>
<td>OUTPATHSLINKS=, OUTPATHSNODES=</td>
</tr>
<tr>
<td>PATTERNMATCH</td>
<td></td>
<td></td>
<td>OUTMATCHNODES=, OUTMATCHLINKS=</td>
</tr>
<tr>
<td>REACH</td>
<td>X</td>
<td></td>
<td>OUTCOUNTS=, OUTREACHLINKS=, OUTREACHNODES=</td>
</tr>
<tr>
<td>SHORTESTPATH</td>
<td></td>
<td></td>
<td>OUTPATHS=, OUTWEIGHTS=</td>
</tr>
<tr>
<td>SUMMARY</td>
<td>X</td>
<td>X</td>
<td>OUT=</td>
</tr>
</tbody>
</table>
**Table 2.4** (continued)

<table>
<thead>
<tr>
<th>Statement</th>
<th>OUTNODES=</th>
<th>OUTLINKS=</th>
<th>Algorithm Statement Options</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRANSITIVECLOSURE</td>
<td></td>
<td></td>
<td>OUT=</td>
</tr>
</tbody>
</table>

**PROC NETWORK Statement**

```
PROC NETWORK < options > ;
```

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

**DIRECTION=DIRECTED | UNDIRECTED**

specifies whether the input graph should be considered directed or undirected. You can specify the following values:

- **DIRECTED** specifies the graph as directed. In a directed graph, each link \((i, j)\) has a direction that defines how something (such as information) can flow over that link. In link \((i, j)\), the flow is from node \(i\) to node \(j\) \((i \rightarrow j)\). Node \(i\) is called the source (tail) node, and node \(j\) is called the sink (head) node.

- **UNDIRECTED** specifies the graph as undirected. In an undirected graph, each link \(\{i, j\}\) has no direction and the flow can be in either direction. That is, \(\{i, j\} = \{j, i\}\).

By default, DIRECTION=UNDIRECTED. For more information, see the section “Graph Input Data” on page 49.

**DISTRIBUTED=TRUE | FALSE**

specifies whether to use a distributed graph. By default, DISTRIBUTED=FALSE, which means that a distributed graph is not used. For more information about the algorithms that support distributed graph computation, see the section “Execution Modes and Data Movement” on page 59.

**INCLUDESELFLINK**

includes self-links, such as \((i, i)\), when an input graph is read. By default, when PROC NETWORK reads the LINKS= data table, or the LINKSQUERY= data table, it removes all self-links.

**INDEXOFFSET=number**

specifies the index offset for identifiers in the log and results output data tables. For example, if three cycles are found in cycle enumeration, they are labeled cycles 1, 2, and 3 by default. If INDEXOFFSET=4, they are labeled cycles 4, 5, and 6. The value of `number` must be an integer greater than or equal to 0. By default, INDEXOFFSET=1.

**LINKS=CAS-libref.data-table**

specifies the input data table that contains the graph link information. `CAS-libref.data-table` is a two-level name, where `CAS-libref` refers to the caslib and session identifier, and `data-table` specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

For more information about this input table, see the section “Links Input Data” on page 50.
LINKSQUERY=\texttt{CAS-libref.data-table}

specifies the input data table that contains the graph link information for the query graph. \texttt{CAS-libref.data-table} is a two-level name, where \texttt{CAS-libref} refers to the caslib and session identifier, and \texttt{data-table} specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

This option is supported only when used in conjunction with the \texttt{PATTERNMATCH} statement. For more information about this input table, see the section “Pattern Matching” on page 114.

\textbf{LOGFREQTIME=}number

\textbf{LOGFREQUENCYTIME=}number

controls the frequency, in number of seconds, for displaying iteration logs for some algorithms. This option is useful for computationally intensive algorithms. Setting this value too low can hurt algorithm performance. The value of number can be any integer greater than or equal to 1. By default, LOGFREQTIME=5.

\textbf{LOGLEVEL=}NONE | BASIC | MODERATE | AGGRESSIVE

controls the amount of information that is displayed in the SAS log. You can specify the following values:

- **NONE** turns off all procedure-related messages in the SAS log.
- **BASIC** displays a brief summary of the algorithmic processing.
- **MODERATE** displays a moderately detailed summary of the input, output, and algorithmic processing.
- **AGGRESSIVE** displays a more detailed summary of the input, output, and algorithmic processing.

By default, LOGLEVEL=BASIC.

\textbf{NODES=}\texttt{CAS-libref.data-table}

specifies the input data table that contains the graph node information. \texttt{CAS-libref.data-table} is a two-level name, where \texttt{CAS-libref} refers to the caslib and session identifier, and \texttt{data-table} specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

For more information about this input table, see the section “Nodes Input Data” on page 53.

\textbf{NODESQUERY=}\texttt{CAS-libref.data-table}

specifies the input data table that contains the graph node information for the query graph. \texttt{CAS-libref.data-table} is a two-level name, where \texttt{CAS-libref} refers to the caslib and session identifier, and \texttt{data-table} specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

This option is supported only when used in conjunction with the \texttt{PATTERNMATCH} statement. For more information about this input table, see the section “Pattern Matching” on page 114.

\textbf{NODESSUBSET=}\texttt{CAS-libref.data-table}

specifies the input data table that contains the graph node subset information. \texttt{CAS-libref.data-table} is a two-level name, where \texttt{CAS-libref} refers to the caslib and session identifier, and \texttt{data-table} specifies the name of the input data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

For more information about this input table, see the section “Nodes Subset Input Data” on page 54.
NTHREADS=number

specifies the maximum number of threads to use for multithreaded processing. Some of the algorithms can take advantage of multicore machines and can run faster when number is greater than 1 (see Table 2.6 for a list). Algorithms that cannot take advantage of this option use only one thread even if number is greater than 1. For distributed execution, number specifies the maximum number of threads to use on each machine. The value of number can be any integer between 1 and 256, inclusive. The default is the number of cores on the machine that executes the process or the number of cores permissible based on your installation (whichever is less). The number of simultaneously active CPUs is limited by your installation and license configuration.

OUTLINKS=CAS-libref.data-table

specifies the output data table to contain the graph link information along with any results from the algorithms that calculate metrics on links. CAS-libref.data-table is a two-level name, where CAS-libref refers to the caslib and session identifier, and data-table specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

For examples of the content of this output data table, see the various algorithm sections.

OUTNODES=CAS-libref.data-table

specifies the output data table to contain the graph node information along with any results from the algorithms that calculate metrics on nodes. CAS-libref.data-table is a two-level name, where CAS-libref refers to the caslib and session identifier, and data-table specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

For examples of the content of this output data table, see the various algorithm sections.

STANDARDIZEDLABELS

specifies that the input graph data are in a standardized format, as described in the section “Standardized Labels” on page 57.

TIMETYPE=CPU | REAL

specifies whether CPU time or real time is used for each algorithm’s MAXTIME= option (where applicable). You can specify the following values:

CPU specifies units of CPU time. The time restriction is applied per processing machine (not across all machines).

REAL specifies units of real time.

By default, TIMETYPE=REAL.

BICONNECTEDCOMPONENTS Statement

BICONNECTEDCOMPONENTS ;

The BICONNECTEDCOMPONENTS statement requests that PROC NETWORK find biconnected components and articulation points of an undirected input graph. For more information, see the section “Biconnected Components and Articulation Points” on page 62.
BY Statement

BY variables;

You can specify a BY statement in PROC NETWORK to obtain separate analyses of observations in groups that are defined by the values of the BY variables. If you specify more than one BY statement, only the last one specified is used. For more information, see the discussion of BY-group processing in SAS Language Reference: Concepts.

The BY statement in PROC NETWORK is not supported when either a nodes or nodes subset data table is used. The BY variable must come from the LINKS= data table. Two examples of this are shown in “Example 2.8: Centrality Metrics for a Simple Undirected Graph by Community” on page 174 and “Example 2.14: Shortest Path in a Road Network by Date and Time” on page 193.

All parameter settings apply to each individual group independently (not to the entire process as a whole). For example, when a stopping criterion such as the MAXTIME= option is specified for a particular algorithm, this limit pertains to each individual group as it is processed.

CENTRALITY Statement

CENTRALITY < options >;

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

For more information about centrality metrics, see the section “Centrality” on page 66.

You can specify the following options:

AUTH=WEIGHT | UNWEIGHT | BOTH
specifies which type of authority centrality to calculate. You can specify the following values:

WEIGHT calculates authority centrality by using the weighted graph.
UNWEIGHT calculates authority centrality by using the unweighted graph.
BOTH calculates authority centrality by using both the weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both produce the same results (if you use 1.0 for each link weight). This centrality metric can be used only for directed graphs. For more information about the authority centrality metric, see the section “Hub and Authority Scoring” on page 77.

BETWEEN=WEIGHT | UNWEIGHT | BOTH
specifies which type of betweenness centrality to calculate for node betweenness or link betweenness. You can specify the following values:
WEIGHT calculates betweenness centrality by using the weighted graph.
UNWEIGHT calculates betweenness centrality by using the unweighted graph.
BOTH calculates betweenness centrality by using both the weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both produce the same results (if you use 1.0 for each link weight). If the OUTNODES= option is specified in the PROC NETWORK statement, the node betweenness metric is produced. If the OUTLINKS= option is specified, the link betweenness metric is produced. For more information about the betweenness centrality metric, see the section “Betweenness Centrality” on page 73.

BETWEENNORM=TRUE | FALSE
specifies whether to normalize the betweenness centrality metrics. You can specify the following values:

TRUE normalizes the betweenness metrics.
FALSE does not normalize the betweenness metrics.

For more information about the normalization factor for betweenness centrality, see the section “Betweenness Centrality” on page 73. By default, BETWEENNORM=TRUE.

CLOSE=WEIGHT | UNWEIGHT | BOTH
specifies which type of closeness centrality to calculate. You can specify the following values:

WEIGHT calculates closeness centrality by using the weighted graph.
UNWEIGHT calculates closeness centrality by using the unweighted graph.
BOTH calculates closeness centrality by using both the weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both produce the same results (if you use 1.0 for each link weight). For more information about the closeness centrality metric, see the section “Closeness Centrality” on page 70.

CLOSENOPATH=DIAMETER | HARMONIC | NNODES | ZERO
specifies a method for accounting for the shortest path distance between two nodes when a path does not exist (disconnected nodes). You can specify the following values:

DIAMETER uses the graph diameter (plus one) as the shortest path distance between disconnected nodes.
HARMONIC uses the harmonic formula for calculating closeness centrality.
NNODES uses the number of nodes as the shortest path distance between disconnected nodes.
ZERO uses zero as the shortest path distance between disconnected nodes.

You cannot specify this option when CLOSE=WEIGHT or CLOSE=BOTH.

For each option, there is a slight variation in the formula for the closeness centrality metric. For more information about these differences, see the section “Closeness Centrality” on page 70. By default, CLOSENOPATH=DIAMETER.
CLUSTERINGCOEF
  calculates the node clustering coefficient. For more information about the clustering coefficient, see the section “Clustering Coefficient” on page 68.

DEGREE
  calculates the degree centrality. For more information about the degree centrality metric, see the section “Degree Centrality” on page 66.

EIGEN=WEIGHT | UNWEIGHT | BOTH
  specifies which type of eigenvector centrality to calculate. You can specify the following values:

  WEIGHT  calculates eigenvector centrality by using the weighted graph.

  UNWEIGHT  calculates eigenvector centrality by using the unweighted graph.

  BOTH  calculates eigenvector centrality by using both the weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both produce the same results (if you use 1.0 for each link weight). For more information about the eigenvector centrality metric, see the section “Eigenvector Centrality” on page 75.

EIGENALGORITHM=AUTOMATIC | JACOBIDAVIDSON | POWER
  specifies the algorithm to use in calculating centrality metrics that require solving eigensystems—that is, when the EIGEN, PAGERANK, HUB, or AUTH option (or some combination) is specified. You can specify the following values:

  AUTOMATIC  automatically determines the eigensolver to use.

  JACOBIDAVIDSON | JD  uses a variant of the Jacobi-Davidson algorithm for solving eigensystems (Sleijpen and van der Vorst 2000). This algorithm is not supported for PageRank calculations.

  POWER  uses the power method to calculate eigenvectors.

By default, EIGENALGORITHM=AUTOMATIC.

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

HUB=WEIGHT | UNWEIGHT | BOTH
  specifies which type of hub centrality to calculate. You can specify the following values:

  WEIGHT  calculates hub centrality by using the weighted graph.

  UNWEIGHT  calculates hub centrality by using the unweighted graph.

  BOTH  calculates hub centrality by using both the weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both produce the same results (if you use 1.0 for each link weight). This centrality metric can be used only for directed graphs. For more information about the hub centrality metric, see the section “Hub and Authority Scoring” on page 77.
INFLUENCE=WEIGHT | UNWEIGHT | BOTH
specifies which type of influence centrality to calculate. You can specify the following values:

WEIGHT calculates influence centrality by using the weighted graph.
UNWEIGHT calculates influence centrality by using the unweighted graph.
BOTH calculates influence centrality by using both the weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both produce the same results (if you use 1.0 for each link or node weight). For more information about the influence centrality metric, see the section “Influence Centrality” on page 67.

INTERNALFORMAT=FULL | THIN
specifies the internal graph format for the centrality calculations to use. You can specify the following values:

FULL stores the graph in standard (adjacency-list-based) format.
THIN stores the graph in thin (simple list of links) format. This option can improve performance in some cases both by reducing memory and by simplifying the construction of the internal data structures. This option causes PROC NETWORK to skip the removal of duplicate links when it reads in the graph.

By default, INTERNALFORMAT=FULL. The only centrality metrics that you can use with the INTERNALFORMAT=THIN option are degree, influence and clustering coefficient. For more information, see the section “Graph Input Data” on page 49.

PAGERANK=WEIGHT | UNWEIGHT | BOTH
specifies which type of PageRank centrality to calculate. You can specify the following values:

WEIGHT calculates PageRank centrality by using the weighted graph.
UNWEIGHT calculates PageRank centrality by using the unweighted graph.
BOTH calculates PageRank centrality by using both the weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both produce the same results (if you use 1.0 for each link or node weight). For more information about the PageRank centrality metric, see the section “PageRank Centrality” on page 79.

PAGERANKALPHA=number
specifies the damping factor to use in the PageRank algorithm. The value of number must be between 0 and 1 (inclusive). The default is 0.85, which means that a random web surfer has a 15% chance of jumping to any other node in the network at any time. The algorithm usually takes more iterations to converge (or does not converge at all) as the damping factor is set closer to 1. See the section “PageRank Centrality” on page 79 for an example that uses this option.

PAGERANKTOL=number
specifies the convergence tolerance value for the PageRank algorithm. The value of number must be a positive number; the default value is 1E−9. The algorithm stops power iterations when the gap between the PageRank scores of the current iteration and the previous iteration is less than or equal to number.
CLIQUES Statement

CLIQUES < options > ;

The CLIQUES statement invokes an algorithm that finds maximal cliques in the input graph. For more information about maximal cliques, see the section “Clique Enumeration” on page 81.

You can specify the following options:

MAXCLIQUE=number | ALL
specifies the maximum number of cliques to return during clique enumeration. You can specify either a number (which can be any 32-bit integer greater than or equal to 1) or ALL (which represents the maximum that can be represented by a 32-bit integer). By default, MAXCLIQUE=1.

MAXLINKWEIGHT=number
specifies the maximum sum of link weights in a clique. Any clique whose sum of link weights is greater than number is removed from the results. The default is the largest number that can be represented by a double, which causes no cliques to be removed from the results.

MAXNODEWEIGHT=number
specifies the maximum sum of node weights in a clique. Any clique whose sum of node weights is greater than number is removed from the results. The default is the largest number that can be represented by a double, which causes no cliques to be removed from the results.

MAXSIZE=number
specifies the maximum number of nodes in a clique. Any clique whose size is greater than number is removed from the results. The default is the largest number that can be represented by a 32-bit integer, which causes no cliques to be removed from the results.

MAXTIME=number
specifies the maximum amount of time to spend finding cliques. The type of time (either CPU time or real time) is determined by the value of the TIMETYPE= option in the PROC NETWORK statement. The default is the largest number that can be represented by a double.

MINLINKWEIGHT=number
specifies the minimum sum of link weights in a clique. Any clique whose sum of link weights is less than number is removed from the results. The default is the largest (in magnitude) negative number that can be represented by a double, which causes no cliques to be removed from the results.

MINNODEWEIGHT=number
specifies the minimum sum of node weights in a clique. Any clique whose sum of node weights is less than number is removed from the results. The default is the largest (in magnitude) negative number that can be represented by a double, which causes no cliques to be removed from the results.

MINSIZE=number
specifies the minimum number of nodes in a clique. Any clique that has fewer nodes than number is removed from the results. By default, MINSIZE=1 and no cliques are removed from the results.
OUT=CAS-libref.data-table
specifies the output data table to contain the maximal cliques. CAS-libref.data-table is a two-level name, where CAS-libref refers to the caslib and session identifier, and data-table specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

COMMUNITY Statement

COMMUNITY < options > ;
The COMMUNITY statement invokes an algorithm that detects communities in the input graph. For more information about community detection, see the section “Community Detection” on page 84.

You can specify the following options:

ALGORITHM=LOUVAIN | LABELPROP | PARALLELLABELPROP
specifies the algorithm to use for community detection. You can specify the following values:

LOUVAIN uses the Louvain algorithm proposed in Blondel et al. (2008).
LABELPROP uses the label propagation algorithm proposed in Raghavan, Albert, and Kumara (2007).
PARALLELLABELPROP uses the parallel (distributed and threaded) label propagation algorithm developed by SAS. This can also be enabled by setting DISTRIBUTED=TRUE on the PROC NETWORK statement.

By default, ALGORITHM=LOUVAIN.

INTERNALFORMAT=FULL | THIN
specifies the internal graph format for the community detection algorithm to use. You can specify the following values:

FULL stores the graph in standard (adjacency-list-based) format.
THIN stores the graph in thin (simple list of links) format. This option can improve performance in some cases both by reducing memory and by simplifying the construction of the internal data structures. This option causes PROC NETWORK to skip the removal of duplicate links when it reads in the graph.

By default, INTERNALFORMAT=THIN. You cannot use the option INTERNALFORMAT=FULL with ALGORITHM=PARALLELLABELPROP. If you do, it will be reset to the default value of THIN. For more information, see the section “Graph Input Data” on page 49.

LINKREMOVALRATIO=number
specifies 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 the 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 you
specify this option, then the links that are incident to each node are examined. If the weight of any link is less than \((\text{number}/100) \times \text{max\_link\_weight}\), where \(\text{max\_link\_weight}\) is the maximum link weight among all links incident to this node, the link is removed provided that its removal does not disconnect any node from the network. This option can often dramatically improve the running time for large graphs. The valid range is between 0 and 100. By default, \text{LINKREMOVALRATIO}=10.

\textbf{MAXITERS=number}

specifies the maximum number of iterations that the algorithm can run. By default, \text{MAXITERS}=20 when \text{ALGORITHM}=LOUVAIN or \text{MAXITERS}=100 when \text{ALGORITHM}=LABELPROP or \text{ALGORITHM}=PARALLELLABELPROP.

\textbf{OUTCOMMLINKS=}\text{CAS-libref.data-table}

specifies the output data table that describes the links between communities. \text{CAS-libref.data-table} is a two-level name, where \text{CAS-libref} refers to the \text{caslib} and session identifier, and \text{data-table} specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

\textbf{OUTCOMMUNITY=}\text{CAS-libref.data-table}

specifies the output data table that contains the number of nodes in each community. \text{CAS-libref.data-table} is a two-level name, where \text{CAS-libref} refers to the \text{caslib} and session identifier, and \text{data-table} specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

\textbf{OUTLEVEL=}\text{CAS-libref.data-table}

specifies the output data table that contains community information at different resolution levels. \text{CAS-libref.data-table} is a two-level name, where \text{CAS-libref} refers to the \text{caslib} and session identifier, and \text{data-table} specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

\textbf{OUTOVERLAP=}\text{CAS-libref.data-table}

specifies the output data table that describes the intensity of each node. \text{CAS-libref.data-table} is a two-level name, where \text{CAS-libref} refers to the \text{caslib} and session identifier, and \text{data-table} specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

\textbf{RANDOMFACTOR=}\text{number}

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

\textbf{RANDOMSEED=}\text{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 \text{RANDOMFACTOR=} option. To choose a different set of random samples, specify a \text{number} in the \text{RANDOMSEED=} option. By default, \text{RANDOMSEED}=1234.

\textbf{RECURSIVE (options)}

breaks down large communities into smaller ones until the specified conditions are satisfied. This option starts with the keyword \text{RECURSIVE} followed by any combination of three suboptions enclosed
in parentheses—for example, RECURSIVE (MAXCOMMSIZE=500) or RECURSIVE (MAXCOMMSIZE=1000 MAXDIAMETER=3 RELATION=AND).

You can specify the following options:

**MAXCOMMSIZE=** specifies the maximum number of nodes to be contained in any community. The default is the largest number that can be represented by a 32-bit integer.

**MAXDIAMETER=** specifies the maximum number of links on the shortest paths between any pair of nodes in any community. The MAXDIAMETER= option is ignored when you specify ALGORITHM=PARALLELLABELPROP. The default is the largest number that can be represented by a 32-bit integer.

**RELATION=** the relationship between the values of MAXCOMMSIZE= and MAXDIAMETER= options. If RELATION=AND, then recursive splitting continues until both of the MAXCOMMSIZE and MAXDIAMETER conditions are satisfied. If RELATION=OR, then recursive splitting continues until either the MAXCOMMSIZE or the MAXDIAMETER condition is satisfied.

**RESOLUTIONLIST=num_list** specifies a list of resolution values (positive numbers) that are separated by spaces (for example, 4.3 2.1 1.0 0.6 0.2). The NETWORK procedure interprets the RESOLUTIONLIST= 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, RESOLUTIONLIST=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=LABELPROP, PROC NETWORK ignores the RESOLUTIONLIST= option. It uses the default value of 1.0.

- When ALGORITHM=PARALLELLABELPROP, specifying multiple resolution values requests that PROC NETWORK perform community detection multiple times, each time with a different resolution value. By default, RESOLUTIONLIST=0.001. In this case, the RESOLUTIONLIST= option is fully compatible with the RECURSIVE option.

For more information about the use of the RESOLUTIONLIST= option, see the section “Large Communities” on page 86.

**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 of modularity gain between two consecutive iterations is less than number. When you specify ALGORITHM=LABELPROP or ALGORITHM=PARALLELLABELPROP, the algorithm stops iterations when the percentage of label changes for all nodes in the graph is less than number. The valid range is strictly between 0 and 1. By default, TOLERANCE=0.001 when you specify ALGORITHM=LOUVAIN or ALGORITHM=LABELPROP, and TOLERANCE=0.01 when ALGORITHM=PARALLELLABELPROP is specified.
CONNECTEDCOMPONENTS Statement

CONNECTEDCOMPONENTS < options > ;

The CONNECTEDCOMPONENTS statement invokes an algorithm that finds the connected components of the input graph. For more information about connected components, see the section “Connected Components” on page 92. You can specify the following options:

**ALGORITHM=** AUTOMATIC | DFS | PARALLEL | UNIONFIND

specifies the algorithm to use for calculating connected components. You can specify the following values:

- **AUTOMATIC** automatically determines the algorithm for connected components.
- **DFS** uses the depth-first search algorithm for connected components.
- **PARALLEL** uses the distributed parallel union-find algorithm for connected components. You can specify this value when the number of machines in your session is greater than 1. You can use this algorithm only with undirected graphs. This can also be enabled by setting DISTRIBUTED=TRUE on the PROC NETWORK statement.
- **UNIONFIND** uses the union-find algorithm for connected components. You can use this algorithm only with undirected graphs.

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

**INTERNALFORMAT=** FULL | THIN

specifies the internal graph format for the connected components algorithm to use. You can specify the following values:

- **FULL** stores the graph in standard (adjacency-list-based) format.
- **THIN** stores the graph in thin (simple list of links) format. This option can improve performance in some cases both by reducing memory and by simplifying the construction of the internal data structures. This option causes PROC NETWORK to skip the removal of duplicate links when it reads in the graph (which has no effect on the resulting components).

By default, INTERNALFORMAT=THIN. You cannot use the option INTERNALFORMAT=FULL with ALGORITHM=PARALLEL. If you do, it will reset to the default value of THIN. For more information, see the section “Graph Input Data” on page 49.

CORE Statement

CORE < option > ;

The CORE statement invokes an algorithm that finds the core decomposition of the input graph. For more information about core decomposition, see the section “Core Decomposition” on page 97.

You can specify the following option:
**MAXTIME=number**
specifies the maximum amount of time to spend calculating the core decomposition. The type of time (either CPU time or real time) is determined by the value of the **TIMETYPE=** option in the PROC NETWORK statement. The default is the largest number that can be represented by a double.

---

**CYCLE Statement**

```
CYCLE < options > ;
```

The CYCLE statement invokes an algorithm that finds the cycles (or the existence of a cycle) in the input graph. For more information about cycles, see the section “Cycle Enumeration” on page 101.

You can specify the following **options**:

**ALGORITHM=BACKTRACK | BUILD**
specifies which algorithm to use in enumerating cycles. You can specify the following values:

- **BACKTRACK** uses a backtracking algorithm based on Johnson (1975).

By default, **ALGORITHM=BACKTRACK** for **MAXLENGTH** greater than 20; otherwise, **ALGORITHM=BUILD**.

**MAXCYCLES=number | ALL**
specifies the maximum number of cycles to return during cycle enumeration. You can specify either a **number** (which can be any 32-bit integer greater than or equal to 1) or **ALL** (which represents the maximum that can be represented by a 32-bit integer). By default, **MAXCYCLES=1**.

**MAXLENGTH=number**
specifies the maximum number of links in a cycle. Any cycle whose length is greater than **number** is removed from the results. The default is the largest number that can be represented by a 32-bit integer, which causes no cycles to be removed from the results.

**MAXLINKWEIGHT=number**
specifies the maximum sum of link weights in a cycle. Any cycle whose sum of link weights is greater than **number** is removed from the results. The default is the largest number that can be represented by a double, which causes no cycles to be removed from the results.

**MAXNODEWEIGHT=number**
specifies the maximum sum of node weights in a cycle. Any cycle whose sum of node weights is greater than **number** is removed from the results. The default is the largest number that can be represented by a double, which causes no cycles to be removed from the results.

**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 in the PROC NETWORK statement. The default is the largest number that can be represented by a double.
MINLENGTH=number
specifies the minimum number of links in a cycle. Any cycle that has fewer links than number is removed from the results. By default, MINLENGTH=1 and no cycles are removed from the results.

MINLINKWEIGHT=number
specifies the minimum sum of link weights in a cycle. Any cycle whose sum of link weights is less than number is removed from the results. The default is the largest (in magnitude) negative number that can be represented by a double, which causes no cycles to be removed from the results.

MINNODEWEIGHT=number
specifies the minimum sum of node weights in a cycle. Any cycle whose sum of node weights is less than number is removed from the results. The default is the largest (in magnitude) negative number that can be represented by a double, which causes no cycles to be removed from the results.

OUT=CAS-libref.data-table
specifies the output data table to contain the cycles found. CAS-libref.data-table is a two-level name, where CAS-libref refers to the caslib and session identifier, and data-table specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

DISPLAY Statement

DISPLAY < table-list > < / options > ;

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The procedure processes the DISPLAY statement on a CAS server and thus sends only a subset of ODS tables to the SAS client. Because ODS statements are processed on a SAS client, first all the generated display tables are sent to the client, and then the client creates a subset.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables that are sent to the client after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see SAS Output Delivery System: Procedures Guide.

You can specify the table-list as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section “ODS Table Names” on page 151. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path Bygroup1.Summary.SelectionSummary. A partial pathname does not include all groups; for example, Selection- and Summary.SelectionSummary are partial pathnames for Bygroup1.Summary.SelectionSummary.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both SelectionSummary and Summary.SelectionSummary select Bygroup1.Summary.SelectionSummary.

A regular expression is enclosed in forward slashes (/). For example, specifying “/tions/” selects all pathnames that contain the substring “tions”; in particular, the Bygroup1.Summary.SelectionSummary table is selected.
Specifying “!/tions/” selects all pathnames that do not contain the substring “tions”; in particular, the `Bygroup1.Summary.SelectionSummary` table is not selected.

You can specify the following options after a slash (/):

**CASESENSITIVE**
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

**EXCLUDE**
displays all display tables except those that you specify in the *table-list*.

**EXCLUDEALL**
suppresses display of all tables. This option takes precedence over the other options.

**TRACE**
displays the display table names, labels, and paths.

---

**DISPLAYOUT Statement**

```
DISPLAYOUT table-spec-list </ options> ;
```

The DISPLAYOUT statement enables you to create CAS output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of CAS output tables to create. Each entry in the list has either a *key=value* format or a *key* format:

- **key=value** specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the CAS output table name.
- **key** specifies *key* as the ODS table name and also as the CAS output table name.

The ODS table names that you can specify are listed in the section “ODS Table Names” on page 151. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following options after a slash (/):

**INCLUDEALL**
creates output CAS tables for all display tables. The name of the created output CAS table is the same as the corresponding display table name. If you specify this option, the *table-spec-list* specification is ignored.

**NOREPLACE**
does not replace any existing CAS output table of the same name.
Chapter 2: The NETWORK Procedure

REPEATED

replicates all CAS output tables on all nodes.

The output tables that the NETWORK procedure produces when you use the DISPLAYOUT statement are a transposed version of the displayed tables. This allows for easier use in subsequent analyses, especially when it is used together with BY-group processing. Two examples of using the DISPLAYOUT statement are shown in “Example 2.8: Centrality Metrics for a Simple Undirected Graph by Community” on page 174 and “Example 2.14: Shortest Path in a Road Network by Date and Time” on page 193.

LINKSVAR Statement

LINKSVAR < options > ;

The LINKSVAR statement enables you to explicitly specify the data variable names for PROC NETWORK to use when it reads the data table that you specify in the LINKS= option in the PROC NETWORK statement. For more information about the format of the links input data table, see the section “Links Input Data” on page 50.

You can specify the following options:

AUXWEIGHT=column

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

FROM=column
FROMVAR= option

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

TO=column
TOVAR= option

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

VARS=(column(s))

specifies the name(s) of the additional data variable(s) to carry over to the output results. The value of the column(s) variable(s) can be numeric or character.

WEIGHT=column

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

LINKSQUERYVAR Statement

LINKSQUERYVAR < options > ;

The LINKSQUERYVAR statement enables you to explicitly specify the data variable names for PROC NETWORK to use when it reads the data table that you specify in the LINKSQUERY= option in the PROC
NODESIMILARITY Statement

NODESIMILARITY < options > ;

The NODESIMILARITY statement invokes an algorithm that calculates how similar the neighborhoods of pairs of nodes are in an input graph. The node similarity problem is described in the section “Node Similarity” on page 105.

You can specify the following options:

**FROM=column**

**FROMVAR=column**

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

**TO=column**

**TOVAR=column**

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

**VARS=(column(s))**

specifies the name(s) of the data variable(s) to consider in the query graph. The value of the *column(s)* variable(s) can be numeric or character.

---

**INTERNALFORMAT=FULL | THIN**

specifies the internal graph format for the node similarity algorithm to use. You can specify the following values:

- **FULL**
  stores the graph in standard (adjacency-list-based) format.
- **THIN**
  stores the graph in thin (simple list of links) format. This option can improve performance in some cases both by reducing memory and by simplifying the construction of the internal data structures. This option causes PROC NETWORK to skip the removal of duplicate links when it reads in the graph.

By default, INTERNALFORMAT=FULL. For more information, see the section “Graph Input Data” on page 49.

**JACCARD=TRUE | FALSE**

specifies whether to calculate Jaccard node similarity. You can specify the following values:

- **TRUE**
  calculates Jaccard node similarity and saves the results in the output OUTSIMILARITY= table.
- **FALSE**
  does not calculate Jaccard node similarity.

By default, JACCARD=TRUE.
**MAXSCORE=**`number`

specifies the maximum similarity value to output in the OUTSIMILARITY= table in the range [0, 1]. By default, MAXSCORE=1.

**MINSCORE=**`number`

specifies the minimum similarity value to output in the OUTSIMILARITY= table in the range [0, 1]. By default, MINSCORE=0.

**NDIMENSIONS=**`number`

specifies the number of dimensions for node-embedding vectors. By default, NDIMENSIONS=100.

**NEGATIVESAMPLEFACTOR=**`number`

specifies a multiplier for the number of negative training samples for the vector algorithm. By default, NEGATIVESAMPLEFACTOR=5 × s, where s is the number of samples specified in the NSAMPLES= option.

**NSAMPLES=**`number`

specifies the number of training samples for the vector algorithm. By default, NSAMPLES=1000 × |E|, where |E| is the number of links in the input graph.

**OUTSIMILARITY=**`CAS-libref.data-table`

specifies the output data table to contain the similarity values for pairs of nodes. `CAS-libref.data-table` is a two-level name, where `CAS-libref` refers to the caslib and session identifier, and `data-table` specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

**PROXIMITYORDER=**`FIRST | SECOND`

specifies the type of proximity to use in the vector algorithm. You can specify the following values:

- **FIRST** uses first-order proximity in the vector algorithm.
- **SECOND** uses second-order proximity in the vector algorithm.

By default, PROXIMITYORDER=SECOND.

**SINK=**`sink-node`

specifies the sink node for node similarity calculations.

**SOURCE=**`source-node`

specifies the source node for node similarity calculations.

**VECTOR=**`TRUE | FALSE`

specifies whether to calculate vector node similarity. You can specify the following values:

- **TRUE** calculates vector node similarity and saves the results in the output OUTSIMILARITY= table.
- **FALSE** does not calculate vector node similarity.

By default, VECTOR=FALSE.
NODESSUBSETVAR Statement

NODESSUBSETVAR < options > ;

The NODESSUBSETVAR statement enables you to explicitly specify the data variable names for PROC NETWORK to use when it reads the data table that you specify in the NODESSUBSET= option in the PROC NETWORK statement. For more information about the format of the node subset input data table, see the section “Nodes Input Data” on page 53.

You can specify the following options:

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

REACH=column
specifies the name of the data variable for the reach identifier. The value of the column variable must be numeric.

SINK=column
specifies the name of the data variable for the sink indicator. The value of the column variable must be numeric.

SOURCE=column
specifies the name of the data variable for the source indicator. The value of the column variable must be numeric.

NODESVAR Statement

NODESVAR < options > ;

The NODESVAR statement enables you to explicitly specify the data variable names for PROC NETWORK to use when it reads the data table that you specify in the NODES= option in the PROC NETWORK statement. For more information about the format of the node input data table, see the section “Nodes Input Data” on page 53.

You can specify the following options:

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

VARS=(column(s))
specifies the name(s) of the additional data variable(s) to carry over to the output results. The value of the column(s) variable(s) can be numeric or character.

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

```
NODEQUERYVAR < options > ;
```

The NODEQUERYVAR statement enables you to explicitly specify the data variable names for PROC NETWORK to use when it reads the data table that you specify in the NODEQUERY= option in the PROC NETWORK statement. This option is supported only when used in conjunction with the PATTERNMATCH statement. For more information about this input table, see the section “Pattern Matching” on page 114.

You can specify the following `options`:

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

- **VARS=(column(s))**
  specifies the name(s) of the data variable(s) to consider in the query graph. The value of the `column(s)` variable(s) can be numeric or character.

**PATH Statement**

```
PATH < options > ;
```

The PATH statement invokes an algorithm that finds the paths in the input graph. For more information about paths, see the section “Path Enumeration” on page 110.

You can specify the following `options`:

- **INTERNALFORMAT=FULL | THIN**
  specifies the internal graph format for the path enumeration algorithm to use. You can specify the following values:
    - **FULL** stores the graph in standard (adjacency-list-based) format.
    - **THIN** stores the graph in thin (simple list of links) format. This option can improve performance in some cases both by reducing memory and by simplifying the construction of the internal data structures. This option causes PROC NETWORK to skip the removal of duplicate links when it reads in the graph.

By default, INTERNALFORMAT=THIN. For more information, see the section “Graph Input Data” on page 49.

- **MAXLENGTH=number**
  specifies the maximum number of links in a path. Any path whose length is greater than `number` is removed from the results. The default is the largest number that can be represented by a 32-bit integer, which causes no paths to be removed from the results.
**MAXLINKWEIGHT=**number
specifies the maximum sum of link weights in a path. Any path whose sum of link weights is greater than number is removed from the results. The default is the largest number that can be represented by a double, which causes no paths to be removed from the results.

**MAXNODEWEIGHT=**number
specifies the maximum sum of node weights in a path. Any path whose sum of node weights is greater than number is removed from the results. The default is the largest number that can be represented by a double, which causes no paths to be removed from the results.

**MAXTIME=**number
specifies the maximum amount of time to spend finding paths. The type of time (either CPU time or real time) is determined by the value of the TIMETYPE= option in the PROC NETWORK statement. The default is the largest number that can be represented by a double.

**MINLENGTH=**number
specifies the minimum number of links in a path. Any path that has fewer links than number is removed from the results. By default, MINLENGTH=1 and no paths are removed from the results.

**MINLINKWEIGHT=**number
specifies the minimum sum of link weights in a path. Any path whose sum of link weights is less than number is removed from the results. The default is the largest (in magnitude) negative number that can be represented by a double, which causes no paths to be removed from the results.

**MINNODEWEIGHT=**number
specifies the minimum sum of node weights in a path. Any path whose sum of node weights is less than number is removed from the results. The default is the largest (in magnitude) negative number that can be represented by a double, which causes no paths to be removed from the results.

**OUTPATHSLINKS=**CAS-libref.data-table
**OUTPATHS=**CAS-libref.data-table
specifies the output data table to contain the path links. CAS-libref.data-table is a two-level name, where CAS-libref refers to the caslib and session identifier, and data-table specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

**OUTPATHSNODES=**CAS-libref.data-table
specifies the output data table to contain the path nodes. CAS-libref.data-table is a two-level name, where CAS-libref refers to the caslib and session identifier, and data-table specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

**SINK=**sink-node
specifies the sink node for path calculations. This setting overrides the use of the variable sink in the data table that you specify in the NODESSUBSET= option in the PROC NETWORK statement.

**SOURCE=**source-node
specifies the source node for path calculations. This setting overrides the use of the variable source in the data table that you specify in the NODESSUBSET= option in the PROC NETWORK statement.
**PATTERNMATCH Statement**

```
PATTERNMATCH < options > ;
```

The PATTERNMATCH statement invokes an algorithm that searches for subgraph patterns in an input graph. The pattern matching problem is described in the section “Pattern Matching” on page 114.

You can specify the following `options`:

- **OUTMATCHLINKS=** `CAS-libref.data-table`
  specifies the links output data table for matching subgraphs. `CAS-libref.data-table` is a two-level name, where `CAS-libref` refers to the caslib and session identifier, and `data-table` specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

- **OUTMATCHNODES=** `CAS-libref.data-table`
  specifies the nodes output data table for matching subgraph mappings. `CAS-libref.data-table` is a two-level name, where `CAS-libref` refers to the caslib and session identifier, and `data-table` specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

**REACH Statement**

```
REACH < options > ;
```

The REACH statement invokes an algorithm that calculates the reach (ego) network in an input graph. For more information about the reach network, see the section “Reach (Ego) Network” on page 122.

You can specify the following `options`:

- **DIGRAPH**
  calculates the directed reach counts when computing the reach networks and includes the directed counts in the resulting output data table, which is specified in the `OUTCOUNTS=` option. This option is ignored unless `MAXREACH=1`.

- **EACHSOURCE**
  treats each node as a source and calculates a reach network from each one.

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

- **OUTCOUNTS=** `CAS-libref.data-table`
  specifies the output data table to contain the node counts in each reach network. `CAS-libref.data-table` is a two-level name, where `CAS-libref` refers to the caslib and session identifier, and `data-table` specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”
OUTREACHLINKS=\texttt{CAS-libref.data-table}
specifies the output data table to contain the links in each reach network. \texttt{CAS-libref.data-table} is a two-level name, where \texttt{CAS-libref} refers to the caslib and session identifier, and \texttt{data-table} specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

OUTREACHNODES=\texttt{CAS-libref.data-table}
specifies the output data table to contain the nodes in each reach network. \texttt{CAS-libref.data-table} is a two-level name, where \texttt{CAS-libref} refers to the caslib and session identifier, and \texttt{data-table} specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

\section*{SHORTESTPATH Statement}

\texttt{SHORTESTPATH < options > ;}

The \texttt{SHORTESTPATH} statement invokes an algorithm that calculates shortest paths between pairs of nodes in the input graph. By default, PROC NETWORK finds a shortest path for each possible combination of source and sink nodes. For more information about the shortest path algorithm, see the section “Shortest Path” on page 129.

You can specify the following \texttt{options}:

\texttt{MAXPATHWEIGHT=number}
specifies the maximum path weight. Any shortest path whose sum of link weights is greater than \texttt{number} is removed from the results. The default is the largest number that can be represented by a double, which causes no paths to be removed from the results.

\texttt{OUTPATHS=\texttt{CAS-libref.data-table}}
\texttt{OUT=\texttt{CAS-libref.data-table}}
specifies the output data table to contain the shortest paths. \texttt{CAS-libref.data-table} is a two-level name, where \texttt{CAS-libref} refers to the caslib and session identifier, and \texttt{data-table} specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

\texttt{OUTWEIGHTS=\texttt{CAS-libref.data-table}}
specifies the output data table to contain the shortest path summaries. \texttt{CAS-libref.data-table} is a two-level name, where \texttt{CAS-libref} refers to the caslib and session identifier, and \texttt{data-table} specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

\texttt{SINK=sink-node}
specifies the sink node for shortest path calculations. This setting overrides the use of the variable \texttt{sink} in the data table that you specify in the NODESSUBSET= option in the PROC NETWORK statement.

\texttt{SOURCE=source-node}
specifies the source node for shortest path calculations. This setting overrides the use of the variable \texttt{source} in the data table that you specify in the NODESSUBSET= option in the PROC NETWORK statement.
SUMMARY Statement

`SUMMARY < options > ;`

The SUMMARY statement invokes an algorithm that calculates various summary metrics for an input graph. For more information about summary metrics, see the section “Summary Statistics” on page 140.

You can specify the following `options`:

**BICONNECTEDCOMPONENTS**
- Calculates information about biconnected components. You can use this option only for an undirected graph.

**CONNECTEDCOMPONENTS**
- Calculates information about connected components.

**DIAMETERAPPROX=WEIGHT | UNWEIGHT | BOTH**
- Calculates information about the approximate diameter and specifies which type of calculation to perform. Use this option when calculating the exact diameter (by calculating all shortest paths) is too computationally expensive. You can specify the following values:
  - `WEIGHT` calculates the approximate diameter by using the weighted graph.
  - `UNWEIGHT` calculates the approximate diameter by using the unweighted graph.
  - `BOTH` calculates the approximate diameter by using both the weighted and unweighted graphs.

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

**OUT=** `CAS-libref.data-table`
- Specifies the output data table to contain the summary results. `CAS-libref.data-table` is a two-level name, where `CAS-libref` refers to the caslib and session identifier, and `data-table` specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

**SHORTESTPATH=WEIGHT | UNWEIGHT | BOTH**
- Calculates information about shortest paths and specifies which type of calculation to perform. You can specify the following values:
  - `WEIGHT` calculates shortest paths by using the weighted graph.
  - `UNWEIGHT` calculates shortest paths by using the unweighted graph.
  - `BOTH` calculates shortest paths by using both the weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both produce the same results (if you use 1.0 for each link weight).
**TRANSITIVECLOSURE Statement**

TRANSITIVECLOSURE < option >;

The TRANSITIVECLOSURE statement invokes an algorithm that calculates the transitive closure of an input graph. For more information about transitive closure, see the section “Transitive Closure” on page 147.

You can specify the following option:

**OUT=**CAS-libref.data-table

specifies the output data table to contain the transitive closure results. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

---

**Details: NETWORK Procedure**

**Graph Input Data**

This section describes how to input a graph for analysis by PROC NETWORK. Let \( G = (N, A) \) define a graph that contains a set \( N \) of nodes and a set \( A \) of links. Consider the directed graph shown in Figure 2.7.

*Figure 2.7 A Simple Directed Graph*

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

The LINKS= option in the PROC NETWORK statement defines the data table 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 table is expected to contain the following possible variables:

- **from**: the *from* node (can be numeric or character)
- **to**: the *to* node (can be numeric or character)

The links data table can also contain some combination of the following built-in variables (attributes):

- **auxweight**: the auxiliary link weight (must be numeric)
- **from**: the *from* node (can be numeric or character)
- **to**: the *to* node (can be numeric or character)
- **weight**: the link weight (must be numeric)

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

The data variable names can have any values that you want. If you use nonstandard names for a built-in variable, you must identify the variables by using the LINKSVAR statement, as described in the section “LINKSVAR Statement” on page 40.

In addition, the links data table can contain any number of user-defined additional variables (attributes). The attributes that are defined in the VARS= option in the LINKSVAR statement are carried over to the resulting output data tables.

For example, the two data tables that are created by the following DATA steps identify the same graph:

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

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

You can present these data tables to PROC NETWORK by using the following equivalent statements:
The directed graph $G$ that is shown in Figure 2.7 can be represented by the links data table, mycas.LinkSetIn, that is created by the following DATA step:

```
data mycas.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 tables. These statements do not run any algorithms, so the resulting output contains only the input graph.

```
proc network
  direction = directed
  links = mycas.LinkSetIn
  outNodes = mycas.NodeSetOut
  outLinks = mycas.LinkSetOut;
run;
```

The output data table mycas.NodeSetOut, shown in Figure 2.8, now contains the nodes that are read from the input links data table. The variable node shows the label associated with each node.

**Figure 2.8** Nodes Data Table 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 output data table mycas.LinkSetOut, shown in Figure 2.9, contains the links that were read from the input links data table. The variables from and to show the associated node labels.
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 duplicate links are removed. PROC NETWORK aggregates the attributes of each duplicate link by taking the minimum value (for each attribute). By default, DIRECTION=UNDIRECTED, so you can simply remove this option to declare the graph as undirected.

The following statements read in this graph, declare it as an undirected graph, and output the resulting links and nodes data tables:

```plaintext
proc network
   links = mycas.LinkSetIn
   outNodes = mycas.NodeSetOut
   outLinks = mycas.LinkSetOut;
run;
```

The progress of the procedure is shown in Figure 2.10. The log now shows the number of links that were declared as duplicates and aggregated.

![Figure 2.10](image)

The output data table mycas.NodeSetOut is equivalent to the one shown in Figure 2.8. However, the new links data table mycas.LinkSetOut, shown in Figure 2.11, contains two fewer links than before, because duplicates are aggregated.
Certain algorithms can perform more efficiently when you specify INTERNALFORMAT=THIN in their respective algorithm statement. However, when you specify this option, PROC NETWORK does not remove duplicate links. Instead, you should use the appropriate DATA steps to clean your data before calling PROC NETWORK.

Nodes Input Data

The NODES= option in the PROC NETWORK statement defines the data table that contains the list of nodes in the graph. This data table is used to assign node attributes.

The nodes data table is expected to contain the following variable:

- **node**: the node label (can be numeric or character)

The nodes data table can also contain the following built-in variable (attribute):

- **node**: the node label (can be numeric or character)
- **weight**: the node weight (must be numeric)

If weights are not given for algorithms that require node weights, all weights are assumed to be 1.

You can specify any value that you want for the data table variable name. If you use a nonstandard name for a built-in variable, you must identify the variable by using the NODESVAR statement, as described in the section “NODESVAR Statement” on page 43.

In addition, the nodes data table can contain any number of user-defined additional variables (attributes). The attributes that are defined in the VARS= option in the NODESVAR statement are carried over to the resulting output data tables.

The data table that you specify in the 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 you must define this node in the

**Figure 2.11** Links Data Table 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>
NODES= data table. The following statements produce a graph that has three links but four nodes, including the singleton node D:

```sas
data mycas.NodeSetIn;
   input node $ @@;
datalines;
A B C D
;

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

If you specify duplicate entries in the nodes data table, PROC NETWORK issues an error message and stops.

**Nodes 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 NODESSUBSET= option in the PROC NETWORK statement. You can use the nodes subset data table in conjunction with the PATH, SHORTESTPATH, NODESIMILARITY, or REACH statement. (See the sections “Path Enumeration” on page 110, “Shortest Path” on page 129, “Node Similarity” on page 105, and “Reach (Ego) Network” on page 122, respectively.) The nodes subset data table is expected to contain some combination of the following variables:

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

Table 2.5 shows how PROC NETWORK 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 2.5** Determining How to Process a Node

<table>
<thead>
<tr>
<th>Algorithm Type</th>
<th>Variable Designations</th>
<th>Example Shown In:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Shortest) path, Node similarity</td>
<td>A value of 1 for the source variable designates that the node is to be processed as a source; a value of 0 (or missing) represents no specification. The same values must 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 129 or “Example 2.16: Node Similarity for Link Prediction” on page 205</td>
</tr>
</tbody>
</table>
Table 2.5  (continued)

<table>
<thead>
<tr>
<th>Algorithm Type</th>
<th>Variable Designations</th>
<th>Example Shown In:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reach</td>
<td>A value greater than 0 defines a marker for the source subgraph to which this node belongs. All nodes that have the same marker are processed together as source nodes. A value of 0 (or missing) for the reach variable designates that the node is not to be processed. The reach identifiers must be consecutive integers starting from 1.</td>
<td>The section “Reach (Ego) Network” on page 122</td>
</tr>
</tbody>
</table>

The following code, which creates a nodes subset data table, might be used with the graph in Figure 2.7:

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

The data table mycas.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 (or node similarity) for the source-sink pairs in \( A;E \) \( g \) \( F \) \( g \) \( (\text{the crossproduct of subsets } \{A, E\} \times \{F\}) \)

Output Carryover Variables

Any link or node attribute can be carried over to the output results table by using the VARS= option in the LINKSVAR or NODESVAR statement. This is supported for all output tables except the table that is specified in the OUTNODES= option when used in conjunction with a distributed graph algorithm. The list of algorithms that operate on a distributed graph is described in the section “Execution Modes and Data Movement” on page 59.

The following DATA step creates a graph that has four nodes (with two additional attributes) and three links (with a weight attribute and two additional attributes):

```plaintext
data mycas.NodeSetIn;
  input node $ attrStr $ attrNum;
datalines;
A ThisIsA 13
B B 1
C LabelC 55.5
D NodeD 7;
;
data mycas.LinkSetIn;
```
The following statements read in this graph and output the resulting nodes and links data tables (including the carryover variables):

```sas
proc network
   nodes = mycas.NodeSetIn
   links = mycas.LinkSetIn
   outNodes = mycas.NodeSetOut
   outLinks = mycas.LinkSetOut;
   nodesVar
      vars = (attrStr attrNum);
   linksVar
      vars = (attrStr1 attrStr2);
run;
```

The nodes data table mycas.NodeSetOut, shown in Figure 2.12, contains the carryover variables that are defined in the VARS= option in the NODESVAR statement.

### Figure 2.12 Nodes Data Table with Carryover Variables

<table>
<thead>
<tr>
<th>Obs</th>
<th>node</th>
<th>attrStr</th>
<th>attrNum</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>ThisIsA</td>
<td>13.0</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>B</td>
<td>1.0</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>LabelC</td>
<td>55.5</td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td>NodeD</td>
<td>7.0</td>
</tr>
</tbody>
</table>

The links data table mycas.LinkSetOut, shown in Figure 2.13, contains the carryover variables that are defined in the VARS= option in the LINKS VAR statement, as well as the built-in variable weight.

### Figure 2.13 Links Data Table with Carryover Variables

<table>
<thead>
<tr>
<th>Obs</th>
<th>from</th>
<th>to</th>
<th>weight</th>
<th>attrStr1</th>
<th>attrStr2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>B</td>
<td>1</td>
<td>Link1</td>
<td>555-789-1234</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>C</td>
<td>2</td>
<td>Link2</td>
<td>556-453-7456</td>
</tr>
<tr>
<td>3</td>
<td>B</td>
<td>C</td>
<td>1</td>
<td>Link3</td>
<td>800-123-7787</td>
</tr>
</tbody>
</table>

As another example, consider the authority centrality calculation in the section “Authority in US Supreme Court Precedent” on page 13. Rather than performing a merge to combine the authority score with the case names, you can use the following statements to carry the case names directly to the output table:

```sas
proc network
   direction = directed
   nodes = mycas.Cases
   links = mycas.LinkSetInCourt
   outNodes = mycas.NodeSetOut;
   linksVar
      vars = (attrStr1 attrStr2);
run;
```
The output table mycas.NodeSetOut contains the same results that are shown in Figure 2.6 without requiring the additional merge step.

**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 STANDARDIZEDLABELS option in the PROC NETWORK statement to speed up this stage:

1. The links data table variables `from` and `to` are numeric.
2. The node and nodes subset data table variable `node` is numeric.
3. The node labels start from 0 and are consecutive nonnegative integers.

Consider the following links data table that uses numeric labels:

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

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

```plaintext
proc network
  links = mycas.LinkSetIn
  outNodes = mycas.NodeSetOut
  outLinks = mycas.LinkSetOut;
run;
```

The log is shown in Figure 2.14.

**Figure 2.14** PROC NETWORK Log: A Simple Undirected Graph
The output data table mycas.NodeSetOut, shown in Figure 2.15, contains the unique numeric node labels, \{0, 1, 3, 5\}.

**Figure 2.15** Nodes Data Table 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 table defines a graph that has six (not four) nodes:

```sas
proc network
    standardizedLabels
    links = mycas.LinkSetIn
    outNodes = mycas.NodeSetOut
    outLinks = mycas.LinkSetOut;
run;
```

The log that results from using standardized labels is shown in Figure 2.16.

**Figure 2.16** PROC NETWORK Log: A Simple Undirected Graph Using Standardized Labels

```
NOTE:  ---------------------------------------------------------------
NOTE: Running NETWORK.                                               
NOTE:  ---------------------------------------------------------------
NOTE: The number of nodes in the input graph is 6.                    
NOTE: The number of links in the input graph is 3.                    
NOTE: The Cloud Analytic Services server processed the request in 0.198155 seconds. 
NOTE: The data set MYCAS.NODESETOUT has 6 observations and 1 variables. 
NOTE: The data set MYCAS.LINKSETOUT has 3 observations and 3 variables. 
```

The output data table mycas.NodeSetOut, shown in Figure 2.17, now contains all node labels from 0 to 5, based on the assumptions when you use the STANDARDIZEDLABELS option.

**Figure 2.17** Nodes Data Table 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>2</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>5</td>
</tr>
</tbody>
</table>
Execution Modes and Data Movement

When you run PROC NETWORK, the algorithmic execution mode and the underlying data movement that is implemented (required) to support that execution mode depend on which algorithm you select.

For a single-machine CAS server, there is no data movement. The algorithm runs on the same machine where the data reside. For a multiple-machine CAS server, the assumption is that the data reside in parts on one or more of the machines in the server.

Certain algorithms run only in single-machine mode. In such cases, one particular machine (chosen randomly) is given the role of the processing machine, and the data from all the other machines are moved over to this processing machine.

Some algorithms use multiple machines, but each machine requires a global view of the input data. In such cases, each data part is repeated on all machines. Each machine processes a portion of the work across the entire graph. The resulting output tables are distributed tables.

Other algorithms use multiple machines and require only a portion of the data (distributed graphs). However, because the original data are usually randomly distributed, the first step is to shuffle data between machines such that the data are appropriately aligned for the particular algorithm’s needs. When the data are aligned correctly, each machine processes a part of the data and then iteratively merges results across the grid to obtain the final result. Again, the resulting output tables are distributed tables.

In addition, on each machine, some of these algorithms (as well as the input phase) take advantage of multicore chip technology by executing on multiple threads simultaneously. You can use the NTHREADS= option in the PROC NETWORK statement to specify the number of threads to use. The default is the number of cores on the machine that executes the process or the number of cores permissible based on your installation (whichever is less). Setting this option to a number greater than the number of available cores might hurt 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 some circumstances, the NETWORK procedure might use fewer threads than the specified number because the procedure’s internal algorithms have determined that a smaller number is preferable.

In the case of BY-group processing, the data must first be partitioned such that each observation within a BY group resides on the same machine. If the data are not already partitioned, PROC NETWORK shuffles the data appropriately as a first step. When the data are partitioned, the selected algorithm runs against the groups (on each machine) by using multiple threads (one group per thread). If the algorithm itself is a multithreaded algorithm, then it uses multiple threads (on each group) if and only if the value of the NTHREADS= option exceeds the number of groups assigned to the processing machine. In this case, the resulting output tables are distributed tables (partitioned by group). You can prepartition your input data by using the PARTITION= option in a DATA step. Prepartitioning avoids the need for PROC NETWORK to shuffle the data. This option is described in SAS Cloud Analytic Services: Language Reference.

The data movement and execution modes for each algorithm are listed in Table 2.6. The table uses the abbreviations SM (single machine), MM (multiple machines), and MT (multithreaded execution).

<table>
<thead>
<tr>
<th>Statement (and Options)</th>
<th>Data Movement</th>
<th>Processing Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>BICONNECTEDCOMPONENTS</td>
<td>Moved to SM</td>
<td>SM</td>
</tr>
</tbody>
</table>
### Table 2.6 (continued)

<table>
<thead>
<tr>
<th>Statement (and Options)</th>
<th>Data Movement</th>
<th>Processing Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>CENTRALITY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BETWEEN=, CLOSE=</td>
<td>Repeated on MM</td>
<td>MM (MT)</td>
</tr>
<tr>
<td>CLUSTERINGCOEF, AUTH=, HUB=, EIGEN=</td>
<td>Moved to SM</td>
<td>SM (MT)</td>
</tr>
<tr>
<td>DEGREE, INFLUENCE=</td>
<td>Moved to SM</td>
<td>SM</td>
</tr>
<tr>
<td>DEGREE (with DISTRIBUTED=)</td>
<td>Shuffled across MM</td>
<td>MM</td>
</tr>
<tr>
<td>CLIQUE</td>
<td>Moved to SM</td>
<td>SM</td>
</tr>
<tr>
<td>COMMUNITY ALGORITHM=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOUVAIN, LABELPROP</td>
<td>Moved to SM</td>
<td>SM</td>
</tr>
<tr>
<td>PARALLEL LABELPROP†</td>
<td>Shuffled across MM</td>
<td>MM (MT)</td>
</tr>
<tr>
<td>CONNECTED COMPONENTS ALGORITHM=</td>
<td>Moved to SM</td>
<td>SM</td>
</tr>
<tr>
<td>DFS, UNIONFIND</td>
<td>Shuffled across MM</td>
<td>MM</td>
</tr>
<tr>
<td>PARALLEL†</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORE</td>
<td>Moved to SM</td>
<td>SM</td>
</tr>
<tr>
<td>CYCLE ALGORITHM=</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BACKTRACK</td>
<td>Moved to SM</td>
<td>SM (MT)</td>
</tr>
<tr>
<td>BUILD</td>
<td>Moved to SM</td>
<td>SM</td>
</tr>
<tr>
<td>NODESIMILARITY</td>
<td>Repeated on MM</td>
<td>MM (MT)</td>
</tr>
<tr>
<td>PATH</td>
<td>Repeated on MM</td>
<td>MM (MT)</td>
</tr>
<tr>
<td>PATTERNMATCH</td>
<td>Moved to SM</td>
<td>MM (MT)</td>
</tr>
<tr>
<td>REACH</td>
<td>Repeated on MM</td>
<td>MM (MT)</td>
</tr>
<tr>
<td>SHORTESTPATH</td>
<td>Repeated on MM</td>
<td>MM (MT)</td>
</tr>
<tr>
<td>SUMMARY (other than shortest path)</td>
<td>Moved to SM</td>
<td>SM</td>
</tr>
<tr>
<td>SHORTESTPATH=</td>
<td>Moved to SM</td>
<td>SM (MT)</td>
</tr>
<tr>
<td>TRANSITIVE CLOSURE</td>
<td>Moved to SM</td>
<td>SM</td>
</tr>
</tbody>
</table>

† To specify a distributed graph algorithm, you can also set DISTRIBUTED=TRUE on the PROC NETWORK statement.

Because of communication costs, increasing the number of machines does not guarantee faster execution, especially when you are dealing with small graphs. For all the documentation examples, unless otherwise noted, the CAS session is configured for four worker nodes, each having 32 cores. For general information about CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*.

### Numeric Limitations

Extremely large or extremely small numerical values might cause computational difficulties for some of the algorithms in PROC NETWORK. 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 NETWORK issues an error message. The value of the threshold limit is different for each algorithm and depends on the operating environment. The threshold limits are listed in Table 2.7, where $M$ is defined as the largest number that can be represented by a double.
Table 2.7  Threshold Limits by Statement

<table>
<thead>
<tr>
<th>Statement (and Options)</th>
<th>Graph Links</th>
<th>Graph Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>weight</td>
<td>auxweight</td>
</tr>
<tr>
<td>CENTRALITY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AUTH=, EIGEN=, HUB=</td>
<td>1E20</td>
<td></td>
</tr>
<tr>
<td>BETWEEN=, CLOSE=</td>
<td>√M</td>
<td>√M</td>
</tr>
<tr>
<td>INFLUENCE=</td>
<td>√M</td>
<td></td>
</tr>
<tr>
<td>COMMUNITY</td>
<td>√M</td>
<td></td>
</tr>
<tr>
<td>CYCLE</td>
<td>√M</td>
<td></td>
</tr>
<tr>
<td>PATH</td>
<td>√M</td>
<td>√M</td>
</tr>
<tr>
<td>REACH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHORTESTPATH</td>
<td>√M</td>
<td>√M</td>
</tr>
<tr>
<td>SUMMARY</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIAMETERAPPROX=, SHORTESTPATH=</td>
<td>√M</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 all the algorithms in PROC NETWORK, there is no valid interpretation for a missing value. If the user data contain a missing value, PROC NETWORK issues an error message.

**Negative Link Weights**

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

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

**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 NETWORK can handle any graph whose numbers of nodes and links are each less than or equal to 2,147,483,647 (the maximum that a 32-bit integer can represent). This maximum also applies to 64-bit
systems. For graphs that contain two billion nodes (or links), memory restrictions 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 86.

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

Two exceptions to this limitation are the parallel union-find algorithm for finding connected components and the parallel label propagation algorithm for community detection. Both of these algorithms are limited to 2,147,483,647 links per machine in your session configuration (rather than total links). These algorithms are still limited to 2,147,483,647 total nodes.

Common Notation and Assumptions

This section briefly introduces some common notation and assumptions that are 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 2.8.

<table>
<thead>
<tr>
<th>Graph Direction</th>
<th>Default</th>
<th>INCLUDESELFLINK</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 you cannot break into disconnected pieces by deleting any single node (and its incident links). An articulation point of a graph is a node 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 $i \in N$ that, if removed, breaks 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 where articulation points are important include airline hubs, electric circuits, network wires, protein bonds, traffic routers, and many other industrial applications.

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

The results of the biconnected components algorithm are written to the output links data table that you specify in the OUTLINKS= option in the PROC NETWORK statement. For each link in the links data table, the variable biconcomp identifies its component. The component identifiers are numbered sequentially, starting from the value of the INDEXOFFSET= option in the PROC NETWORK statement. The results of the articulation points are written to the output nodes data table that you specify in the OUTNODES= option in
the PROC NETWORK statement. For each node in the nodes data table, the variable artpoint is either 1 (if the node is an articulation point) or 0 (otherwise).

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

**Biconnected Components of a Simple Undirected Graph**

This section illustrates the use of the biconnected components algorithm on the simple undirected graph $G$ shown in Figure 2.18.

*Figure 2.18 A Simple Undirected Graph $G***

The undirected graph $G$ can be represented by the following links data table, mycas.LinkSetInBiCC:

```plaintext
data mycas.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 for $G$ and output the results in the data tables mycas.LinkSetOut and mycas.NodeSetOut:

```plaintext
proc network
  links = mycas.LinkSetInBiCC
  outLinks = mycas.LinkSetOut 
  outNodes = mycas.NodeSetOut;
```
The output data table `mycas.LinkSetOut` contains the biconnected components of the input graph, as shown in Figure 2.19.

**Figure 2.19** 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>1</td>
</tr>
<tr>
<td>A</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>G</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>3</td>
</tr>
<tr>
<td>G</td>
<td>I</td>
<td>4</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
<td>4</td>
</tr>
<tr>
<td>H</td>
<td>I</td>
<td>4</td>
</tr>
</tbody>
</table>

The output data table `mycas.NodeSetOut` contains the articulation points of the input graph, as shown in Figure 2.20.

**Figure 2.20** Articulation Points of a Simple Undirected Graph

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

The biconnected components are shown graphically in Figure 2.21 and Figure 2.22.
For a more detailed example, see “Example 2.1: Articulation Points in a Terrorist Network” on page 153.
Centrality

In general terms, the *centrality* of a node or link in a graph gives some indication of its relative importance within the 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 NETWORK to calculate several of these metrics. The options for this statement are described in the section “CENTRALITY Statement” on page 28.

The following sections describe each of the possible centrality metrics that you can calculate in PROC NETWORK.

**Degree Centrality**

The *degree* (or *valency*) of a node \( v \) in an undirected graph is the number of links that are incident to node \( v \), with self-links counted twice. 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 to that node. For an undirected graph, the terms *degree* and *out-degree* are interchangeable. *Degree centrality* is simply the in- or out-degree of a node and can be interpreted as some form of the node’s 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 who have a higher out-degree can lead to faster and more widespread transmission. In a friendship network, in-degree often indicates popularity.

Degree centrality is calculated using the value specified for the **DEGREE** option in the CENTRALITY statement. The results are provided in the node output data table that you specify in the **OUTNODERS=** option in the PROC NETWORK statement.

To specify a distributed graph algorithm, you can set **DISTRIBUTED=TRUE** on the PROC NETWORK statement.

The algorithm that PROC NETWORK uses 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 2.7 with the data table mycas.LinkSetIn, which is defined in the section “Links Input Data” on page 50. The following statements calculate the degree centrality for both in- and out-degrees:

```plaintext
proc network
direction = directed
links = mycas.LinkSetIn
outNodes = mycas.NodeSetOut;
    centrality
degree;
run;
```

The nodes data table mycas.NodeSetOut now contains the degree centrality of the input graph. For a directed graph, the data table provides the in-degree (the **centr_degree_in** variable), the out-degree (the **centr_degree_out** variable), and the degree that is the sum of in- and out-degrees (the **centr_degree** variable). This data table is shown in Figure 2.23.
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 of link $(u, v)$, and let $w_u$ define the node weight of 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 indicates potential influence, performance, or ability to transfer knowledge.

Influence centrality is calculated using the value of the INFLUENCE= option in the CENTRALITY statement. The results are provided in the node output data table that you specify in the OUTNODES= option in the PROC NETWORK statement.

The algorithm that PROC NETWORK uses 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 2.7. 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 network
direction = directed
links = mycas.LinkSetIn
outNodes = mycas.NodeSetOut;
centrality
   influence = unweight;
run;
```
The nodes data table mycas.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 table is shown in Figure 2.24.

**Figure 2.24** 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 2.2: Influence Centrality for Project Groups in a Research Department” on page 155.

**Clustering Coefficient**

The clustering coefficient of 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 CLUSTERINGCOEF option in the CENTRALITY statement. The results are provided in the node output data table that you specify in the OUTNODEN= option in the PROC NETWORK statement.

The algorithm that PROC NETWORK 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 2.25.
Define the three links data tables as follows:

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

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

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

The following statements use three calls to PROC NETWORK to calculate the clustering coefficients of each graph:

```plaintext
proc network
  links = mycas.LinkSetInCC1
  outNodes = mycas.NodeSetOut1;
  centrality
    clusteringCoef;
run;

proc network
  links = mycas.LinkSetInCC2
  outNodes = mycas.NodeSetOut2;
  centrality
    clusteringCoef;
run;
```
Chapter 2: The NETWORK Procedure

```
proc network
   links = mycas.LinkSetInCC3
   outNodes = mycas.NodeSetOut3;
   centrality
      clusteringCoeff;
run;
```

The nodes data tables provide the clustering coefficients of each graph (the `centr_cluster` variable), as shown in Figure 2.26 through Figure 2.28.

**Figure 2.26** Clustering Coefficient of a Simple Undirected Graph 1

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

**Figure 2.27** Clustering Coefficient of a Simple Undirected Graph 2

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

**Figure 2.28** 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>C</td>
<td>0</td>
</tr>
<tr>
<td>B</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. The higher the closeness value of a particular node, the faster the information will spread from that node to other nodes.

Define $d_{uv}$ to be the shortest path distance from node $u$ to node $v$, with link weight defined by the AUXWEIGHT= option in the LINKSVAR statement. If the auxiliary link weight is not specified, then the link weight defaults to $1/w$, where $w$ represents the weight assigned by the WEIGHT= option in the LINKSVAR statement. By default, this means that a higher link weight implies a stronger relationship between its nodes (similar to other centrality metrics).
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 \texttt{CLOSENOPATH=} 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

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

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 2.9.

<table>
<thead>
<tr>
<th>CLOSENOPATH=</th>
<th>$p$</th>
<th>$n(u)$</th>
<th>$s(u)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIAMETER</td>
<td>$\max_{(i,j) \in A} {d_{ij} : d_{ij} &lt; \infty}$ + 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, $R^{\text{out}}(u) = \{ v \in N : d_{uv} < \infty \}$ is the set of reachable nodes from node $u$, whereas $R^{\text{in}}(u) = \{ v \in N : d_{vu} < \infty \}$ is the set of nodes from which there is a finite path to node $u$. The set of unreachable nodes from node $u$ is $N \setminus R^{\text{out}}(u) = \{ v \in N : d_{uv} = \infty \}$, whereas the set of nodes from which there is no finite path to node $u$ is $N \setminus R^{\text{in}}(u) = \{ v \in N : d_{vu} = \infty \}$.

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

$$ C^{\text{out}}_c(u) = s^{\text{out}}(u) \left( \frac{n^{\text{out}}(u)}{\sum_{v \in R^{\text{out}}(u)} d_{uv} + |N \setminus R^{\text{out}}(u)|} \right) $$

where $n^{\text{out}}(u)$ defines the number of nodes that are considered in calculating the average and $s^{\text{out}}(u)$ is a scaling factor, as shown in Table 2.10.

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

$$ C^{\text{in}}_c(u) = s^{\text{in}}(u) \left( \frac{n^{\text{in}}(u)}{\sum_{v \in R^{\text{in}}(u)} d_{vu} + |N \setminus R^{\text{in}}(u)|} \right) $$

where $n^{\text{in}}(u)$ defines the number of nodes that are considered in calculating the average and $s^{\text{in}}(u)$ is a scaling factor, as shown in Table 2.10.
The overall closeness centrality for directed graphs is calculated as

\[
C_c(u) = \frac{C^\text{out}_c(u) + C^\text{in}_c(u)}{2}
\]

**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 the 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 CLOSENOPATH=HARMONIC option.

Closeness centrality is calculated using the value of the CLOSE= option in the CENTRALITY statement. The results are provided in the node output data table that you specify in the OUTNODES= option in the PROC NETWORK 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 NETWORK 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), the algorithm uses multiple threads and multiple machines (depending on your session configuration and license).

Consider again the directed graph in Figure 2.7 with the data table mycas.LinkSetIn, which is defined in the section “Links Input Data” on page 50. The following statements calculate the closeness centrality for both the weighted and unweighted graphs:

```sas
proc network
direction = directed
links = mycas.LinkSetIn
outNodes = mycas.NodeSetOut;
centrality
  close = both;
run;
```
The nodes data table mycas.NodeSetOut now contains the weighted and unweighted directed closeness centrality of the input graph. The output data table 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 table is shown in Figure 2.29.

**Figure 2.29** 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>0.38835</td>
<td>0.00000</td>
<td>0.77670</td>
<td>0.22222</td>
<td>0.00000</td>
<td>0.44444</td>
</tr>
<tr>
<td>B</td>
<td>0.55134</td>
<td>0.30000</td>
<td>0.80268</td>
<td>0.33333</td>
<td>0.22222</td>
<td>0.44444</td>
</tr>
<tr>
<td>C</td>
<td>0.38260</td>
<td>0.34043</td>
<td>0.42478</td>
<td>0.27885</td>
<td>0.25000</td>
<td>0.30769</td>
</tr>
<tr>
<td>D</td>
<td>0.40113</td>
<td>0.41202</td>
<td>0.39024</td>
<td>0.29178</td>
<td>0.30769</td>
<td>0.27586</td>
</tr>
<tr>
<td>E</td>
<td>0.45962</td>
<td>0.43439</td>
<td>0.48485</td>
<td>0.32000</td>
<td>0.32000</td>
<td>0.32000</td>
</tr>
<tr>
<td>F</td>
<td>0.44357</td>
<td>0.50314</td>
<td>0.38400</td>
<td>0.30725</td>
<td>0.34783</td>
<td>0.26667</td>
</tr>
<tr>
<td>G</td>
<td>0.56402</td>
<td>0.79470</td>
<td>0.33333</td>
<td>0.32500</td>
<td>0.40000</td>
<td>0.25000</td>
</tr>
<tr>
<td>H</td>
<td>0.41663</td>
<td>0.48290</td>
<td>0.35036</td>
<td>0.27885</td>
<td>0.30769</td>
<td>0.25000</td>
</tr>
<tr>
<td>I</td>
<td>0.30227</td>
<td>0.60453</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 in 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 through 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

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

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 \(u\). As with closeness centrality, the shortest path is calculated with respect to the link weight defined by the AUXWEIGHT= option in the LINKSVAR statement. If the auxiliary link weight is not specified, then the link weight defaults to \(1/w\), where \(w\) represents the weight assigned by the WEIGHT= option in the LINKSVAR statement. By default, this means that a higher link weight implies a stronger relationship between the link’s nodes.

The formula for link betweenness centrality is

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

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

By default, this metric is normalized by dividing by the number of pairs of nodes, not including \(u\), which is \((|N| - 1)(|N| - 2)\). You can disable this normalization by using the BETWEENNORM= 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. You can accomplish this by simply using the LINKSVAR statement to reverse the interpretation of *from* and *to*. 
Betweenness centrality is calculated using the value of the BETWEEN= option in the CENTRALITY statement. The node betweenness results are provided in the node output data table that you specify in the OUTNODES= option in the PROC NETWORK statement. The link betweenness results are provided in the link output data table that you specify in the OUTLINKS= option in the PROC NETWORK statement. As with closeness centrality, if BETWEEN=WEIGHT (or BOTH), then the calculation of shortest paths is performed using the weighted graph.

The algorithm that PROC NETWORK uses to compute betweenness centrality relies on calculating shortest paths for all source-sink pairs and runs in time $O(|N| 	imes (|N| \log |N| + |A|))$. Therefore, it is not expected to scale to very large graphs. As with closeness centrality, because shortest path computations can be calculated independently (for each source node), the algorithm uses multiple threads and multiple machines (depending on your session configuration and license). When closeness and betweenness centrality are run together, PROC NETWORK calculates both metrics in one pass.

Consider again the directed graph in Figure 2.7 with data table mycas.LinkSetIn, which is defined in the section “Links Input Data” on page 50. The following statements calculate the betweenness centrality for both the weighted and unweighted graphs:

```plaintext
proc network
direction   = directed
links       = mycas.LinkSetIn
outLinks    = mycas.LinkSetOut
outNodes    = mycas.NodeSetOut;
centrality
   between = both;
run;
```

The nodes data table mycas.NodeSetOut now contains the weighted (the centr_between_wt variable) and unweighted (the centr_between_unwt variable) node betweenness centrality of the input graph. This data table is shown in Figure 2.30.

**Figure 2.30** 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>0.07143</td>
<td>0.07738</td>
</tr>
<tr>
<td>C</td>
<td>0.00000</td>
<td>0.00595</td>
</tr>
<tr>
<td>D</td>
<td>0.01786</td>
<td>0.00595</td>
</tr>
<tr>
<td>E</td>
<td>0.17857</td>
<td>0.17857</td>
</tr>
<tr>
<td>F</td>
<td>0.26786</td>
<td>0.26786</td>
</tr>
<tr>
<td>G</td>
<td>0.21429</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 links data table mycas.LinkSetOut contains the weighted (the centr_between_wt variable) and unweighted (the centr_between_unwt variable) link betweenness centrality of the input graph. This data table is shown in Figure 2.31.
Centrality

Figure 2.31  Link Betweenness Centrality of a Simple Directed Graph

<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>A</td>
<td>B</td>
<td>1</td>
<td>0.08929</td>
<td>0.09524</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>2</td>
<td>0.01786</td>
<td>0.02381</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>4</td>
<td>0.03571</td>
<td>0.02381</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>1</td>
<td>0.01786</td>
<td>0.01786</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>2</td>
<td>0.03571</td>
<td>0.04167</td>
</tr>
<tr>
<td>B</td>
<td>F</td>
<td>5</td>
<td>0.14286</td>
<td>0.14286</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
<td>1</td>
<td>0.10714</td>
<td>0.11310</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
<td>1</td>
<td>0.10714</td>
<td>0.09524</td>
</tr>
<tr>
<td>E</td>
<td>D</td>
<td>1</td>
<td>0.05357</td>
<td>0.05357</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>2</td>
<td>0.21429</td>
<td>0.21429</td>
</tr>
<tr>
<td>F</td>
<td>G</td>
<td>6</td>
<td>0.32143</td>
<td>0.32143</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
<td>1</td>
<td>0.12500</td>
<td>0.12500</td>
</tr>
<tr>
<td>H</td>
<td>G</td>
<td>2</td>
<td>0.01786</td>
<td>0.01786</td>
</tr>
<tr>
<td>G</td>
<td>I</td>
<td>1</td>
<td>0.12500</td>
<td>0.12500</td>
</tr>
<tr>
<td>H</td>
<td>I</td>
<td>3</td>
<td>0.01786</td>
<td>0.01786</td>
</tr>
</tbody>
</table>

For more detailed examples, see “Example 2.3: Betweenness and Closeness Centrality for Computer Network Topology” on page 159 and “Example 2.4: Betweenness and Closeness Centrality for Project Groups in a Research Department” on page 162.

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. The 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 represented as

\[ x_i = \frac{1}{\lambda} \sum_{j \in \delta_i} w_{ij} x_j \]

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 \( w_{ij} \) 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 using the value that you specify in the EIGEN= option in the CENTRALITY statement. The results are provided in the node output data table that you specify in the OUTNODES= option in the PROC NETWORK statement.

The following example illustrates the use of eigenvector centrality on the undirected graph \( G \) shown in Figure 2.32.
Figure 2.32 Eigenvector Centrality Example of a Simple Undirected Graph

The graph can be represented by the following links data table, mycas.LinkSetIn:

```plaintext
data mycas.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:

```plaintext
proc network
   links = mycas.LinkSetIn
   outNodes = mycas.NodeSetOut;
   centrality
      eigen = unweight;
run;
```

The output data table mycas.NodeSetOut now contains the eigenvector centrality of each node, as shown in Figure 2.33.
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 2.5: Eigenvector Centrality for Word Sense Disambiguation” on page 165.

**Hub and Authority Scoring**

*Hub and authority centrality* was originally developed by Kleinberg (1998) to rank the importance of web pages. Certain web pages (called hubs) are important in the sense that they point to many important pages. On the other hand, some web pages (called authorities) are important because they are pointed to by many important pages. 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, you can apply it 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 US Supreme Court Precedent” on page 13 presents 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,

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

where \(x_i\) is the authority centrality of node \(i\), \(y_i\) is the hub centrality of node \(i\), \(w_{ij}\) 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 NETWORK provides two algorithms: the Jacobi-Davidson algorithm and the power method. You use the EIGENALGORITHM= option in the CENTRALITY statement to specify which algorithm to use. JACOBI DAVIDSON, which is the default, specifies the Jacobi-Davidson algorithm, 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 2.34. 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 2.34** Hub and Authority Centrality Example of a Simple Directed Graph

The graph can be represented by the following links data table, mycas.LinkSetIn:

```r
data mycas.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:

```r
proc network
direction = directed
links = mycas.LinkSetIn
outNodes = mycas.NodeSetOut;
centrality
  hub = unweight
  auth = unweight;
run;
```
The output data table mycas.NodeSetOut now contains the hub and authority scores of each node, as shown in Figure 2.35.

<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, and K have high hub scores because they all point to good authority nodes B and E.

### PageRank Centrality

PageRank centrality, named after Google’s cofounder Larry Page, is a variant of eigenvector centrality but is more effective on directed graphs. For the calculation of PageRank centrality, PROC NETWORK implements a version of the power method, as described in Langville and Meyer (2006).

The PageRank algorithm models the stationary distribution of a Markov process, assuming that each node is a web page and each link is a hyperlink from one page to another. A web surfer might choose a random link on every page or might jump to a random page on the whole web with some probability. The PageRank of a page is the percentage of time the surfer spends on the page. Mathematically, it is defined as

\[
P_i = \frac{1 - d}{|N|} + d \sum_{j \in \delta_{i}^{in}} \frac{P_j}{w_j} \]

\[
w_{uv} = \sum_{(u,v) \in A} w_{uv}
\]

where \( P_i \) is the PageRank score of node \( i \), \( \delta_{i}^{in} \) is the set of nodes that link to node \( i \), \( w_j \) is the sum of the outbound link weights from node \( j \), \( d \) is a damping factor that ranges from 0 to 1, and \( 1 - d \) is the probability of jumping to a random page on the whole web.

When the damping factor \( d \) is 1, PageRank centrality resolves to the same definition as eigenvector centrality. The damping factor is an important concept in the PageRank algorithm. Without the damping factor, the pages without any incoming links would have a PageRank score of zero, so these pages cannot be distinguished by PageRank scores. The damping factor also affects the convergence of the power method. Typically, many more iterations are required as the damping factor is set closer to 1.

When the PageRank algorithm is applied to undirected graphs, each undirected link is treated as two directed links that go in both directions.
This section concludes with an example that illustrates the use of PageRank centrality on the directed graph $G$ that is shown in Figure 2.34 in the previous section, “Hub and Authority Scoring” on page 77.

The following statements compute the PageRank score by using the same data as in the previous section:

```plaintext
proc network
  direction = directed
  links = mycas.LinkSetIn
  outNodes = mycas.NodeSetOut;
  centrality
    pagerankAlpha = 0.85
    pagerank = unweight;
run;
```

The output data table `mycas.NodeSetOut` now contains the PageRank score of each node, as shown in Figure 2.36.

**Figure 2.36** PageRank Centrality of a Simple Directed Graph (Damping Factor = 0.85)

<table>
<thead>
<tr>
<th>node</th>
<th>centr_pagerank_unwt</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0.38440</td>
</tr>
<tr>
<td>C</td>
<td>0.34291</td>
</tr>
<tr>
<td>E</td>
<td>0.08089</td>
</tr>
<tr>
<td>D</td>
<td>0.03909</td>
</tr>
<tr>
<td>F</td>
<td>0.03909</td>
</tr>
<tr>
<td>A</td>
<td>0.03278</td>
</tr>
<tr>
<td>G</td>
<td>0.01617</td>
</tr>
<tr>
<td>H</td>
<td>0.01617</td>
</tr>
<tr>
<td>I</td>
<td>0.01617</td>
</tr>
<tr>
<td>J</td>
<td>0.01617</td>
</tr>
<tr>
<td>K</td>
<td>0.01617</td>
</tr>
</tbody>
</table>

The PageRank score can be thought of as the percentage of time a web surfer spends on a web page. In this example, 38.4% of the surfer’s time is spent on page (node) B, and 34.3% of the surfer’s time is spent on page (node) C. Notice that node C has a higher PageRank score than node E, even though it has fewer links. The reason is that node C has an incoming link from high-value node B, whereas node E has no incoming links from high-value nodes.

The following statements modify the damping factor to be 0.999 to show the changes in PageRank scores:

```plaintext
proc network
  direction = directed
  links = mycas.LinkSetIn
  outNodes = mycas.NodeSetOut;
  centrality
    pagerankAlpha = 0.999
    pagerank = unweight;
run;
```

The new PageRank scores are shown in Figure 2.37.
Figure 2.37  PageRank Centrality of a Simple Directed Graph (Damping Factor = 0.999)

<table>
<thead>
<tr>
<th>node</th>
<th>centr_pagerank_unwrt</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>0.49905</td>
</tr>
<tr>
<td>C</td>
<td>0.49866</td>
</tr>
<tr>
<td>E</td>
<td>0.00070</td>
</tr>
<tr>
<td>D</td>
<td>0.00035</td>
</tr>
<tr>
<td>F</td>
<td>0.00035</td>
</tr>
<tr>
<td>A</td>
<td>0.00029</td>
</tr>
<tr>
<td>G</td>
<td>0.00012</td>
</tr>
<tr>
<td>H</td>
<td>0.00012</td>
</tr>
<tr>
<td>I</td>
<td>0.00012</td>
</tr>
<tr>
<td>J</td>
<td>0.00012</td>
</tr>
<tr>
<td>K</td>
<td>0.00012</td>
</tr>
</tbody>
</table>

As expected, the scores of all nodes except B and C are getting close to zero. This is because after the web surfer arrives at pages B and C, the surfer cannot jump off the pages because there are no outbound links from nodes B and C to other nodes.

Clique Enumeration

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 that is added. Finding cliques in graphs has applications in many industries, including bioinformatics, social networks, electrical engineering, and chemistry.

You can find the maximal cliques of an input graph by using the CLIQUE statement. The options for this statement are described in the section “CLIQUE Statement” on page 32. The clique algorithm works only with undirected graphs.

The results of the clique algorithm are written to the output data table that you specify in the OUT= option in the CLIQUE statement. Each node of each clique is listed in the output data table along with the variable clique to identify the clique to which it belongs. The clique identifiers are numbered sequentially, starting from the value of the INDEXOFFSET= option in the PROC NETWORK statement. A node can appear multiple times in this data table if it belongs to multiple cliques.

The algorithm that PROC NETWORK 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 usually does not scale to very large graphs.

Maximal Cliques of a Simple Undirected Graph

This section illustrates the use of the clique algorithm on the simple undirected graph $G$ shown in Figure 2.38.
The undirected graph $G$ can be represented by the following links data table, mycas.LinkSetIn:

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

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

```plaintext
proc network
  links = mycas.LinkSetIn;
  clique
    out = mycas.Cliques
    maxCliques = all;
run;

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

The output data table mycas.Cliques now contains the maximal cliques of the input graph, as shown in Figure 2.39.
In addition, the output data table `mycas.CliqueSizes` contains the number of nodes in each clique, as shown in Figure 2.40.

**Figure 2.40** Sizes of Maximal Cliques of a Simple Undirected Graph

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

The maximal cliques are shown graphically in Figure 2.41 and Figure 2.42.

**Figure 2.41** Maximal Cliques $C^1$ and $C^2$

\[ C^1 = \{0, 1, 2, 3, 4\} \quad C^2 = \{0, 2, 5, 6\} \]
Community Detection

Community detection partitions a graph into communities such that the nodes within the community subgraphs are more densely connected than the nodes from different communities.

In PROC NETWORK, you can specify community detection by using the COMMUNITY statement. The options for this statement are described in the section “COMMUNITY Statement” on page 33.

PROC NETWORK 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.

When you specify ALGORITHM=PARALLELLABELPROP in the COMMUNITY statement, community detection supports only directed graphs and the option DIRECTION=DIRECTED has to be explicitly set. If the data are presented as an undirected graph—that is, each pair of nodes \( \{i, j\} \) representing an undirected link is listed only once—this graph has to be transformed to a directed graph containing links in both directions, \( (i, j) \) and \( (j, i) \), before the COMMUNITY statement is used. When you specify ALGORITHM=LOUVAIN or ALGORITHM=LABELPROP 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. If you use this strategy, the weight of the undirected link between \( i \) and \( j \) is 7.5.

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 of community detection. Modularity is a measure of the quality of a division of a graph into communities. The modularity of a division is 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

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

$$w = \sum_{(u,v) \in A} w_{uv}$$

$$w_u = \sum_{v \in \delta_u} w_{uv}$$

where $Q$ is the modularity, $w_{uv}$ is the link weight between nodes $u$ and $v$ (or 0 if $(u, v) \notin A$), $\delta_u$ is the set of nodes that connect 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 RANDOMFACTOR= or RANDOMSEED= option in the COMMUNITY statement.
Memory Requirement

When you specify INTERNALFORMAT=THIN in the PROC NETWORK statement and ALGORITHM=LOUVAIN or ALGORITHM=LABELPROP in the COMMUNITY statement, the memory (number of bytes) that is 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 INTERNALFORMAT=THIN and ALGORITHM=PARALLELABELPROP, the memory required for community detection is approximately twice this amount (in a single-machine configuration).

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

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

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

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

PROC NETWORK uses significantly more memory if INTERNALFORMAT=FULL. It is recommended that you use INTERNALFORMAT=THIN when you perform community detection on large graphs.

Graph Direction

If you specify ALGORITHM=PARALLELABELPROP in the COMMUNITY statement, community detection supports only directed graphs. For a directed graph, the algorithm finds communities based on the information flow along the directed links. That is, the algorithm propagates the community identifier along the outgoing links of a node. Therefore, nodes are likely to be in the same community if they form cycles along the outgoing links. If the directed graph lacks this cycle structure, the nodes are likely to continue 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. This algorithm can be used on an undirected graph as well. First, the undirected graph has to be modeled by a directed graph, for example by a graph where each undirected edge is represented by a pair of directed edges in opposite directions (each carrying the same or half of the weight of the undirected edge). The results have to be interpreted with the understanding that they come from processing a directed graph model of the undirected graph.

Large Communities

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 NETWORK provides two approaches to alleviate the imbalance in the number of nodes across communities: one uses the RECURSIVE option, and the other uses the RESOLUTIONLIST= option.
Recursive
You can apply the RECURSIVE option to recursively break large communities into smaller ones. At the first step, PROC NETWORK 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 criterion. If the community result satisfies this criterion, PROC NETWORK stops iterations and outputs results. Otherwise, it treats each large community as an independent graph and recursively performs community detection on top of it.

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

Resolution List
The second way to combat the imbalance, provided that you have specified ALGORITHM=LOUVAIN in the COMMUNITY statement, is to specify a larger value than the default value of 1 for the RESOLUTIONLIST= option. When ALGORITHM=LOUVAIN, the value that is specified for the RESOLUTIONLIST= 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 NETWORK 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. Because of 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 RESOLUTIONLIST=1 0.7 0.5 and the other with RESOLUTIONLIST=1 0.5.

If you specify ALGORITHM=PARALLEL LABELPROP in the COMMUNITY statement, the resolution value can be interpreted as the minimal density of a community 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 is likely to result 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 LABELPROP, the NETWORK procedure performs community detection multiple times, each time with a different resolution value. This is equivalent to calling PROC NETWORK several times, each time with a different (single) resolution value specified for the RESOLUTIONLIST= option.

If you specify ALGORITHM=PARALLEL LABELPROP in the COMMUNITY statement, the value that is specified in the RESOLUTIONLIST= 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 NETWORK again. “Example 2.6: Community Detection on Zachary’s Karate Club Data” on page 168 shows the use of the RESOLUTIONLIST= option in finding communities.

**Large Graphs**

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

- Use INTERNALFORMAT=THIN instead of INTERNALFORMAT=FULL. This enables PROC NETWORK to store the data in memory compactly.
- Use the LINKREMOVALRATIO= option to remove unimportant links. This practice can often dramatically improve the run time of large graphs.

**Output Data Tables**

Community detection produces up to six output data tables. In these data tables, if you specify ALGORITHM=LOUVAIN in the COMMUNITY statement, resolution level numbers appear in decreasing order of the values that are specified in the RESOLUTIONLIST= option. That is, resolution level 1 corresponds to the largest value specified in the RESOLUTIONLIST= option, and resolution level K corresponds to the smallest value specified in the RESOLUTIONLIST= option. For example, if RESOLUTIONLIST=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.

The community identifiers are numbered sequentially, starting from the value of the INDEXOFFSET= option in the PROC NETWORK statement.

**OUTNODES= Data Table**

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

- node: the 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 RESOLUTIONLIST= option.

**OUTLINKS= Data Table**

The OUTLINKS= data table describes the community identifier of each link. If multiple resolution values have been specified, the data table reports the community identifier of each link at each resolution level. If a particular link contains a from node and a to node assigned to different communities, then the community identifier is the missing indicator (.). This data table contains the following columns:

- from: the from node label
- to: the to 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 RESOLUTIONLIST= option.

**OUTLEVEL= Data Table**

The OUTLEVEL= data table 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

**OUTCOMMUNITY= Data Table**

The OUTCOMMUNITY= data table 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

**OUTOVERLAP= Data Table**

The OUTOVERLAP= data table 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 table is computationally expensive to produce, and it requires a large amount of disk space. Therefore, the data table contains only results that correspond to the smallest value of the RESOLUTIONLIST= option. The data table contains the following columns:

- node: the node label
- community: community identifier
- intensity: intensity of the node that belongs to the community
OUTCOMMLINKS= Data Table
The OUTCOMMLINKS= data table describes how communities are connected. It 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

Community Detection on an Undirected Simple Graph
This section illustrates the use of the community detection algorithm on the simple undirected graph \( G \) shown in Figure 2.43.

Figure 2.43 A Simple Undirected Graph \( G \)

The undirected graph \( G \) can be represented by the following links data table, mycas.LinkSetIn:

```plaintext
data mycas.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 tables. The Louvain algorithm is used by default because no value is specified for the ALGORITHM= option.

```sas
proc network
  links = mycas.LinkSetIn
  outNodes = mycas.NodeSetOut;
  community
    resolutionList = 1.0 0.5
    outLevel = mycas.CommLevelOut
    outCommunity = mycas.CommOut
    outOverlap = mycas.CommOverlapOut
    outCommLinks = mycas.CommLinksOut;
  run;
```

The output data table mycas.NodeSetOut contains the community identifier of each node, as shown in Figure 2.44.

**Figure 2.44** Community Detection on an Undirected 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>1</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>G</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>H</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>I</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

The output data table mycas.CommLevelOut contains summary information at each resolution level, as shown in Figure 2.45.

**Figure 2.45** Community Detection on an Undirected 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 output data table mycas.CommOut contains the number of nodes in each community, as shown in Figure 2.46.

**Figure 2.46** Community Detection on an Undirected 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>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Chapter 2: The NETWORK Procedure

The output data table mycas.CommOverlapOut contains community overlap information, as shown in Figure 2.47.

**Figure 2.47** Community Detection on an Undirected Simple Graph: Community Overlap

<table>
<thead>
<tr>
<th>node</th>
<th>community</th>
<th>intensity</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>1</td>
<td>0.66667</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>0.33333</td>
</tr>
<tr>
<td>B</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>F</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>G</td>
<td>1</td>
<td>0.33333</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
<td>0.66667</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>D</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>E</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>H</td>
<td>2</td>
<td>1.00000</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
<td>1.00000</td>
</tr>
</tbody>
</table>

The output data table mycas.CommLinksOut describes how the communities are connected, as shown in Figure 2.48.

**Figure 2.48** Community Detection on an Undirected 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>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>1</td>
<td>2</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 is 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 NETWORK, you can invoke connected components by using the CONNECTEDCOMPONENTS statement. The options for this statement are described in the section “CONNECTEDCOMPONENTS Statement” on page 36.

There are three algorithms for finding connected components in an undirected graph: a depth-first search algorithm (ALGORITHM=DFS), a union-find algorithm (ALGORITHM=UNIONFIND), and a distributed parallel union-find algorithm (ALGORITHM=PARALLEL). For a graph $G = (N, A)$, each algorithm runs in time $O(|N| + |A|)$ and can usually scale to very large graphs. The default is the sequential union-find algorithm (ALGORITHM=UNIONFIND). For directed graphs, only the depth-first search algorithm (ALGORITHM=DFS) is available.

The results of the connected components algorithm are written to the output nodes data table that you specify in the OUTNODES= option in the PROC NETWORK statement and the output links data table that you specify in the OUTLINKS= option in the PROC NETWORK statement. For each node in the nodes data table
(or link in the links data table), the variable concomp identifies its component. The component identifiers are numbered sequentially, starting from the value of the INDEXOFFSET= option in the PROC NETWORK statement.

**Connected Components of a Simple Undirected Graph**

This section illustrates the use of the connected components algorithm on the simple undirected graph $G$ shown in Figure 2.49.

![A Simple Undirected Graph $G$](image)

The undirected graph $G$ can be represented by the following links data table, mycas.LinkSetIn:

```plaintext
data mycas.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 find the connected components and output the results in the data table mycas.NodeSetOut:

```plaintext
proc network
   links = mycas.LinkSetIn
   outNodes = mycas.NodeSetOut;
   connectedComponents;
run;
```

The output data table mycas.NodeSetOut contains the connected components of the input graph, as shown in Figure 2.50.
Notice that you define the graph by using only the links data table. As seen in Figure 2.49, this graph also contains a singleton node labeled J, which has no associated links. By definition, this node defines its own component. But because you define the input graph by using only the links data table, it does not show up in the results data table. To define a graph by using nodes that have no associated links, you should also define the input nodes data table. In this case, define the nodes data table `mycas.NodeSetIn` as follows:

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

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

```sas
proc network
  nodes = mycas.NodeSetIn
  links = mycas.LinkSetIn
  outNodes = mycas.NodeSetOut;
  connectedComponents;
run;
```

The resulting data table, `mycas.NodeSetOut`, includes the singleton node J as its own component, as shown in Figure 2.51.


**Figure 2.51** Connected Components of a Simple Undirected Graph

<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>1</td>
</tr>
<tr>
<td>H</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>2</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
</tr>
<tr>
<td>D</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 $G$ shown in Figure 2.52.

**Figure 2.52** A Simple Directed Graph $G$

The directed graph $G$ can be represented by the following links data table, mycas.LinkSetIn:
data mycas.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 find the connected components and output the results in the data table mycas.NodeSetOut:

proc network
direction = directed
links = mycas.LinkSetIn
outNodes = mycas.NodeSetOut;
connectedComponents;
run;

The output data table mycas.NodeSetOut, shown in Figure 2.53, now contains the connected components of the input graph.

**Figure 2.53  Connected Components of a Simple Directed Graph**

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

The connected components are represented graphically in Figure 2.54.
Core Decomposition

An alternative to community detection for finding cohesive subgroups is a method of 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 provide a coarse approximation of cohesive structure at a very low computational cost. Let \( G = (N, A) \) define a graph that contains 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 \( G_{S_k} \) is contained in \( G_{S_{k+1}} \).

In PROC NETWORK, you can invoke core decomposition by using the CORE statement. The options for this statement are described in the section “CORE Statement” on page 36.

The results of the core decomposition algorithm are given in the output nodes data table that you specify in the OUTNODES= option in the PROC NETWORK statement. For each node in the nodes data table, the variable core_out identifies its core number, the highest-order core that contains this node.

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

Core Decomposition of a Simple Undirected Graph

This section illustrates the use of the core decomposition algorithm on the simple undirected graph \( G \) shown in Figure 2.55.
The undirected graph $G$ can be represented using the following nodes data table, mycas.NodeSetIn, and links data table, mycas.LinkSetIn:

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

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

**Figure 2.55** Simple Undirected Graph

![Simple Undirected Graph](image-url)
Core Decomposition

The following statements calculate the core decomposition and output the results in the data table mycas.NodeSetOut:

```sas
proc network
  nodes = mycas.NodeSetIn
  links = mycas.LinkSetIn
  outNodes = mycas.NodeSetOut;
  core;
run;
```

The nodes output data table mycas.NodeSetOut contains the core number (variable core_out) for each node, as shown in Figure 2.56.

**Figure 2.56** 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>v11</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>v10</td>
<td>2</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>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 2.57 shows the graph layered by its core number.
**Figure 2.57** Core Decomposition
Cycle Enumeration

A path in a graph is a sequence of links such that the to node of each link is the from node of the next link. An elementary cycle is a path in which the starting node and the ending node are the same and no node appears more than twice in the sequence.

In PROC NETWORK, you can find (or just count) the elementary cycles of an input graph by specifying the CYCLE statement. The options for this statement are described in the section “CYCLE Statement” on page 37. To find the cycles and report them in an output data table, use the OUT= option. To simply count the cycles, omit 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 enumeration algorithm are written to the output data table that you specify in the OUT= option in the CYCLE statement. Each node of each cycle is listed in the OUT= data table 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 identifiers are numbered sequentially, starting from the value of the INDEXOFFSET= option in the PROC NETWORK statement.

The default algorithm that PROC NETWORK uses to compute cycles when the value of the MAXLENGTH= option is greater than 20 is a variant of the algorithm in Johnson (1975) (ALGORITHM=BACKTRACK). This algorithm runs in time $O(|V|(|E|+|A|)(c+1))$, 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. The default when the value of the MAXLENGTH= option is less than or equal to 20 is described in Liu and Wang (2006) (ALGORITHM=BUILD). This algorithm is usually much faster than the backtracking algorithm when the length of the cycles is sufficiently restricted.

If MAXCYCLES=ALL and there are many cycles, the OUT= data table can become very large. It might be beneficial to check the number of cycles before you try to create the OUT= data table. By default (MAXCYCLES=1), 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 37.

Cycle Enumeration of a Simple Directed Graph

This section provides a simple example of using the cycle enumeration algorithm on the simple directed graph $G$ shown in Figure 2.58. For a more detailed example involving both cycle enumeration and transitive closure, see “Example 2.10: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System” on page 185.
The directed graph $G$ can be represented by the following links data table, `mycas.LinkSetIn`:

```plaintext
data mycas.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 count the number of cycles in the graph (without storing them):

```plaintext
proc network
   direction = directed
   links = mycas.LinkSetIn;
   cycle
      maxCycles = all;
run;
%put &_NETWORK_;
```

The result is written to the log of the NETWORK procedure, as shown in Figure 2.59.

**Figure 2.59** PROC NETWORK Log: Count the Number of Cycles in a Simple Directed Graph

`NOTE: ` Running NETWORK.

`NOTE: ` The number of nodes in the input graph is 6.

`NOTE: ` The number of links in the input graph is 10.

`NOTE: ` Processing cycle detection.

`NOTE: ` The algorithm found 7 cycles.

`NOTE: ` Processing cycle detection used 0.00 (cpu: 0.00) seconds.

`NOTE: ` The Cloud Analytic Services server processed the request in 0.071107 seconds.

`STATUS=OK  PROBLEM_TYPE=CYLE SOLUTION_STATUS=OK  NUM_CYCLES=7  CPU_TIME=0.08  REAL_TIME=0.07`

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

```sql
proc network
direction = directed
links = mycas.LinkSetIn;
cycle
  out = mycas.Cycles;
maxCycles = all;
run;
```

The output data table mycas.Cycles now contains all the cycles in the input graph, as shown in Figure 2.62.
Figure 2.62  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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cycle</th>
<th>order</th>
<th>node</th>
</tr>
</thead>
<tbody>
<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>E</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 2.63 through Figure 2.65.

Figure 2.63  Cycles

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

\[ A \rightarrow E \rightarrow C \rightarrow A \]
Node Similarity

*Node similarity* is a measure of how similar the neighborhoods of two nodes are.

In PROC NETWORK, you can calculate node similarity by using the NODESIMILARITY statement. The options for this statement are described in the section “NODESIMILARITY Statement” on page 41.

PROC NETWORK implements three different measures of node similarity: the measure that you specify using the JACCARD= option, based on the Jaccard coefficient; and two measures that you specify using the VECTOR= option, for first- and second-order proximity, specified by PROXIMITYORDER= option, based on cosine similarity between vector representations of nodes that are calculated using the edge sampling...
One simple measure of structural node similarity in a network is the number of neighbors that two nodes have in common. That is, the nodes are similar to the extent that their neighborhoods overlap. Let $\delta(u)$ represent the list of nodes that are connected to node $u$. The similarity between nodes $u$ and $v$ can be defined as

$$S(u, v) = |\delta(u) \cap \delta(V)|$$

Based on this measure, high-degree nodes are more similar, even when a small fraction of their neighbors is shared, compared to low-degree nodes. This bias can be corrected by dividing by the union of two neighborhoods. The measure is called the Jaccard similarity index, and for the neighborhoods of two nodes in a graph, can be expressed as

$$S_J(u, v) = \frac{|\delta(u) \cap \delta(V)|}{|\delta(u) \cup \delta(V)|}$$

By convention, when the union of the two neighborhoods is empty, the Jaccard similarity index is 0.

Another way to compare nodes in the graph is to first embed them in a $d$-dimensional vector space, in such a way that the network structure is preserved. The method that is used here was introduced in Tang et al. (2015), which is designed to produce node embeddings that capture the first- or second-order proximity between nodes as proximity between corresponding vector-space embeddings. The first-order proximity in a graph can be represented by the weight of the link, $w_{uv}$. The second-order proximity between a pair of nodes $(u, v)$ in a graph is the similarity between their neighborhood structures. If the first-order proximity of $u$ to all other nodes is represented as a vector $p_u = (w_{u1}, \ldots, w_{u|V|})$, then the second-order proximity between $u$ and $v$ is determined by the similarity between $p_u$ and $p_v$. The second-order proximity assumes that the nodes that share many connections to other nodes are similar to each other. In the method of Tang et al. (2015), a $d$-dimensional vector $\tilde{e}_u$ is calculated for each node $u$ by minimizing a corresponding objective function. In case of the first-order proximity, the objective function is

$$O_1 = -\sum_{(u,v) \in E} w_{uv} \log p_1(uv)$$

where $p_1$ is the joint probability of an edge between nodes $u$ and $v$, which is modeled as

$$p_1(uv) = \frac{1}{1 + \exp(-\tilde{e}_u^T \cdot \tilde{e}_v)}$$

The embeddings that capture second-order proximity are calculated by minimization of the objective function,

$$O_2 = -\sum_{(uv) \in E} w_{uv} \log p_2(v|u)$$

where

$$p_2(v|u) = \frac{\exp(\tilde{e}_v^T \cdot \tilde{e}_u)}{\sum_{k=1}^{|V|} \exp(\tilde{e}_k^T \cdot \tilde{e}_u)}$$

The optimization is performed by using the asynchronous stochastic gradient descent algorithm with random sampling of edges from the input edge set, where the number of samples is specified in the NSAMPLES= option. In addition, Tang et al. (2015) adopt the approach of negative sampling, which samples multiple negative
edges that are drawn from the uniform noise distribution. You can use the NEGATIVESAMPLEFACTOR= option to specify the number of negative edges for each input edge sample.

After the embeddings for each node in the graph are calculated, the similarity between a pair of nodes \( u \) and \( v \) is calculated as a cosine function between their embeddings \( \tilde{e}_u \) and \( \tilde{e}_v \) and then mapped to the interval \( [0, 1] \):

\[
S_V(u, v) = \left( \frac{\tilde{e}_u \cdot \tilde{e}_v}{\|\tilde{e}_u\| \|\tilde{e}_v\| + 1} \right) / 2
\]

By default, PROC NETWORK finds the node similarity for all pairs of nodes in the input graph. That is, it finds the node similarity for each possible combination of source nodes and sink nodes. Alternatively, you can use the SOURCE= option in the NODESIMILARITY statement option to fix a particular source node and find all node similarity scores from the fixed source node to all possible sink nodes. Conversely, by using the SINK= option in the NODESIMILARITY statement, you can fix a sink node and find all node similarity scores from all possible source nodes to the fixed sink node. By using both options together, you can calculate all similarity scores for a specific source-sink pair. In addition, you can use the NODESSUBSET= option in the PROC NETWORK statement to define a list of source-sink pairs to process, as described in the section “Nodes Subset Input Data” on page 54. The following section provides an example of finding the node similarity for all source-sink pairs. You can find additional examples of how to define the specific source-sink pairs of interest in the section “Shortest Path” on page 129.

**Output Data Tables**

The node similarity algorithm populates up to two output data tables. The output data table that you specify in the OUTSIMILARITY= option in the NODESIMILARITY statement contains the node similarity scores for each source-sink pair. The output data table that you specify in the OUTNODES= option in the PROC NETWORK statement option contains the vector representations of the nodes when you use the VECTOR= option.

**OUTSIMILARITY= Data Table**

The OUTSIMILARITY= data table contains the similarity scores for each node pair. For large graphs and a large requested number of source-sink pairs, this output data table can be extremely large. Generating the output can sometimes take longer than computing the similarity. This output data table is a distributed table when you are running on multiple machines. The only restriction is the total available cache disk space that is enabled by your configuration, as described in *SAS Cloud Analytic Services: Language Reference*.

The OUTSIMILARITY= data table contains the following columns:

- **source**: the source node label of this pair
- **sink**: the sink node label of this pair
- **jaccard**: the Jaccard similarity score, if you specify JACCARD=TRUE
- **vector**: the vector similarity score, if you specify VECTOR=TRUE
**OUTNODES= Data Table**

When VECTOR=TRUE, the OUTNODES= data table contains the vector representations of each node in the input graph. The columns are defined as vec_1–vec_d, where d is specified by the NDIMENSIONS= option in the NODESIMILARITY statement.

**Node Similarity in an Undirected Graph**

This section provides a simple example of calculation of node similarity in the undirected graph G that is shown in Figure 2.66.

*Figure 2.66 A Simple Undirected Graph G*

![Figure 2.66 A Simple Undirected Graph G](image)

The undirected graph G can be represented by the links data table, mycas.LinkSetIn, that is created by the following DATA step:

```plaintext
data mycas.LinkSetIn;
  input from $ to $ weight;
  datalines;
  A B 1
  A D 1
  C B 1
  C D 1
  E B 1
  E D 1
  E F 1
;
```

The following statements find all node pairs that have a similarity score greater than or equal to 0.7:

```plaintext
test network
  links = mycas.LinkSetIn;
  nodeSimilarity
    vector = true
    nDimensions = 10
    proximityOrder = second
    minScore = 0.7
    outSimilarity = mycas.NodeSim;
run;
```

The output data table mycas.NodeSim, as shown in Figure 2.67, contains the pairs of nodes that have either a Jaccard or vector similarity score at or above the specified MINSCORE= option value of 0.7.
Figure 2.67  Node Pairs with Similarity Scores Greater Than 0.7

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>jaccard</th>
<th>vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>1.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>1.00000</td>
<td>0.99332</td>
</tr>
<tr>
<td>A</td>
<td>E</td>
<td>0.66667</td>
<td>0.80379</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>1.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>1.00000</td>
<td>0.99405</td>
</tr>
<tr>
<td>B</td>
<td>F</td>
<td>0.33333</td>
<td>0.83317</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
<td>1.00000</td>
<td>0.99332</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>1.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
<td>0.66667</td>
<td>0.80979</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
<td>1.00000</td>
<td>0.99405</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>1.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td>0.33333</td>
<td>0.77269</td>
</tr>
<tr>
<td>E</td>
<td>A</td>
<td>0.66667</td>
<td>0.80379</td>
</tr>
<tr>
<td>E</td>
<td>C</td>
<td>0.66667</td>
<td>0.80979</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>1.00000</td>
<td>1.00000</td>
</tr>
<tr>
<td>F</td>
<td>B</td>
<td>0.33333</td>
<td>0.83317</td>
</tr>
<tr>
<td>F</td>
<td>D</td>
<td>0.33333</td>
<td>0.77269</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>1.00000</td>
<td>1.00000</td>
</tr>
</tbody>
</table>

This example illustrates the intuition behind node similarity values. The nodes A and C have the same set of nearest neighbors (see Figure 2.66), B and D, which results in the Jaccard similarity score of 1 and a vector similarity score very close to 1; thus the (A,C) pair is present in the output. The pair (B,F) appears in the output because the vector similarity score is greater than 0.7, whereas the Jaccard similarity score is lower than the MINSCORE= option threshold value. In this case, although B and F share only one neighbor out of four total, they do have common neighbors of the nearest neighbors; this results in a greater vector similarity value but not in a greater value of Jaccard similarity, which captures only the similarity of nearest-neighbor structures.

The first-order proximity can be calculated by the following statements:

```sas
proc network
   links = mycas.LinkSetIn
   outNodes = mycas.NodeVecs;
   nodeSimilarity
      jaccard = false
      vector = true
      nDimensions = 10
      proximityOrder = first
      minScore = 0.7
      outSimilarity = mycas.NodeSim;
run;
```

The output data table mycas.NodeSim, as shown in Figure 2.68, contains the pairs of nodes and similarity scores for the first-order proximity. The example also demonstrates the intuition behind similarity that is based on first-order proximity and how it is different from Jaccard-based proximity and the vector for the second-order proximity. Intuitively, the first-order vector similarity value between the nodes that are directly connected is higher than the first-order vector similarity value between the nodes that are separated by a longer path in a graph, whereas the Jaccard similarity and second-order vector similarity values are higher.
between the nodes that share more neighbors. The first-order vector similarity is more closely related to results that are produced by label propagation community detection algorithms, in the sense that the pairs of nodes that have high first-order vector similarity scores between them can be placed in one community.

**Figure 2.68** Node Pairs with Similarity Scores Greater Than 0.7

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>1.0000</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>0.75728</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>0.98006</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
<td>0.75728</td>
</tr>
<tr>
<td>B</td>
<td>B</td>
<td>1.00000</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>0.93072</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>0.77411</td>
</tr>
<tr>
<td>C</td>
<td>B</td>
<td>0.93072</td>
</tr>
<tr>
<td>C</td>
<td>C</td>
<td>1.00000</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>0.76273</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>0.98006</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
<td>0.77411</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
<td>0.76273</td>
</tr>
<tr>
<td>D</td>
<td>D</td>
<td>1.00000</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>1.00000</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>0.99995</td>
</tr>
<tr>
<td>F</td>
<td>E</td>
<td>0.99995</td>
</tr>
<tr>
<td>F</td>
<td>F</td>
<td>1.00000</td>
</tr>
</tbody>
</table>

The output data table mycas.NodeVecs contains node-embedding vectors, shown in **Figure 2.69**.

**Figure 2.69** Node-Embedding Vectors

<table>
<thead>
<tr>
<th>node</th>
<th>vec_1</th>
<th>vec_2</th>
<th>vec_3</th>
<th>vec_4</th>
<th>vec_5</th>
<th>vec_6</th>
<th>vec_7</th>
<th>vec_8</th>
<th>vec_9</th>
<th>vec_10</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>-0.07034</td>
<td>-0.40001</td>
<td>-0.21441</td>
<td>-0.32031</td>
<td>-0.16938</td>
<td>0.15121</td>
<td>-0.09203</td>
<td>0.02494</td>
<td>-0.33596</td>
<td>-0.71621</td>
</tr>
<tr>
<td>B</td>
<td>0.16255</td>
<td>-0.37394</td>
<td>-0.37278</td>
<td>0.05596</td>
<td>0.36977</td>
<td>0.26216</td>
<td>0.28379</td>
<td>0.35574</td>
<td>-0.44346</td>
<td>-0.28714</td>
</tr>
<tr>
<td>C</td>
<td>0.36939</td>
<td>-0.29829</td>
<td>-0.32164</td>
<td>0.05861</td>
<td>0.15852</td>
<td>0.40084</td>
<td>0.26596</td>
<td>0.59520</td>
<td>-0.13276</td>
<td>-0.19814</td>
</tr>
<tr>
<td>D</td>
<td>0.06216</td>
<td>-0.39580</td>
<td>-0.22653</td>
<td>-0.30858</td>
<td>-0.24037</td>
<td>0.25740</td>
<td>-0.06774</td>
<td>0.19653</td>
<td>-0.21348</td>
<td>-0.69291</td>
</tr>
<tr>
<td>E</td>
<td>-0.13405</td>
<td>0.43510</td>
<td>0.32109</td>
<td>0.18303</td>
<td>0.00985</td>
<td>-0.30442</td>
<td>-0.08303</td>
<td>-0.31651</td>
<td>0.32287</td>
<td>0.59332</td>
</tr>
<tr>
<td>F</td>
<td>-0.13322</td>
<td>0.43473</td>
<td>0.32330</td>
<td>0.17843</td>
<td>0.00008</td>
<td>-0.30481</td>
<td>-0.08791</td>
<td>-0.31628</td>
<td>0.32917</td>
<td>0.58981</td>
</tr>
</tbody>
</table>

**Path Enumeration**

A path in a graph is a sequence of links such that the to node of each link is the from node of the next link. An elementary path is a path in which no node appears more than twice in the sequence. A path between two nodes, u and v, in a graph is a path that starts at u and ends at v. The starting node is called the source node, and the ending node is called the sink node.

In PROC NETWORK, you can find the elementary paths of an input graph by specifying the PATH statement. The options for this statement are described in the section “PATH Statement” on page 44.
By default, PROC NETWORK finds paths for all pairs of nodes in the input graph. That is, it finds all paths for each possible combination of source nodes and sink nodes. Alternatively, you can use the SOURCE= option to fix a particular source node and find all 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 all paths from all possible source nodes to the fixed sink node. By using both options together, you can request all paths for a specific source-sink pair. In addition, you can use the NODESSUBSET= option to define a list of source-sink pairs to process, as described in the section “Nodes Subset Input Data” on page 54. The following section provides an example of how to use one of these options. Additional examples that show how to define the source-sink pairs of interest are found in the section “Shortest Path” on page 129.

For weighted graphs, the algorithm uses the weight variable that is defined in the links (nodes) data table to evaluate a path’s total link (node) weight. You can also use the AUXWEIGHT= option in the LINKSVAR statement to define an auxiliary link weight.

**Output Data Tables**

The path enumeration algorithm produces up to two output data tables. The output data table that you specify in the OUTPATHSLINKS= option contains the links of the paths for each source-sink pair. The output data table that you specify in the OUTPATHSNODES= option contains the nodes of the paths for each source-sink pair.

**OUTPATHSLINKS= Data Table**

The OUTPATHSLINKS= data table contains the links present in each path. For large graphs and a large requested number of source-sink pairs, this output data table can be extremely large. Generating the output can sometimes take longer than computing the paths. This output data table is a distributed table when you are running on multiple machines. The only restriction is the total available cache disk space enabled by your configuration, as described in *SAS Cloud Analytic Services: Language Reference*.

The OUTPATHSLINKS= data table contains the following columns:

- **source**: the source node label of this path
- **sink**: the sink node label of this path
- **path**: for this source-sink pair, the path identifier of this path
- **order**: for this source-sink pair, the order of this link in this path
- **from**: the from node label of this link in this path
- **to**: the to node label of this link in this path
- **weight**: the weight of this link in this path
- **column**: the auxiliary weight of this link (if the AUXWEIGHT=column is defined in the LINKSVAR statement)
OUTPATHSNODES= Data Table
The OUTPATHSNODES= data table contains the nodes present in each path. This output data table can also be extremely large. This output data table is a distributed table when you are running on multiple machines.

The OUTPATHSNODES= data table contains the following columns:

- source: the source node label of this path
- sink: the sink node label of this path
- path: for this source-sink pair, the path identifier of this path
- order: for this source-sink pair, the order of this node in this path
- node: the node label of this node in this path
- weight: the weight of this node in this path

Path Enumeration for One Source-Sink Pair
This section provides a simple example of using the path enumeration algorithm on the simple directed graph $G$ shown in Figure 2.70 to find all paths between one source-sink pair by using the SOURCE= and SINK= options.

Figure 2.70 A Simple Directed Graph $G$
The directed graph $G$ can be represented by the following links data table, mycas.LinkSetIn:

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

The following statements find all paths between node $D$ and node $A$ whose path link weight is less than or equal to 10:

```plaintext
proc network
  direction = directed
  links = mycas.LinkSetIn;
  path
    source = D
    sink = A
    maxLinkWeight = 10
    outPathsLinks = mycas.PathLinks
    outPathsNodes = mycas.PathNodes;
run;
```

The output data table mycas.PathLinks contains the links of the three paths from $D$ to $A$ whose path link weight is less than or equal to 10, as shown in Figure 2.71.

**Figure 2.71** Links for All (Short) Paths in a Simple Directed Graph

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>path</th>
<th>order</th>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>D</td>
<td>A</td>
<td>1</td>
<td>1</td>
<td>D</td>
<td>E</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>1</td>
<td>2</td>
<td>E</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>2</td>
<td>1</td>
<td>D</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>2</td>
<td>2</td>
<td>F</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>2</td>
<td>3</td>
<td>E</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>3</td>
<td>1</td>
<td>D</td>
<td>F</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>3</td>
<td>2</td>
<td>F</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>3</td>
<td>3</td>
<td>E</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>3</td>
<td>4</td>
<td>B</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>A</td>
<td>3</td>
<td>5</td>
<td>C</td>
<td>A</td>
<td>6</td>
</tr>
</tbody>
</table>

The output data table mycas.PathNodes contains the nodes of the three paths, as shown in Figure 2.72.
Figure 2.72  Nodes for All (Short) Paths in a Simple Directed Graph

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

The three (short) paths are shown graphically in Figure 2.73.

**Figure 2.73**  Paths

\[ D \rightarrow E \rightarrow A \quad D \rightarrow F \rightarrow E \rightarrow A \quad D \rightarrow F \rightarrow E \rightarrow B \rightarrow C \rightarrow A \]

**Pattern Matching**

Given two graphs, \( G \) (input) and \( Q \) (query), *subgraph isomorphism* is the problem of finding all subgraphs \( Q' \) of \( G \) that are isomorphic to \( Q \). In other words, each such subgraph \( Q' \) has the same topology as graph \( Q \). *Pattern matching* addresses the analogous problem in the presence of node and link attributes. It is the
problem of finding all subgraphs $Q'$ of $G$ isomorphic to graph $Q$ such that all node and link attributes defined in $Q$ are preserved in $Q'$ under the isomorphism map.

In PROC NETWORK, you can find pattern matches by using the PATTERNMATCH statement. The options for this statement are described in the section “PATTERNMATCH Statement” on page 46. The search algorithm is a multithreaded implementation that is based conceptually on the algorithms proposed in Saltz (2013).

The query graph is specified using the LINKSQUERY= or NODESQUERY= option (or both) in the PROC NETWORK statement. The specification of the query graph and its attributes works the same way that it does for the input graph, which is described in more detail in the section “Graph Input Data” on page 49. One exception is the concept of a wildcard when an attribute’s value is missing. A missing value in the query graph indicates that the attribute value in the matching subgraph can be any value. An example of this is shown in the section “Pattern Matching of a Simple Undirected Graph” on page 118.

Output Data Tables

Depending on the options that you specify, the pattern matching algorithm produces output data tables as described in the following sections.

**OUTMATCHNODES= Data Table**

The OUTMATCHNODES= data table describes the mappings from nodes in the query graph to nodes in the input graph for each pattern match. This data tables contains the following columns:

- match: match identifier
- nodeQ: node label for each node in the query graph
- node: node label for each node in the input graph

**OUTMATCHLINKS= Data Table**

The OUTMATCHLINKS= data table describes the subgraph in the input graph for each pattern match. This data table contains the following columns:

- match: match identifier
- from: the from node label
- to: the to node label

**Pattern Matching of a Simple Directed Graph**

This section illustrates the use of the pattern matching algorithm on the simple directed graph $G$ shown in Figure 2.74.
Figure 2.74 Simple Directed Graph $G$

The directed graph $G$ can be represented using the links data table, mycas.Links, and nodes data table, mycas.Nodes, that are created by the following DATA steps:

```plaintext
data mycas.Links;
  input from $ to $ @@;
datalines;
   A B A I A H
   B D B E B C
   C F C G C A
   F G F A G B
   E C I H D G
   F C C D D E
;
data mycas.Nodes;
  input node $ @@;
  label=node;
datalines;
   A B C D E F G H I
;
```

Each node has a node attribute label that is defined as the name of the node.

In this example, you want to find all subgraphs that are directed cycles of length 3 that pass through node C. The query graph $Q$ that defines the pattern to search for is shown in Figure 2.75.

Figure 2.75 Query Graph $Q$
The query graph $Q$ can be represented using the links data table, mycas.LinksQuery, and nodes data table, mycas.NodesQuery, that are created by the following DATA steps:

```plaintext
data mycas.LinksQuery;
   input from to;
   datalines;
   1 2
   2 3
   3 1
;
data mycas.NodesQuery;
   input node label $;
   datalines;
   2 C
;
```

You can use the following statements to find all subgraphs that have the specified pattern:

```plaintext
proc network
   direction = directed
   links = mycas.Links
   nodes = mycas.Nodes
   linksQuery = mycas.LinksQuery
   nodesQuery = mycas.NodesQuery;
   nodesVar
      vars = (label);
   nodesQueryVar
      vars = (label);
   patternMatch
      outMatchNodes = mycas.OutMatchNodes
      outMatchLinks = mycas.OutMatchLinks;
run;
```

The output data table mycas.OutMatchNodes now contains the mapping from nodes in the query graph to nodes in the input graph for each pattern match, as shown in Figure 2.76.

**Figure 2.76** Node Mappings for Pattern Matches

<table>
<thead>
<tr>
<th>match</th>
<th>nodeQ</th>
<th>node</th>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>D</td>
<td>D</td>
</tr>
</tbody>
</table>

The output data table mycas.OutMatchLinks now contains the subgraphs for each pattern match, as shown in Figure 2.77.
Pattern Matching of a Simple Undirected Graph

This section illustrates the use of the pattern matching algorithm on the simple undirected graph $G$ shown in Figure 2.79.
The undirected graph $G$ has one link attribute (weight) and one node attribute (color). The graph can be represented using the links data table, mycas.Links, and nodes data table, mycas.Nodes, that are created by the following DATA steps:

```plaintext
data mycas.Links;
   input from $ to $ weight @@;
   datalines;
   A B 5 A C 5 A D 5 A E 5 B C 5
   B D 5 B E 5 C D 5 C E 5 D E 5
   F G 3 F H 3 G H 3 E F 2 E J 2
   J I 4 J K 4 J L 4 I K 4 I L 4
   K L 4 I O 1 K N 1 L M 2 M N 4
   M O 4 M P 4 N P 4 N O 4 P O 4
;
data mycas.Nodes;
   input node $ color $ @@;
   datalines;
   A red B blue C green D red E green
   F blue G red H green I purple J blue
   K green L red M blue N yellow O purple
   P green
;
```

Each node has a node attribute color, and each link has a link attribute weight.
In this example, you want to find pairs of triangles (cliques of size 3) that are connected by a link whose weight is 1. In addition, you want the outer nodes of the triangle to have the colors blue and green. The query graph \( Q \) that defines the pattern to search for is shown in Figure 2.80.

**Figure 2.80** Query Graph \( Q \)

![Query Graph](image)

The query graph \( Q \) can be represented using the links data table, `mycas.LinksQuery`, and nodes data table, `mycas.NodesQuery`, that are created by the following DATA steps:

```latex
\begin{verbatim}
data mycas.LinksQuery;
   input from to weight;
datalines;
1 2 .
1 3 .
2 3 .
3 4 1
4 5 .
5 6 .
4 6 .
;
data mycas.NodesQuery;
   input node color $;
datalines;
1 green
2 blue
5 blue
6 green
;
\end{verbatim}
```

The missing link weight values in the query graph are treated as wildcards. They indicate that the weight of that particular link in the pattern can be any value. The missing observations in the node data table for the query graph are also treated as wildcards. This indicates that the node attributes of the missing nodes can take any value.

You can use the following statements to find all subgraphs that have the specified pattern:

```latex
\begin{verbatim}
proc network
   links = mycas.Links
   nodes = mycas.Nodes
   nodesQuery = mycas.NodesQuery
   linksQuery = mycas.LinksQuery;
   nodesVar
      vars = (color);
\end{verbatim}
```
nodesQueryVar
vars = (color);

linksQueryVar
vars = (weight);

patternMatch
outMatchLinks = mycas.OutMatchLinks
outMatchNodes = mycas.OutMatchNodes;
run;

The output data table mycas.OutMatchNodes now contains the mapping from nodes in the query graph to nodes in the input graph for each pattern match, as shown in Figure 2.81.

**Figure 2.81** Node Mappings for Pattern Matches

<table>
<thead>
<tr>
<th>match</th>
<th>nodeQ</th>
<th>node</th>
<th>color</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>P</td>
<td>green</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>M</td>
<td>blue</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>O</td>
<td>purple</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>I</td>
<td>purple</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>J</td>
<td>blue</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
<td>K</td>
<td>green</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>K</td>
<td>green</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>J</td>
<td>blue</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>I</td>
<td>purple</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>O</td>
<td>purple</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>M</td>
<td>blue</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>P</td>
<td>green</td>
</tr>
</tbody>
</table>

The output data table mycas.OutMatchLinks now contains the subgraphs for each pattern match, as shown in Figure 2.82.

**Figure 2.82** Subgraphs for Pattern Matches

<table>
<thead>
<tr>
<th>match</th>
<th>from</th>
<th>to</th>
<th>weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I</td>
<td>K</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>I</td>
<td>O</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>J</td>
<td>I</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>J</td>
<td>K</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>M</td>
<td>O</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>M</td>
<td>P</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>P</td>
<td>O</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>I</td>
<td>K</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>I</td>
<td>O</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>J</td>
<td>I</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>J</td>
<td>K</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>M</td>
<td>O</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>M</td>
<td>P</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>P</td>
<td>O</td>
<td>4</td>
</tr>
</tbody>
</table>

Only one set of links (one topology) matches the specified pattern. However, there are two isomorphic mappings. The results are displayed graphically in Figure 2.83.
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. In the context of social networks, reach networks are often referred to as ego networks, because they focus on the neighbors of one particular individual (or more than one).

In PROC NETWORK, reach networks can be found by using the REACH statement. The options for this statement are described in the section “REACH Statement” on page 46.

In most cases, the set of source nodes from which to calculate reach are defined in a nodes subset data table, as described in the section “Nodes Subset Input Data” on page 54. The nodes subset data table can be used to define several sets of sources nodes. Each source node set is used to find the reach networks. The reach network identifier is given in the reach column of the nodes subset data table. When you specify the EACHSOURCE option, every node in the original graph’s node set $N$ is used to find a reach network from each node separately. The mapping between node and reach identifier is created in the data table specified in the OUTNODES= option in the PROC NETWORK statement.

Output Data Tables

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

OUTREACHNODES= Data Table

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

- reach: reach network identifier (which defines the set of source nodes that was used)
Reach (Ego) Network

- node: node label for each node in each reach network

**OUTREACHLINKS= Data Table**
The OUTREACHLINKS= data table 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 computationally expensive compared to calculating only the nodes or counts in the reach networks. This data table contains the following columns:

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

**OUTCOUNTS= Data Table**
The OUTCOUNTS= data table describes the number of nodes in each reach network for each set of source nodes. This data table contains the following columns:

- reach: reach network identifier (which defines the set of source nodes that was used)
- node: node label for each node in the source node sets
- 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 specify the DIGRAPH option, then the OUTCOUNTS= data table 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 either 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 OUTCOUNTS= data table 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
Chapter 2: The NETWORK Procedure

- `count_in_or_out_wt`: the sum of the weights of the nodes reachable using either 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

Reach Network of a Simple Directed Graph

This section illustrates the use of the reach networks algorithm on the simple directed graph $G$ shown in Figure 2.84.

![Simple Directed Graph $G$](image)

The directed graph $G$ can be represented using the following links data table, `mycas.LinkSetIn`:

```plaintext
data mycas.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 nodes subset data tables as follows:

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

```plaintext
data mycas.NodeSubSetIn2;
    input node $ reach;
datalines;
B 1
;
```

For the first set of source nodes, you can use the following statements to find the reach network that is restricted by a hop limit of 1:
proc network
direction  = directed
links      = mycas.LinkSetIn
nodesSubset = mycas.NodeSubSetIn1;
reach
  outReachNodes = mycas.ReachNodes1
  outReachLinks = mycas.ReachLinks1
  outCounts    = mycas.ReachCounts1
  maxReach     = 1;
run;

The output data tables mycas.ReachNodes1, mycas.ReachLinks1, and mycas.ReachCounts1 now contain the nodes, links, and counts of the reach network, respectively, that come from \( S_1 \). They are shown in Figure 2.85.

**Figure 2.85** Reach Network for \( S_1 = \{ A, G \} \) with Hop Limit of 1

### ReachNodes1

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

### ReachLinks1

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

### ReachCounts1

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

The results are displayed graphically in Figure 2.86.
For the second set of source nodes, you can use the following statements to find the reach network that is restricted by a hop limit of 2:

```plaintext
proc network
direction = directed
links = mycas.LinkSetIn
nodesSubset = mycas.NodeSubSetIn2;
reach
outReachNodes = mycas.ReachNodes2
outReachLinks = mycas.ReachLinks2
outCounts = mycas.ReachCounts2
maxReach = 2;
run;
```

The output data tables `mycas.ReachNodes2`, `mycas.ReachLinks2`, and `mycas.ReachCounts2` now contain the nodes, links, and counts of the reach network, respectively, that come from $S_2$. They are shown in Figure 2.87.

**Figure 2.87** Reach Network for $S_2 = \{B\}$ with Hop Limit of 2

**ReachNodes2**

<table>
<thead>
<tr>
<th>reach</th>
<th>node</th>
</tr>
</thead>
<tbody>
<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>E</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>G</td>
</tr>
</tbody>
</table>
**Figure 2.87 continued**

**ReachLinks2**

<table>
<thead>
<tr>
<th>reach</th>
<th>from</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>E</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>E</td>
</tr>
<tr>
<td>1</td>
<td>D</td>
<td>E</td>
</tr>
<tr>
<td>1</td>
<td>E</td>
<td>D</td>
</tr>
<tr>
<td>1</td>
<td>E</td>
<td>F</td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td>G</td>
</tr>
</tbody>
</table>

**ReachCounts2**

<table>
<thead>
<tr>
<th>reach</th>
<th>node</th>
<th>count</th>
<th>count_not</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>B</td>
<td>6</td>
<td>3</td>
</tr>
</tbody>
</table>

The results are displayed graphically in Figure 2.88.

**Figure 2.88** 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 by using a nodes subset data table. The MAXREACH= option applies to all the reach networks that are requested. If the nodes subset data table column reach is set to 0 or missing (.), then the node is not processed. If the reach column is set to a value greater than 0, then the node is processed along with other nodes by using the same marker.

Consider again the graph shown in Figure 2.84, now with source node sets $S_1 = \{C\}$ and $S_2 = \{A, H\}$. These source node sets can be defined together as follows:
data mycas.NodeSubSetIn;
  input node $ reach;
datalines;
  A 2
  C 1
  H 2
;  
You can use the following statements to process both one-hop-limit reach networks in one pass:

```sas
proc network
  direction = directed
  links = mycas.LinkSetIn
  nodesSubset = mycas.NodeSubSetIn;
  reach
  outReachNodes = mycas.ReachNodes
  outReachLinks = mycas.ReachLinks
  outCounts = mycas.ReachCounts
  maxReach = 1;
run;
```

The output data tables mycas.ReachNodes, mycas.ReachLinks, and mycas.ReachCounts now contain the nodes, links, and counts of the reach networks, respectively, that come from $S_1$ and $S_2$. They are shown in Figure 2.89.

**Figure 2.89** Reach Networks for $S_1 = \{C\}$ and $S_2 = \{A, H\}$ with Hop Limit of 1

### ReachNodes

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

### ReachLinks

<table>
<thead>
<tr>
<th>reach</th>
<th>from</th>
<th>to</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>D</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>2</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>2</td>
<td>H</td>
<td>G</td>
</tr>
<tr>
<td>2</td>
<td>G</td>
<td>I</td>
</tr>
<tr>
<td>2</td>
<td>H</td>
<td>I</td>
</tr>
<tr>
<td>1</td>
<td>C</td>
<td>E</td>
</tr>
</tbody>
</table>
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 NETWORK, you can find shortest paths by using the SHORTESTPATH statement. The options for this statement are described in the section “SHORTESTPATH Statement” on page 47.

By default, PROC NETWORK finds shortest paths for all pairs of nodes in the input graph. That is, it finds a shortest path for each possible combination of source nodes 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 NODESSUBSET= option to define a list of source-sink pairs to process, as described in the section “Nodes Subset Input Data” on page 54. The following sections show examples of how to use these options.

Which algorithm PROC NETWORK uses to find shortest paths depends on the data. The algorithm and run-time complexity for each link type are shown in Table 2.11.

<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>

You can find details for each algorithm in Ahuja, Magnanti, and Orlin (1993).

For weighted graphs, the algorithm uses the weight variable that is defined in the links data table to evaluate a path’s total weight (cost). You can also use the AUXWEIGHT= option in the LINKSVAR 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 Tables

The shortest path algorithm produces up to two output data tables. The output data table that you specify in the OUTPATHS= option contains the links of a shortest path for each source-sink pair. The output data
table that you specify in the OUTWEIGHTS= option contains the total weight for the shortest path for each source-sink pair.

**OUTPATHS= Data Table**
The OUTPATHS= data table contains the links present in each shortest path. For large graphs and a large requested number of source-sink pairs, this output data table can be extremely large. 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 table of 11 GB. Finding shortest paths for all pairs from all nodes would produce an enormous output data table. This output data table is a distributed table when you are running on multiple machines. The only restriction is the total available cache disk space enabled by your configuration, as described in SAS Cloud Analytic Services: Language Reference. An example of finding the all-pairs shortest path for this road network is shown in “Example 2.13: Shortest Paths of the New York Road Network” on page 192.

The OUTPATHS= data table 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
- **column**: the auxiliary weight of this link (if the AUXWEIGHT=column is defined in the LINKSVAR statement)

**OUTWEIGHTS= Data Table**
The OUTWEIGHTS= data table contains the total weight (and total auxiliary weight) of each shortest path. This data table 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_auxweight**: the total auxiliary weight of the shortest path for this source-sink pair (if you specify the AUXWEIGHT= option in the LINKSVAR statement)

**Shortest Paths for All Pairs**
This example illustrates the use of the shortest path algorithm for all source-sink pairs on the simple undirected graph $G$ shown in Figure 2.90.
The undirected graph $G$ can be represented by the following links data table, mycas.LinkSetIn:

```plaintext
data mycas.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 find shortest paths for all source-sink pairs:

```plaintext
proc network
  links = mycas.LinkSetIn;
  shortestPath
    outWeights = mycas.ShortPathW
    outPaths = mycas.ShortPathP;
run;
```

The output data table mycas.ShortPathP contains the shortest paths, as shown in Figure 2.91.
The output data table `mycas.ShortPathW` contains the path weights of the shortest paths of each source-sink pair, as shown in Figure 2.92.

<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>
</tbody>
</table>

<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>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>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>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>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>
Shortest Paths for a Subset of Source-Sink Pairs

This section illustrates the use of a nodes subset data table, the NODESSUBSET= option, and the shortest path algorithm to find shortest paths for a subset of source-sink pairs. The data table 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 NETWORK then calculates all the source-sink pairs in the crossproduct of these two sets.

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

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

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

```sas
proc network
   nodesSubset = mycas.NodeSubSetIn
   links = mycas.LinkSetIn;
   shortestPath
      outPaths = mycas.ShortPath;
run;
```

The output data table mycas.ShortPath contains the shortest paths, as shown in Figure 2.93.
Figure 2.93 Shortest Paths for a Subset of Source-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>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 find the shortest paths between a subset of source (or sink) nodes and all the other sink (or source) nodes.

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

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

```latex
\begin{verbatim}
data mycas.NodeSubSetIn;
  input node $ source;
datalines;
  B 1
  E 1
;
\end{verbatim}
```

You can use the same PROC NETWORK call that is used in the section “Shortest Paths for a Subset of Source-Sink Pairs” on page 133 to find all the shortest paths from nodes B and E. The output data table mycas.ShortPath contains the shortest paths, as shown in Figure 2.94.
Conversely, the following DATA step designates nodes B and E as sink nodes:

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

You can use the same PROC NETWORK call again to find all the shortest paths to nodes B and E. The output data table mycas.ShortPath contains the shortest paths, as shown in Figure 2.95.
Figure 2.95  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>C</td>
<td>A</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>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 find the shortest paths between one source-sink pair by using the SOURCE= and SINK= options.

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

```plaintext
proc network
  links = mycas.LinkSetIn;
  shortestPath
    source = C
    sink = F
    outPaths = mycas.ShortPath;
run;
```

The output data table mycas.ShortPath contains this shortest path, as shown in Figure 2.96.

Figure 2.96  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 2.97.
Shortest Paths with Auxiliary Weight Calculation

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

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

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

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

```plaintext
proc network
  links    = mycas.LinkSetIn;
  linksVar = count;
  shortestPath
    auxWeight  = count;
    outWeights = mycas.ShortPathW;
run;
```

The output data table `mycas.ShortPathW` contains the total path weight of shortest paths in each source-sink pair, as shown in Figure 2.98. Because the variable `count` in `mycas.LinkSetIn` has a value of 1 for all links, the value in the output data table variable `path_auxweight` contains the number of links in each shortest path.
Figure 2.98  Shortest Paths Including Auxiliary Weights in Calculation

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>path_weight</th>
<th>path_auxweight</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>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>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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>source</th>
<th>sink</th>
<th>path_weight</th>
<th>path_auxweight</th>
</tr>
</thead>
<tbody>
<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>
<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>

The section “Road Network Shortest Path” on page 10 shows an example of using the shortest path algorithm to minimize travel time to and from work based on traffic conditions.

Shortest Paths with Negative Link Weights

This section illustrates the use of the shortest path algorithm on a simple directed graph $G$ with negative link weights, shown in Figure 2.99.
You can represent the directed graph $G$ by using the following links data table, mycas.LinkSetIn:

```plaintext
data mycas.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 find a shortest path between the source node $E$ and the sink node $B$:

```plaintext
proc network
direction   = directed
links       = mycas.LinkSetIn;
shortestPath
  source   = E
  sink     = B
  outPaths = mycas.ShortPathP;
run;
```

The output data table mycas.ShortPathP contains a shortest path from node $E$ to node $B$, as shown in Figure 2.100.
Now, consider the following adjustment to the weight of link $(B, E)$:

```sas
data mycas.LinkSetIn;
  set mycas.LinkSetIn;
  if(from="B" and to="E") then
    weight=1;
run;
```

In this case, there is a negative weight cycle $(E \rightarrow D \rightarrow B \rightarrow E)$. The Bellman-Ford algorithm catches the cycle and produces an error message, as shown in Figure 2.101.

### Figure 2.101  PROC NETWORK Log: Negative Weight Cycle

```
NOTE: ----------------------------------------------------------------------------------------------------------------------------------------
NOTE: Running NETWORK.
NOTE: ----------------------------------------------------------------------------------------------------------------------------------------
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 8.
NOTE: Processing the shortest paths problem using 32 threads on each of 4 machines.
NOTE: Processing the shortest paths problem between 1 source nodes and 1 sink nodes.
ERROR: The graph contains a negative weight cycle.
NOTE: Processing the shortest paths problem used 0.02 (cpu: 0.00) seconds.
ERROR: The action stopped due to errors.
NOTE: The Cloud Analytic Services server processed the request in 0.129125 seconds.
NOTE: The SAS System stopped processing this step because of errors.
STATUS=ERROR  PROBLEM_TYPE=SHORTESTPATH  CPU_TIME=0.11  REAL_TIME=0.13
```

### Summary Statistics

In PROC NETWORK, 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 48.

### Output Data Tables

The summary statistics that PROC NETWORK produces are divided 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 tables that you specify in the OUTNODES= and OUTLINKS= option in the PROC NETWORK statement. The former statistics are contained in the data table 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 Table**

By default, the summary output data table 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 ($\frac{|A|}{\binom{|N|}{2}}$)
- **self_links_ignored**: the number of self-links that are ignored
- **dup_links_ignored**: the number of duplicate links that are 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 $\delta^{\text{in}}(u) > 0$
- **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 CONNECTEDCOMPONENTS and BICONNECTEDCOMPONENTS options. For more information about connected components and biconnected components, see the sections “Connected Components” on page 92 and “Biconnected Components and Articulation Points” on page 62, respectively. If you use the CONNECTEDCOMPONENTS or BICONNECTEDCOMPONENTS option, the following columns might also appear in the summary output data table 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 SHORTESTPATH= option. The \textit{diameter} of a graph is the longest possible shortest path distance of all source-sink pairs that the graph can contain. For more information about shortest paths, see the section “Shortest Path” on page 129. If you use the SHORTESTPATH= option, the following columns also appear in the summary output data table:

• diameter_wt: the longest weighted shortest path in the graph

• diameter_unwt: the longest unweighted shortest path in the graph

• avg_shortpath_wt: the average weighted shortest path in the graph

• avg_shortpath_unwt: the average unweighted shortest path 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 DIAMETERAPPROX= 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 \( O\left(\sqrt{|N|}\right) \). If you use the DIAMETERAPPROX= option, the following columns also appear in the summary output data table:

• diameter_approx_wt: the approximate longest weighted shortest path in the graph

• diameter_approx_unwt: the approximate longest unweighted shortest path in the graph

\textbf{OUTNODES= Data Table}

In addition, you can produce summary statistics about the nodes of the graph. By default, the following columns are appended to the data table that you specify in the OUTNODES= option in the PROC NETWORK statement:

• sum_in_and_out_wt: the 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
You can produce statistics about the connectedness of the graph by using the CONNECTEDCOMPONENTS and BICONNECTEDCOMPONENTS options. If you use these options, the following column also appears in the nodes output data table for undirected graphs:

- **isolated_star**: the identifier, if the node is in an isolated star; otherwise, missing (.)

The following columns also appear 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 SHORTESTPATH= option. The eccentricity of a node \( u \) is the longest of all possible shortest path distances between \( u \) and any other node. If you use the SHORTESTPATH= option, the following columns also appear in the nodes output data table for undirected graphs:

- **eccentr_out_wt**: the longest weighted shortest path distance from the node
- **eccentr_out_unwt**: the longest unweighted shortest path distance from the node

The following columns also appear for directed graphs:

- **eccentr_in_wt**: the longest weighted shortest path distance to the node
- **eccentr_in_unwt**: the longest unweighted shortest path distance to the node

**OUTLINKS= Data Table**

In addition, you can produce summary statistics about the connectedness of the links of the graph. If you use the CONNECTEDCOMPONENTS or BICONNECTEDCOMPONENTS option, the following columns are appended to the data table that you specify in the OUTLINKS= option in the PROC NETWORK 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 \( G \) shown in Figure 2.102.

**Figure 2.102  A Simple Directed Graph \( G \)**

You can represent the directed graph \( G \) by using the following nodes data table, `mycas.NodeSetIn`, and links data table, `mycas.LinkSetIn`:

```sas
data mycas.NodeSetIn;
  input node $ @@;
datalines;
A B C D E F G H I J K L M N O P
;

data mycas.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 5
;
```

The following statements calculate the default summary statistics and output the results in the data table `mycas.Summary`:

```sas
proc network
  direction = directed
  nodes = mycas.NodeSetIn
  links = mycas.LinkSetIn;
  summary
    out = mycas.Summary;
run;
```
The output data table mycas.Summary contains the default summary statistics of the input graph, as shown in Figure 2.103.

**Figure 2.103**  Graph Summary 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>

The following statements calculate the default summary statistics and produce information about the connectedness of the graph. They output the results in the data table mycas.Summary.

``` Sas
proc network
direction  = directed
nodes       = mycas.NodeSetIn
links       = mycas.LinkSetIn;
summary
  connectedComponents
  out         = mycas.Summary;
run;
```

The output data table mycas.Summary contains the summary statistics of the input graph, as shown in Figure 2.104.

**Figure 2.104**  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>
<th>concomp</th>
<th>isolated_pairs</th>
<th>isolated_stars_out</th>
<th>isolated_stars_in</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>
<td>13</td>
<td>1</td>
<td>1</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 $G$ shown in Figure 2.105.
You can represent the undirected graph $G$ by using the following links data table, mycas.LinkSetIn:

```
data mycas.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 produce information about the shortest path distances of the graph. They output the results in the data table mycas.Summary. In addition, node statistics are produced and output in the data table mycas.NodeSetOut.

```
proc network
    links = mycas.LinkSetIn
    outNodes = mycas.NodeSetOut;
    summary
        out = mycas.Summary
        shortestPath = weight;
run;
```

The output data tables mycas.Summary and mycas.NodeSetOut now contain the summary statistics of the input graph, as shown in Figure 2.106.

**Figure 2.106** Graph Summary and Shortest Path Statistics of a Simple Undirected 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>
<th>diameter_wt</th>
<th>avg_shortpath_wt</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>6</td>
<td>1</td>
<td>0.4</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>3.06667</td>
</tr>
</tbody>
</table>
Transitive Closure

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

The transitive closure of a graph can help efficiently answer questions about reachability. Suppose you want to find out 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 $(i, j)$. Transitive closure has many applications, including speeding up the processing of structured query languages, which are often used in databases.

In PROC NETWORK, you can invoke the transitive closure algorithm by using the TRANSITIVECLOSURE statement. The options for this statement are described in the section “TRANSITIVECLOSURE Statement” on page 49.

The links that define the transitive closure of the input graph are written to the output data table that you specify in the OUT= option in the TRANSITIVECLOSURE statement.

The algorithm that PROC NETWORK 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 $G$ shown in Figure 2.107.
The directed graph $G$ can be represented by the following links data table, mycas.LinkSetIn:

```plaintext
data mycas.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 table mycas.TransClosure:

```plaintext
proc network
direction = directed
links = mycas.LinkSetIn;
transitiveClosure
   out = mycas.TransClosure;
run;
```

The output data table mycas.TransClosure contains the transitive closure of $G$, as shown in Figure 2.108.

**Figure 2.108** 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 2.109.
For a more detailed example, see “Example 2.10: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System” on page 185.

Macro Variable _NETWORK_

The NETWORK procedure defines a macro variable named _NETWORK_. This variable contains a character string that indicates the status of PROC NETWORK upon termination and details about the selected algorithm. 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.
- **ERROR** The procedure encountered an error.

**PROBLEM_TYPE**

indicates the selected problem type (algorithm class). The PROBLEM_TYPE term can take one of the following values:

- **BICONNECTEDCOMPONENTS** Biconnected components
- **CENTRALITY** Centrality
- **CLIQUE** Clique enumeration
- **COMMUNITY** Community detection
Chapter 2: The NETWORK Procedure

CONNECTEDCOMPONENTS  Connected components
CORE                  Core decomposition
CYCLE                 Cycle enumeration
NODESIMILARITY        Node similarity
PATH                  Path enumeration
PATTERNMATCH          Pattern matching
REACH                 Reach (ego) networks
SHORTESTPATH          Shortest path
SUMMARY               Graph summary
TRANSITIVECLOSURE     Transitive closure

SOLUTION_STATUS
indicates the solution status of the selected problem type (algorithm class). The SOLUTION_STATUS term can take one of the following values:

OK         The algorithm terminated normally.
ERROR      The algorithm encountered an error.
INTERRUPTED The algorithm was interrupted by the user.
OUTMEM_SOL The algorithm ran out of memory but still found a solution.
OUTMEM_NOSOL The algorithm ran out of memory and either did not find a solution or failed to output the solution due to insufficient memory.
TIMELIMIT  The algorithm reached its execution time limit.
SOLUTION_LIM The algorithm reached its limit on the number of solutions found.

CPU_TIME
indicates the total CPU time (in seconds) that PROC NETWORK used.

REAL_TIME
indicates the elapsed time (in seconds) that PROC NETWORK used.

In addition, each algorithm might report some additional details. The following section provides more information about these details.

Macro Variable _NETWORK_ Details

The BICONNECTEDCOMPONENTS algorithm provides the following additional information:

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

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

The CLIQUE algorithm provides the following additional information:
NUM_CLIQUES
indicates the number of cliques that the algorithm found.

The CONNECTEDCOMPONENTS algorithm provides the following additional information:

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

The CYCLE algorithm provides the following additional information:

NUM_CYCLES
indicates the number of cycles that the algorithm found.

The PATH algorithm provides the following additional information:

NUM_PATHS
indicates the number of paths that the algorithm found.

The PATTERNMATCH algorithm provides the following additional information:

NUM_MATCHES
indicates the number of pattern matches that the algorithm found.

The SHORTESTPATH algorithm provides the following additional information:

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

ODS Table Names
For general information about ODS tables, see SAS Output Delivery System: Procedures Guide. Each ODS table that the NETWORK procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. These names are listed in Table 2.12.

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProblemSummary</td>
<td>Summary of the graph</td>
</tr>
<tr>
<td>SolutionSummary</td>
<td>Summary of the solution status, timing, and results</td>
</tr>
<tr>
<td>OutputCasTables</td>
<td>See the section “OutputCasTables Table” on page 152</td>
</tr>
</tbody>
</table>

The following statements use the example in the section “Shortest Paths for All Pairs” on page 130 and find all-pairs shortest paths for a small undirected graph. By default, this code produces the two ODS output tables listed in Table 2.12.

data mycas.LinkSetIn;
  input from $ to $ weight @@;
datalines;
  A B 3  A C 2  A D 6  A E 4  B D 5
Chapter 2: The NETWORK Procedure

B F 5 C E 1 D E 2 D F 1 E F 4

proc network
  links = mycas.LinkSetIn;
  shortestPath
    outWeights = mycas.ShortPathW;
    outPaths = mycas.ShortPathP;
run;

The problem summary table in Figure 2.110 provides a basic summary of the graph input.

**Figure 2.110** Problem Summary Table

<table>
<thead>
<tr>
<th>Problem Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Nodes</td>
</tr>
<tr>
<td>Number of Links</td>
</tr>
<tr>
<td>Graph Direction</td>
</tr>
</tbody>
</table>

The solution summary table in Figure 2.111 provides a basic solution summary for the algorithm that is processed. The information in this table is similar to the information that is provided in the macro variable `_NETWORK_`, described in the section “Macro Variable `_NETWORK_” on page 149. The timing information in this table (and in the log) represents the time spent running the algorithm, excluding the time spent in input, graph building, and output. In the case of a distributed algorithm, which uses multiple machines, the real time represents the maximum amount of time that an individual machine used to run the algorithm, and the CPU time represents the total amount of time across all active machines in your configured session.

**Figure 2.111** Solution Summary Table

<table>
<thead>
<tr>
<th>Solution Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Problem Type</td>
</tr>
<tr>
<td>Solution Status</td>
</tr>
<tr>
<td>Number of Paths</td>
</tr>
<tr>
<td>CPU Time</td>
</tr>
<tr>
<td>Real Time</td>
</tr>
</tbody>
</table>

**OutputCasTables Table**

The OutputCasTables table is a special table that has information about each CAS table that is created during a CAS action execution. The information for each CAS table consists of the CAS table name, the caslib in which the table resides, and the number of columns and rows in the CAS table. Because this table is not a typical ODS table that contains analytical results, you cannot include it in the `table-spec-list` in the DISPLAYOUT statement.
Example 2.1: Articulation Points in a Terrorist Network

This example considers the terrorist communications network from the attacks on the United States on September 11, 2001, described in Krebs (2002). Figure 2.112 shows this network, which was constructed after the attacks, based on collected intelligence information.

Figure 2.112 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 BICONNECTEDCOMPONENTS statement in this context:

```plaintext
data mycas.LinkSetInTerror911;
  input from & $32. to & $32.;
data lines;
  Abu Zubeida   Djamal Beghal
```

Examples: NETWORK Procedure
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:

```plaintext
proc network
   links   = mycas.LinkSetInTerror911
   outNodes = mycas.NodeSetOut;
   biconnectedComponents;
run;

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

The output data table `mycas.ArtPoints` contains members of the network who are articulation points. By focusing on cutting off these particular members, investigators could have significantly disrupted the terrorists’ ability to communicate when planning the attack.
Example 2.2: 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 person B work together or that person A reports to person B. The graph represents the six main project groups.

- 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 groups, which 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 2.113.
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 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, receive a link weight of 3, and reporting links to Yu receive a weight of 2. Links that represent people working together on a project all receive an 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 nodes data tables:

```sas
data mycas.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
```

Figure 2.113  Project Groups in a Research Department
Example 2.2: Influence Centrality for Project Groups in a Research Department

The following statements calculate influence centrality (in addition to degree centrality):

```plaintext
proc network
  logLevel = moderate
  links = mycas.LinkSetInDept
  nodes = mycas.NodeSetInDept
  outNodes = mycas.NodeSetOut;
```

```
Zhuo    Oliver    1
Zhuo    Gotti     1
Zhuo    Patrick   1
Kabutz  Gotti     1
Leon    Gotti     1
Polark  Yu        2
Polark  Chang     1
Chang   Angel     1
Polark  Angel     1
Weng    Polark    1
Weng    Chang     1
Weng    Angel     1
Christoph    Yu     2
Christoph    Nardo  1
Christoph    Gotti  1
Christoph    Zhuo   1
Nardo     Gotti    1
Nardo     Zhuo     1
Graffe    Yu       2
Graffe    Hund     1
Graffe    Zhuo     1
Zhuo      Hund     1

; data mycas.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):

```plaintext
proc network
  logLevel = moderate
  links = mycas.LinkSetInDept
  nodes = mycas.NodeSetInDept
  outNodes = mycas.NodeSetOut;
```
centrality
degree
    influence = weight;
run;
%put &_NETWORK_;

The progress of the procedure is shown in Output 2.2.1.

**Output 2.2.1** PROC NETWORK Log: Influence Centrality for Project Groups in a Research Department

```
NOTE: -------------------------------------------------------------------------------------------------------------------------------------
NOTE: Running NETWORK.
NOTE: -------------------------------------------------------------------------------------------------------------------------------------
NOTE: Reading the nodes data.
NOTE: Reading the links data.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: Building the input (full) graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 18.
NOTE: The number of links in the input graph is 35.
NOTE: Processing centrality metrics.
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: 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: Processing centrality metrics used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.061285 seconds.
NOTE: The data set MYCAS.NODESETOUT has 18 observations and 5 variables.
STATUS=OK  PROBLEM_TYPE=CENTRALITY  SOLUTION_STATUS=OK  CPU_TIME=0.11  REAL_TIME=0.06
```
The nodes data table mycas.NodeSetOut now contains the weighted influence centrality of the department’s graph, including $C_1$ (the centr_influence1_wt variable) and $C_2$ (the centr_influence2_wt variable). This data table is shown in Output 2.2.2.

**Output 2.2.2** 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>Kabutz</td>
<td>1</td>
<td>4</td>
<td>0.14286</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>Snopp</td>
<td>2</td>
<td>4</td>
<td>0.21429</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>Graffe</td>
<td>1</td>
<td>3</td>
<td>0.14286</td>
<td>0.64286</td>
</tr>
<tr>
<td>Polark</td>
<td>2</td>
<td>4</td>
<td>0.17857</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>Angel</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>Weng</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, because 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 2.3: 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 2.114. This is an example of the well-known kite network, which was popularized by Krackhardt (1990) for better understanding of social networks in the workplace.
Define the links data table as follows:

```plaintext
data mycas.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.

```plaintext
proc network
  links      = mycas.LinkSetInCompNet
  outLinks   = mycas.LinkSetOut
  outNodes   = mycas.NodeSetOutCentr;
  centrality
    degree
    close   = unweight
    between = unweight;
run;

proc network
  links      = mycas.LinkSetInCompNet
  outNodes   = mycas.NodeSetOutBiCC;
  biconnectedComponents;
run;
data mycas.NodeSetOut;
  by node;
run;
%put &_NETWORK_;```
Example 2.3: Betweenness and Closeness Centrality for Computer Network Topology

Output 2.3.1 shows the resulting nodes data table mycas.NodeSetOut sorted by closeness.

Output 2.3.1  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>H</td>
<td>3</td>
<td>0.60000</td>
<td>0.38889</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>0.60000</td>
<td>0.10185</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>B</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 2.3.2 shows the resulting nodes data table (mycas.NodeSetOut) sorted by node betweenness.

Output 2.3.2  Node Closeness and Betweenness Centrality, Sorted by Betweenness

<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>H</td>
<td>3</td>
<td>0.60000</td>
<td>0.38889</td>
<td>1</td>
</tr>
<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>I</td>
<td>2</td>
<td>0.42857</td>
<td>0.22222</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>0.60000</td>
<td>0.10185</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>B</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>J</td>
<td>1</td>
<td>0.31034</td>
<td>0.00000</td>
<td>0</td>
</tr>
</tbody>
</table>

Output 2.3.3 shows the resulting links data table (mycas.LinkSetOut) sorted by link betweenness.
Output 2.3.3 Link Betweenness Centrality, Sorted by Betweenness

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>centr_between_unwt</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>I</td>
<td>0.44444</td>
</tr>
<tr>
<td>C</td>
<td>H</td>
<td>0.29167</td>
</tr>
<tr>
<td>F</td>
<td>H</td>
<td>0.29167</td>
</tr>
<tr>
<td>I</td>
<td>J</td>
<td>0.25000</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>0.12963</td>
</tr>
<tr>
<td>E</td>
<td>F</td>
<td>0.12963</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>0.12500</td>
</tr>
<tr>
<td>F</td>
<td>G</td>
<td>0.12500</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>0.09259</td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td>0.09259</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>0.08333</td>
</tr>
<tr>
<td>D</td>
<td>G</td>
<td>0.08333</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>0.07407</td>
</tr>
<tr>
<td>C</td>
<td>F</td>
<td>0.07407</td>
</tr>
<tr>
<td>D</td>
<td>E</td>
<td>0.05093</td>
</tr>
<tr>
<td>B</td>
<td>D</td>
<td>0.05093</td>
</tr>
<tr>
<td>A</td>
<td>B</td>
<td>0.04167</td>
</tr>
<tr>
<td>E</td>
<td>G</td>
<td>0.04167</td>
</tr>
</tbody>
</table>

The computers that have the highest closeness centrality are C and F, because they have the average shortest paths to all the 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 that has 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 that they are always online.

Example 2.4: Betweenness and Closeness Centrality for Project Groups in a Research Department

This example uses the same data as in “Example 2.2: Influence Centrality for Project Groups in a Research Department” on page 155, which illustrates influence centrality by considering the link weights that represent some measure of reporting magnitude. In Example 2.2, links between managers (or leads) and direct reports have higher link weights than links between nonmanagers. This interpretation makes sense in the context of influence centrality because weight and the metric are directly related. However, for closeness and betweenness centrality, weight and the metric are inversely related.

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. As described in the section “Closeness Centrality” on page 70, by default, PROC NETWORK uses the reciprocal of the link weight to find the shortest paths of the closeness and betweenness centrality metrics.
The following statements calculate weighted (and unweighted) closeness and betweenness centrality.

```
proc network
  logLevel = moderate
  links = mycas.LinkSetInDept
  outLinks = mycas.LinkSetOut
  outNodes = mycas.NodeSetOut;
  centrality
    close = both
    between = both;
run;
%put &_NETWORK_;
```

The progress of the procedure is shown in Output 2.4.1.

**Output 2.4.1** PROC NETWORK Log: Closeness and Node Betweenness Centrality for Project Groups in a Research Department

```
NOTE: ---------------------------------------------------------------------------
NOTE: Running NETWORK.
NOTE: ---------------------------------------------------------------------------
NOTE: Reading the links data.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: Building the input (full) graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 18.
NOTE: The number of links in the input graph is 35.
NOTE: Processing centrality metrics.
NOTE: Processing between/close centrality metrics using 32 threads on each of 4 machines.
           Real
       Algorithm  Nodes  Complete  Time
       centrality  18   100%    0.06
NOTE: Processing between/close centrality metrics used 0.06 seconds.
NOTE: Processing centrality metrics used 0.05 (cpu: 0.02) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.220742 seconds.
NOTE: The data set MYCAS.LINKSETOUT has 35 observations and 5 variables.
NOTE: The data set MYCAS.NODESETOUT has 18 observations and 5 variables.
STATUS=OK  PROBLEM_TYPE=CENTRALITY  SOLUTION_STATUS=OK  CPU_TIME=0.21  REAL_TIME=0.22
```

The nodes data table mycas.NodeSetOut shows the weighted and unweighted closeness and node betweenness centrality, as shown in Output 2.4.2.
### Output 2.4.2  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>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>Christoph</td>
<td>0.68456</td>
<td>0.48571</td>
<td>0.05882</td>
<td>0.11275</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>Nardo</td>
<td>0.51777</td>
<td>0.42500</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
<tr>
<td>Yu</td>
<td>0.87179</td>
<td>0.50000</td>
<td>0.50000</td>
<td>0.41262</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>Chapman</td>
<td>0.88696</td>
<td>0.50000</td>
<td>0.44118</td>
<td>0.23235</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>Patrick</td>
<td>0.50000</td>
<td>0.37778</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>
<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>Kaboutz</td>
<td>0.50746</td>
<td>0.38636</td>
<td>0.00000</td>
<td>0.03885</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>Polark</td>
<td>0.69388</td>
<td>0.38636</td>
<td>0.30882</td>
<td>0.30882</td>
</tr>
<tr>
<td>Weng</td>
<td>0.44156</td>
<td>0.29310</td>
<td>0.00000</td>
<td>0.00000</td>
</tr>
</tbody>
</table>

The links data table mycas.LinkSetOut shows the weighted and unweighted link betweenness centrality, as shown in Output 2.4.3.
Output 2.4.3 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>Chang</td>
<td>Angel</td>
<td>1</td>
<td>0.00735</td>
<td>0.00735</td>
</tr>
<tr>
<td>Polark</td>
<td>Chang</td>
<td>1</td>
<td>0.11029</td>
<td>0.11029</td>
</tr>
<tr>
<td>Weng</td>
<td>Chang</td>
<td>1</td>
<td>0.00735</td>
<td>0.00735</td>
</tr>
<tr>
<td>Polark</td>
<td>Angel</td>
<td>1</td>
<td>0.11029</td>
<td>0.11029</td>
</tr>
<tr>
<td>Weng</td>
<td>Angel</td>
<td>1</td>
<td>0.00735</td>
<td>0.00735</td>
</tr>
<tr>
<td>Christoph</td>
<td>Gotti</td>
<td>1</td>
<td>0.02574</td>
<td>0.09620</td>
</tr>
<tr>
<td>Christoph</td>
<td>Nardo</td>
<td>1</td>
<td>0.04779</td>
<td>0.04412</td>
</tr>
<tr>
<td>Christoph</td>
<td>Yu</td>
<td>2</td>
<td>0.13603</td>
<td>0.15870</td>
</tr>
<tr>
<td>Christoph</td>
<td>Zhuo</td>
<td>1</td>
<td>0.03309</td>
<td>0.05147</td>
</tr>
<tr>
<td>Nardo</td>
<td>Gotti</td>
<td>1</td>
<td>0.05515</td>
<td>0.05147</td>
</tr>
<tr>
<td>Zhuo</td>
<td>Gotti</td>
<td>1</td>
<td>0.05515</td>
<td>0.10184</td>
</tr>
<tr>
<td>Gotti</td>
<td>Chapman</td>
<td>3</td>
<td>0.20221</td>
<td>0.09767</td>
</tr>
<tr>
<td>Gotti</td>
<td>Oliver</td>
<td>1</td>
<td>0.00000</td>
<td>0.03431</td>
</tr>
<tr>
<td>Gotti</td>
<td>Patrick</td>
<td>1</td>
<td>0.05882</td>
<td>0.06066</td>
</tr>
<tr>
<td>Leon</td>
<td>Gotti</td>
<td>1</td>
<td>0.07353</td>
<td>0.12586</td>
</tr>
<tr>
<td>Kabutz</td>
<td>Gotti</td>
<td>1</td>
<td>0.07353</td>
<td>0.12586</td>
</tr>
<tr>
<td>Nardo</td>
<td>Zhuo</td>
<td>1</td>
<td>0.02206</td>
<td>0.02941</td>
</tr>
<tr>
<td>Yu</td>
<td>Chapman</td>
<td>3</td>
<td>0.39706</td>
<td>0.25576</td>
</tr>
<tr>
<td>Graffe</td>
<td>Yu</td>
<td>2</td>
<td>0.18015</td>
<td>0.12402</td>
</tr>
<tr>
<td>Polark</td>
<td>Yu</td>
<td>2</td>
<td>0.41176</td>
<td>0.41176</td>
</tr>
<tr>
<td>Zhuo</td>
<td>Oliver</td>
<td>1</td>
<td>0.02574</td>
<td>0.03885</td>
</tr>
<tr>
<td>Zhuo</td>
<td>Patrick</td>
<td>1</td>
<td>0.02941</td>
<td>0.04412</td>
</tr>
<tr>
<td>Graffe</td>
<td>Zhuo</td>
<td>1</td>
<td>0.03676</td>
<td>0.08578</td>
</tr>
<tr>
<td>Zhuo</td>
<td>Hund</td>
<td>1</td>
<td>0.05515</td>
<td>0.07696</td>
</tr>
<tr>
<td>Oliver</td>
<td>Chapman</td>
<td>3</td>
<td>0.14338</td>
<td>0.07623</td>
</tr>
<tr>
<td>Snopp</td>
<td>Chapman</td>
<td>3</td>
<td>0.26471</td>
<td>0.16005</td>
</tr>
<tr>
<td>Oliver</td>
<td>Patrick</td>
<td>1</td>
<td>0.03676</td>
<td>0.02022</td>
</tr>
<tr>
<td>Graffe</td>
<td>Hund</td>
<td>1</td>
<td>0.06985</td>
<td>0.04804</td>
</tr>
<tr>
<td>Gukrishnan</td>
<td>Leon</td>
<td>1</td>
<td>0.00735</td>
<td>0.03431</td>
</tr>
<tr>
<td>Kabutz</td>
<td>Gukrishnan</td>
<td>1</td>
<td>0.00735</td>
<td>0.03431</td>
</tr>
<tr>
<td>Snopp</td>
<td>Gukrishnan</td>
<td>1</td>
<td>0.11029</td>
<td>0.05637</td>
</tr>
<tr>
<td>Kabutz</td>
<td>Leon</td>
<td>1</td>
<td>0.00735</td>
<td>0.00735</td>
</tr>
<tr>
<td>Snopp</td>
<td>Leon</td>
<td>1</td>
<td>0.03676</td>
<td>0.03517</td>
</tr>
<tr>
<td>Kabutz</td>
<td>Snopp</td>
<td>1</td>
<td>0.03676</td>
<td>0.03517</td>
</tr>
<tr>
<td>Weng</td>
<td>Polark</td>
<td>1</td>
<td>0.11029</td>
<td>0.11029</td>
</tr>
</tbody>
</table>

Note that Chapman (the director) and Yu (a manager who reports to Chapman) both have the highest weighted closeness centrality. However, Yu's weighted betweenness centrality is highest because he serves as a gatekeeper between his three groups (D4a, D4b, and D4c) and the rest of the department.

Example 2.5: 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 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 preceding word senses and the weights are determined by the similarity metric. The resulting graph is shown in Figure 2.115.

**Figure 2.115** Eigenvector Centrality for Word Sense Disambiguation
To identify the correct senses, you run eigenvector centrality on the graph and select the highest-ranking sense for each word:

```sql
data mycas.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 network
   links   = mycas.LinkSetIn
   outNodes = mycas.NodeSetOut;
   centrality
      eigen   = weight;
run;

data mycas.NodeSetOut;
   length word $8 sense $1;
   set mycas.NodeSetOut;
   word = scan(node,1,'_');
   sense = scan(node,2,'_');
run;

data NodeSetOut;
   set mycas.NodeSetOut;
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 2.5.1.
Example 2.6: 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 US university in the 1970s. This is one of the standard publicly available data tables for testing community detection algorithms. It contains 34 nodes and 78 links. The graph is shown in Figure 2.116.

Figure 2.116  Zachary’s Karate Club Graph
The graph can be represented using the following links data table, mycas.LinkSetIn:

```sql
data mycas.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 RESOLUTIONLIST= 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 87.

```sql
proc network
    links = mycas.LinkSetIn
    outNodes = mycas.NodeSetOut;
    community
        resolutionList = 1.0 0.5
        outLevel = mycas.CommLevelOut
        outCommunity = mycas.CommOut
        outOverlap = mycas.CommOverlapOut
        outCommLinks = mycas.CommLinksOut;
run;
```

The output data table mycas.NodeSetOut contains the community identifier of each node, as shown in Output 2.6.1.
Output 2.6.1 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>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>28</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>29</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>32</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>

Different node colors are used to represent different communities in Figure 2.117 and Figure 2.118. As you can see from the figures, four communities at resolution 1.0 are merged into two communities at resolution 0.5.

Figure 2.117 Karate Club Communities (Resolution = 1.0)
Example 2.6: Community Detection on Zachary’s Karate Club Data

The output data table `mycas.CommLevelOut` contains the number of communities and the corresponding modularity values found at each resolution level. It is shown in **Output 2.6.2**.

**Output 2.6.2** 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 output data table `mycas.CommOut` contains the number of nodes in each community, as shown in **Output 2.6.3**.

**Output 2.6.3** 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>1</td>
<td>11</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>2</td>
<td>17</td>
</tr>
</tbody>
</table>

The output data table `mycas.CommOverlapOut` contains the intensity of each node that belongs to multiple communities. It is shown in **Output 2.6.4**. Note that only the communities in the last resolution level (the smallest resolution value) appear as output in this data table. In this example, Node 0 belongs to two communities, with 82.3% of its links connecting to Community 1 and 17.6% of its links connecting to Community 2.
Chapter 2: The NETWORK Procedure

Output 2.6.4  Community Overlap Output

<table>
<thead>
<tr>
<th>node</th>
<th>community</th>
<th>intensity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0.82353</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0.17647</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>0.60000</td>
</tr>
<tr>
<td>9</td>
<td>2</td>
<td>0.40000</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>0.50000</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>0.50000</td>
</tr>
<tr>
<td>14</td>
<td>1</td>
<td>0.20000</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>0.80000</td>
</tr>
<tr>
<td>15</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>19</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>0.33333</td>
</tr>
<tr>
<td>20</td>
<td>2</td>
<td>0.66667</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>27</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>0.75000</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
<td>0.25000</td>
</tr>
<tr>
<td>29</td>
<td>1</td>
<td>0.66667</td>
</tr>
<tr>
<td>29</td>
<td>2</td>
<td>0.33333</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>31</td>
<td>1</td>
<td>0.75000</td>
</tr>
<tr>
<td>31</td>
<td>2</td>
<td>0.25000</td>
</tr>
<tr>
<td>32</td>
<td>1</td>
<td>0.83333</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>0.16667</td>
</tr>
<tr>
<td>33</td>
<td>1</td>
<td>0.91667</td>
</tr>
<tr>
<td>33</td>
<td>2</td>
<td>0.08333</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.11111</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0.88889</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0.12500</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0.87500</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0.40000</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>0.60000</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1.00000</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>1.00000</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1.00000</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>1.00000</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>1.00000</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>1.00000</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>1.00000</td>
</tr>
<tr>
<td>13</td>
<td>2</td>
<td>1.00000</td>
</tr>
<tr>
<td>17</td>
<td>2</td>
<td>1.00000</td>
</tr>
<tr>
<td>18</td>
<td>2</td>
<td>1.00000</td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td>1.00000</td>
</tr>
<tr>
<td>26</td>
<td>1</td>
<td>1.00000</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>1.00000</td>
</tr>
</tbody>
</table>

The output data table mycas.CommLinksOut shows how the communities are interconnected. It is shown in Output 2.6.5. In this example, when the resolution value is 1, the link weight between Communities 1 and 2 is 7, and the link weight between Communities 2 and 3 is 3.

Output 2.6.5  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>1</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>1</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>1.0</td>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>1</td>
<td>2</td>
<td>10</td>
</tr>
</tbody>
</table>

Example 2.7: Recursive Community Detection on Zachary’s Karate Club Data

This example illustrates the use of the RECURSIVE option in PROC NETWORK for community detection on Zachary’s Karate Club data (Zachary 1977). The data table appears in “Example 2.6: Community Detection on Zachary’s Karate Club Data” on page 168. The current example forces each community to contain no more than five nodes and limits the number of links between any pair of nodes within any community to be no greater than 2.
proc network
  links = mycas.LinkSetIn
  outNodes = mycas.NodeSetOut;
community
  resolutionList = 1.0
  recursive (maxCommSize = 5 maxDiameter = 2 relation = AND)
  outCommunity = mycas.CommOut;
run;

The output data table `mycas.NodeSetOut` contains the community identifier of each node, as shown in Output 2.7.1.

<table>
<thead>
<tr>
<th>node</th>
<th>community_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</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>4</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>19</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>7</td>
</tr>
<tr>
<td>21</td>
<td>4</td>
</tr>
<tr>
<td>23</td>
<td>4</td>
</tr>
<tr>
<td>24</td>
<td>9</td>
</tr>
<tr>
<td>27</td>
<td>3</td>
</tr>
<tr>
<td>28</td>
<td>9</td>
</tr>
<tr>
<td>29</td>
<td>10</td>
</tr>
<tr>
<td>30</td>
<td>3</td>
</tr>
<tr>
<td>31</td>
<td>2</td>
</tr>
<tr>
<td>32</td>
<td>10</td>
</tr>
<tr>
<td>33</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>8</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>7</td>
</tr>
<tr>
<td>22</td>
<td>8</td>
</tr>
<tr>
<td>26</td>
<td>9</td>
</tr>
<tr>
<td>25</td>
<td>9</td>
</tr>
</tbody>
</table>

The output data table `mycas.CommOut` contains the number of nodes in each community, as shown in Output 2.7.2.

<table>
<thead>
<tr>
<th>level</th>
<th>resolution</th>
<th>community_nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1 5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2 2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>3 2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>4 7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>5 3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>6 3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>7 3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>8 3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>9 4</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>10 2</td>
</tr>
</tbody>
</table>
The community graph is shown in Figure 2.119, with different node shapes and colors representing different communities.

**Figure 2.119** Karate Club Recursive Communities

As you can see from Output 2.7.2, Community 4, whose nodes are drawn as black ellipses in Figure 2.119, contains seven nodes even though the maximum number of nodes in any community is set to 5. This is because Community 4 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 2.8: Centrality Metrics for a Simple Undirected Graph by Community**

When you are trying to understand the roles of certain entities in a social network, a typical workflow is to first divide the network into communities and then calculate centrality metrics on the induced subgraphs defined by those communities. You can process these induced subgraphs of the original input graph with only one call to PROC NETWORK by using the **BY** statement. This section presents an example of how to use the **COMMUNITY** statement, followed by the **CENTRALITY** statement in conjunction with the **BY** statement.

Consider the graph depicted in Figure 2.120.
The following statements create the data table `mycas.LinkSetIn`:

```\texttt{\small}
\begin{verbatim}
data mycas.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
;
\end{verbatim}
```

First, call the community detection method as follows:

```\texttt{\small}
\begin{verbatim}
proc network
  links = mycas.LinkSetIn
  outNodes = mycas.OutNodesComms
  outLinks = mycas.OutLinksComms;
  community;
run;
\end{verbatim}
```

The resulting output is a partition of the links and nodes of the original graph into communities.

The data table that contains the assignment of nodes to communities, `mycas.OutNodesComms`, is shown in Output 2.8.1.
Output 2.8.1  Nodes for the Communities of a Simple Undirected Graph

<table>
<thead>
<tr>
<th>node</th>
<th>community_1</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>
<tr>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>2</td>
</tr>
<tr>
<td>H</td>
<td>2</td>
</tr>
<tr>
<td>I</td>
<td>2</td>
</tr>
<tr>
<td>J</td>
<td>3</td>
</tr>
<tr>
<td>K</td>
<td>3</td>
</tr>
<tr>
<td>L</td>
<td>3</td>
</tr>
</tbody>
</table>

The data table that contains the assignment of links to communities, mycas.OutLinksComms, is shown in Output 2.8.2.

Output 2.8.2  Links for the Communities of a Simple Undirected Graph

<table>
<thead>
<tr>
<th>from</th>
<th>to</th>
<th>community_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>D</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>E</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>F</td>
<td>.</td>
</tr>
<tr>
<td>F</td>
<td>G</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>H</td>
<td>2</td>
</tr>
<tr>
<td>F</td>
<td>I</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>H</td>
<td>2</td>
</tr>
<tr>
<td>G</td>
<td>I</td>
<td>2</td>
</tr>
<tr>
<td>I</td>
<td>J</td>
<td>.</td>
</tr>
<tr>
<td>J</td>
<td>K</td>
<td>3</td>
</tr>
<tr>
<td>J</td>
<td>L</td>
<td>3</td>
</tr>
<tr>
<td>K</td>
<td>L</td>
<td>3</td>
</tr>
</tbody>
</table>

The graph seems to have three distinct parts, which are connected by just a few links. The induced subgraphs on these communities are shown in blue in Figure 2.121 through Figure 2.123.
Now, using one call to PROC NETWORK, you can calculate the centrality metrics for all three induced subgraphs by using the BY statement and the links partition defined by the community detection algorithm. In addition, because these subgraphs are completely independent, the processing is done in parallel across machines and threads (depending on your server configuration).
Chapter 2: The NETWORK Procedure

```sas
proc network
  links = mycas.OutLinksComms(where=(community_1 ne .))
  outNodes = mycas.NodeSetOut;
  centrality
    degree
      influence = unweight
      close = unweight
      between = unweight
      eigen = unweight;
  displayout
    ProblemSummary = ProblemSummary
    SolutionSummary = SolutionSummary;
  by community_1;
run;
%put &_NETWORK_;
```

Assuming that your grid has a total of at least three cores, all three subgraphs are processed simultaneously with one call to PROC NETWORK. The progress of the procedure is shown in Output 2.8.3.

**Output 2.8.3** PROC NETWORK Log: Centrality by Cluster for a Simple Undirected Graph

```
NOTE: -------------------------------------------------------------------------------
NOTE: Running NETWORK.
NOTE: -------------------------------------------------------------------------------
NOTE: The number of nodes in the input graph is 5.  
NOTE: The number of links in the input graph is 6.  
NOTE: Processing centrality metrics.  
NOTE: Processing centrality metrics used 0.01 (cpu: 0.00) seconds.  
NOTE: The above message was for the following BY group: community_1=1  
NOTE: The number of nodes in the input graph is 4.  
NOTE: The number of links in the input graph is 5.  
NOTE: Processing centrality metrics.  
NOTE: Processing centrality metrics used 0.05 (cpu: 0.00) seconds.  
NOTE: The above message was for the following BY group: community_1=2  
NOTE: The number of nodes in the input graph is 3.  
NOTE: The number of links in the input graph is 3.  
NOTE: Processing centrality metrics.  
NOTE: Processing centrality metrics used 0.02 (cpu: 0.00) seconds.  
NOTE: The above message was for the following BY group: community_1=3  
NOTE: The CAS table 'PROBLEMSUMMARY' in caslib 'CASUSERHDFS(tiarno)' has 3 rows and 4 columns.  
NOTE: The CAS table 'SOLUTIONSUMMARY' in caslib 'CASUSERHDFS(tiarno)' has 3 rows and 5 columns.  
NOTE: The Cloud Analytic Services server processed the request in 0.338862 seconds.  
NOTE: The data set MYCAS.NODESETOUT has 12 observations and 8 variables.
STATUS=OK  PROBLEM_TYPE=CENTRALITY  CPU_TIME=0.33  REAL_TIME=0.34
```

Notice that links that connect different partitions have been removed by using a WHERE clause on the LINKS= option in the PROC NETWORK statement.
The output table `mycas.ProblemSummary` contains a summary of each induced subgraph that is processed by PROC NETWORK.

### Output 2.8.4 Problem Summary by Community

<table>
<thead>
<tr>
<th>community</th>
<th>numNodes</th>
<th>numLinks</th>
<th>graphDirection</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>6</td>
<td>Undirected</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>Undirected</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>Undirected</td>
</tr>
</tbody>
</table>

The output table `mycas.SolutionSummary` contains a solution summary for the processing on each of the induced subgraphs.

### Output 2.8.5 Solution Summary by Community

<table>
<thead>
<tr>
<th>community</th>
<th>problemType</th>
<th>status</th>
<th>cpuTime</th>
<th>realTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Centrality</td>
<td>OK</td>
<td>0.00</td>
<td>0.01</td>
</tr>
<tr>
<td>2</td>
<td>Centrality</td>
<td>OK</td>
<td>0.00</td>
<td>0.05</td>
</tr>
<tr>
<td>3</td>
<td>Centrality</td>
<td>OK</td>
<td>0.00</td>
<td>0.02</td>
</tr>
</tbody>
</table>

The centrality results (by community) are shown in Output 2.8.6.

### Output 2.8.6 Centrality for All Induced Subgraphs

#### community 1=1

<table>
<thead>
<tr>
<th>node</th>
<th>centr_degree</th>
<th>centr_eigen</th>
<th>centr_close</th>
<th>centr_between</th>
<th>centr_influence1</th>
<th>centr_influence2</th>
</tr>
</thead>
<tbody>
<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>
<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>
</tbody>
</table>

#### community 1=2

<table>
<thead>
<tr>
<th>node</th>
<th>centr_degree</th>
<th>centr_eigen</th>
<th>centr_close</th>
<th>centr_between</th>
<th>centr_influence1</th>
<th>centr_influence2</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>3</td>
<td>1.00000</td>
<td>1.00000</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>0.75000</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>0.75000</td>
<td>0.00000</td>
<td>0.50</td>
<td>1.50</td>
</tr>
<tr>
<td>F</td>
<td>3</td>
<td>1.00000</td>
<td>1.00000</td>
<td>0.16667</td>
<td>0.75</td>
<td>1.75</td>
</tr>
</tbody>
</table>

#### community 1=3

<table>
<thead>
<tr>
<th>node</th>
<th>centr_degree</th>
<th>centr_eigen</th>
<th>centr_close</th>
<th>centr_between</th>
<th>centr_influence1</th>
<th>centr_influence2</th>
</tr>
</thead>
<tbody>
<tr>
<td>K</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.66667</td>
<td>1.33333</td>
</tr>
<tr>
<td>L</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.66667</td>
<td>1.33333</td>
</tr>
<tr>
<td>J</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0.66667</td>
<td>1.33333</td>
</tr>
</tbody>
</table>
Example 2.9: Cycle Enumeration for Kidney Donor Exchange

This example looks at an application of cycle enumeration to help create a kidney donor exchange. Suppose someone needs a kidney transplant and a family member is willing to be a donor. If the donor and recipient are incompatible (because of blood type, tissue mismatch, and so on), the transplant cannot happen. Now suppose two donor-recipient pairs, \( i \) and \( j \), are in this situation, but donor \( i \) is compatible with recipient \( j \) and donor \( j \) is compatible with recipient \( i \). Then two transplants can take place in a two-way swap, shown in Figure 2.124. More generally, an \( n \)-way swap can be performed involving \( n \) donors and \( n \) recipients (Willingham 2009).

Figure 2.124 Kidney Donor Exchange Two-Way Swap

Donor \( i \)  
Recipient \( i \)  
Donor \( j \)  
Recipient \( j \)

Figure 2.125 Kidney Donor Exchange Network

Pair \( i \)  
\( w_{ij} \)  
Pair \( j \)  
\( w_{ji} \)

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 \( j \), as shown in Figure 2.125. 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 \( i \) donates 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 SAS/OR User’s Guide: Network Optimization Algorithms. But doing so would allow arbitrarily long cycles in the solution. For 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 NETWORK to generate the cycles and then PROC OPTMODEL to read the PROC NETWORK output, formulate the set packing problem, call the mixed integer linear programming solver, and output the optimal solution. See Chapter 10, “The OPTMODEL Procedure” (SAS Optimization: Mathematical Optimization Procedures).

The following DATA step sets up the problem by first creating a random graph on \( n \) nodes with link probability \( p \) and Uniform(0,1) weight:
/* create random graph on \( n \) nodes with arc probability \( p \) and uniform(0,1) weight */
%let n = 100;
%let p = 0.02;
data mycas.LinkSetIn;
call streaminit(1);
do from = 0 to &n - 1;
do to = 0 to &n - 1;
  if from eq to then continue;
  else if rand('UNIFORM') < &p then do;
    weight = rand('UNIFORM');
    output;
  end;
end;
end;
run;

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

/* generate all cycles with 2 \( \leq \) length \( \leq \) max_length */
%let max_length = 10;
proc network
  logLevel = moderate
  direction = directed
  links = mycas.LinkSetIn;
cycle
  minLength = 2
  maxLength = &max_length
  out = mycas.Cycles
  maxCycles = all;
run;
%put &_NETWORK_;

PROC NETWORK finds 395 cycles of the appropriate length, as shown in Output 2.9.1.
From the resulting data table `mycas.Cycles`, use the following DATA step to convert the cycles into one observation per arc:

```sas
/* convert cycles into one observation per arc */
data Cycles;
set mycas.Cycles;
run;
proc sort data=Cycles;
by cycle order;
run;
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;
```

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

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

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

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

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

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

   /* call solver */
   solve with milp;

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

PROC OPTMODEL solves the problem by using the mixed integer linear programming solver.
```
Output 2.9.2  Cycles for Kidney Donor Exchange PROC OPTMODEL Log

NOTE: There were 208 observations read from the data set MYCAS.LINKSETIN.
NOTE: There were 3431 observations read from the data set WORK.CYCLES0.
NOTE: Problem generation will use 16 threads.
NOTE: The problem has 395 variables (0 free, 0 fixed).
NOTE: The problem has 395 binary and 0 integer variables.
NOTE: The problem has 64 linear constraints (64 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 3431 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 125 variables and 30 constraints.
NOTE: The MILP presolver removed 1669 constraint coefficients.
NOTE: The MILP presolver modified 0 constraint coefficients.
NOTE: The presolved problem has 270 variables, 34 constraints, and 1762 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 16 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>20.6710373</td>
<td>1147.4221881</td>
<td>98.20%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>20.6710373</td>
<td>25.4194215</td>
<td>18.68%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>20.6710373</td>
<td>25.1227054</td>
<td>17.72%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>3</td>
<td>20.6710373</td>
<td>25.0541979</td>
<td>17.49%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>4</td>
<td>20.8194348</td>
<td>24.9277715</td>
<td>17.08%</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>5</td>
<td>24.8508554</td>
<td>24.8508554</td>
<td>0.00%</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>5</td>
<td>24.8508554</td>
<td>24.8508554</td>
<td>0.00%</td>
<td>1</td>
</tr>
</tbody>
</table>

NOTE: The MILP solver added 11 cuts with 1021 cut coefficients at the root.
NOTE: Objective = 24.850855395.
NOTE: The data set WORK.SOLUTION has 7 observations and 2 variables.
STATUS=OK ALGORITHM=BAC SOLUTION_STATUS=OPTIMAL OBJECTIVE=24.850855395 RELATIVE_GAP=0
ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=4.440892E-16 BOUND_INFEASIBILITY=4.440892E-16
INTEGER_INFEASIBILITY=1.554312E-15 BEST_BOUND=24.850855395 NODES=1 SOLUTIONS_FOUND=5
ITERATIONS=111 PRESOLVE_TIME=0.06 SOLUTION_TIME=1.90

The output data table mycas.Solution, shown in Output 2.9.3, now contains the cycles that define the best exchange and their associated weight (quality).
Example 2.10: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System

Most systems that track software errors, or bugs, 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 dependency*. 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. 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 dependency, and no primary bug is defined for the development team to focus on. You can easily see this circular dependency in the graph representation, because \( A \rightarrow B \rightarrow C \rightarrow D \rightarrow A \). You can find such circular dependencies by using cycle enumeration, which is described in the section “Cycle Enumeration” on page 101. The second situation that needs to be checked is more general. If one 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. You can identify the existence of these chains 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, which define some duplicated bugs (called *defects*) in a small sample of the bug tracking system:

```plaintext
data mycas.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
```

---

Output 2.9.3 Maximum-Quality Solution for Kidney Donor Exchange

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>29</td>
<td>4.3542</td>
</tr>
<tr>
<td>117</td>
<td>4.9748</td>
</tr>
<tr>
<td>200</td>
<td>4.3403</td>
</tr>
<tr>
<td>243</td>
<td>5.0843</td>
</tr>
<tr>
<td>272</td>
<td>1.7253</td>
</tr>
<tr>
<td>277</td>
<td>2.4295</td>
</tr>
<tr>
<td>385</td>
<td>1.9424</td>
</tr>
<tr>
<td>24.8509</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 2: The NETWORK Procedure

The following statements calculate cycles in addition to the transitive closure of the graph $G$ that is defined by the duplicated defects in mycas.DefectLinks. The output data table mycas.Cycles contains any circular dependencies, and the data table mycas.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 $G$.

```sas
proc network
  logLevel    = moderate
  direction   = directed
  links       = mycas.DefectLinks;
  linksVar
    from     = defectId
    to       = linkedDefect;
  cycle
    out      = mycas.Cycles
    maxCycles = all;
run;
%put &_NETWORK_;

proc network
  logLevel    = moderate
  direction   = directed
  links       = mycas.DefectLinks;
  linksVar
    from     = defectId
    to       = linkedDefect;
  transitiveClosure
    out      = mycas.TransClosure;
run;
%put &_NETWORK_;

proc sql;
  create table Chains as
    select defectId, linkedDefect
    from mycas.TransClosure(where=(defectId ne linkedDefect)) except
    select defectId, linkedDefect
    from mycas.DefectLinks;
quit;
```

The progress of the procedure is shown in Output 2.10.1.
Example 2.10: Transitive Closure for Identification of Circular Dependencies

Output 2.10.1 PROC NETWORK Log: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System

| NOTE: Running NETWORK. |
| NOTE: Reading the links data. |
| NOTE: Building the input (full) graph storage used 0.00 (cpu: 0.00) seconds. |
| NOTE: The number of nodes in the input graph is 16. |
| NOTE: Processing cycle detection. |

Output 2.10.2 displays the output data table mycas.Cycles, which contains one case of a circular dependency in which the DUPs start and end at S0152674.

Output 2.10.2 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>

Output 2.10.3 displays the local data set Chains, which contains the chains in the bug tracking system that come from the links in $G^T$ that are not in $G$.
Example 2.11: Reach Networks for Computing the 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 a 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 do this, you could first generate all the reach networks for each customer by using PROC NETWORK. These networks could then be used in a set-covering problem, which you can solve as an integer linear program by using PROC OPTMODEL. Let $N$ be the set of customers that you want to reach, and let the links $A$ define the social network of 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 the advertisement 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 “Example 2.1: Articulation Points in a Terrorist Network” on page 153. In this case, customers are alleged terrorists. Solving the covering problem here can give you a subset of people to focus on in an investigation in order to cover all members of the network.

The following macro, %GenerateReach, runs PROC NETWORK to generate the reach network for each person in the terrorist network for a variable hop limit:

```sas
%macro GenerateReach(limit=);
proc network
  outNodes = mycas.NodeSetOut
  links = mycas.LinkSetInTerror911;
  reach
    eachSource
    outNodes = mycas.ReachNode
    maxReach = &limit;
run;
%mend GenerateReach;
```

The following macro, %SolverCover, runs PROC OPTMODEL to solve the set-covering problem:
Example 2.11: Reach Networks for Computing the Market Coverage of a Terrorist Network

%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 mycas.NodeSetOut into NODE_ID=_[n_] nodeIdToLabel=node;
    read data mycas.ReachNode into PAIRS=[node reach];
    for{i in NODE_ID} do;
        tmpLabel = nodeIdToLabel[i];
        NODE_LABEL = NODE_LABEL union {tmpLabel};
        nodeLabelToId[tmpLabel] = i;
    end;
    for{<label,i> in PAIRS} do;
        REACH_SET[i] = REACH_SET[i] union {nodeLabelToId[label]};
    end;

    /* declare decision variables */
    var x {NODE_ID} binary;

    /* declare objective */
    minimize numNodes = sum{j in NODE_ID} x[j];

    /* cover constraint */
    con cover {i in NODE_ID}:
        sum{j in REACH_SET[i]} x[j] >= 1;

    /* solve */
    solve;

    create data Solution from [label]=
        (setof{j in NODE_ID : round(x[j].sol)=1}nodeIdToLabel[j]);
quit;
%mend SolverCover;

The following statements calculate the minimal cover for the one-hop limit:

    %GenerateReach(limit=1);
    %SolverCover();

To cover the network, assuming a one-hop limit, the investigators would need to investigate the people listed in the data table mycas.Solution, shown in Output 2.11.1.
Output 2.11.1 Minimal One-Hop Cover for Terrorist Communications Network

<table>
<thead>
<tr>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mohamed Atta</td>
</tr>
<tr>
<td>Hani Hanjour</td>
</tr>
<tr>
<td>Djamal Beghal</td>
</tr>
<tr>
<td>Nabil al-Marabh</td>
</tr>
<tr>
<td>Essid Sami Ben Khemais</td>
</tr>
<tr>
<td>Fayez Ahmed</td>
</tr>
<tr>
<td>Zacarias Moussaoui</td>
</tr>
<tr>
<td>Mamduh Mahmud Salim</td>
</tr>
<tr>
<td>Nawaf AlHazmi</td>
</tr>
<tr>
<td>Mohammed Belfas</td>
</tr>
</tbody>
</table>

The following statements calculate the minimal cover for the two-hop limit:

```sas
%GenerateReach(limit=2);
%SolverCover();
```

If investigators assume a two-hop limit, they could focus their attention on the two people shown in Output 2.11.2. Then, by following their links (and their links’ links), they could cover the entire network.

Output 2.11.2 Minimal Two-Hop Cover for Terrorist Communications Network

<table>
<thead>
<tr>
<th>label</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mohamed Atta</td>
</tr>
<tr>
<td>Zacarias Moussaoui</td>
</tr>
</tbody>
</table>

Example 2.12: Connected Components for US Patent Citations

This example looks at the structural relationship of US patent citations by using a large data set that is maintained by the Stanford Network Analysis Project (SNAP) (Leskovec 2014). The citation graph includes over 16 million citations made to patents between 1975 and 1999.

The following statements construct the links data table mycas.Patents from a local copy of the raw patent citation data:

```sas
filename in 'cit-Patents.txt';
data mycas.Patents;
infile in firstobs=5 dlm='09'X;
input from to;
run;
```

The following statements find the connected components of the citation graph by using a distributed union-find algorithm. This algorithm takes advantage of all the machines in your configured session.

```sas
proc network
  links = mycas.Patents
  outNodes = mycas.OutNodes;
connectedComponents
  algorithm = parallel;
```
The progress of the procedure is shown in **Output 2.12.1**.

### Output 2.12.1 PROC NETWORK Log: Connected Components for US Patent Citations

<table>
<thead>
<tr>
<th>NOTE:</th>
<th>---</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running NETWORK.</td>
<td></td>
</tr>
<tr>
<td>The graph contains 1 self links that are ignored.</td>
<td></td>
</tr>
<tr>
<td>The number of nodes in the input graph is 3774768.</td>
<td></td>
</tr>
<tr>
<td>The number of links in the input graph is 16518947.</td>
<td></td>
</tr>
<tr>
<td>Processing connected components using 4 machines.</td>
<td></td>
</tr>
<tr>
<td>The graph has 3627 connected components.</td>
<td></td>
</tr>
<tr>
<td>Processing connected components used 2.42 (cpu: 3.51) seconds.</td>
<td></td>
</tr>
<tr>
<td>The Cloud Analytic Services server processed the request in 17.168292 seconds.</td>
<td></td>
</tr>
<tr>
<td>The data set MYCAS.OUTNODES has 3774768 observations and 2 variables.</td>
<td></td>
</tr>
<tr>
<td>STATUS=OK  PROBLEM_TYPE=CONNECTEDCOMPONENTS  SOLUTION_STATUS=OK  NUM_COMPONENTS=3627</td>
<td></td>
</tr>
<tr>
<td>CPU_TIME=46.72  REAL_TIME=17.17</td>
<td></td>
</tr>
</tbody>
</table>

The following statements use PROC SQL to calculate the size of each component:

```
proc sql;
  create table FreqCount as
  select concomp, count(*) as count
  from mycas.OutNodes
  group by concomp
  order by count descending;
quit;
```

The 10 biggest components are shown in **Output 2.12.2**. It is interesting to note that the vast majority of patents (over 99%) are all contained in the same component. This is not too surprising, because many of the seminal patent claims are required in order to understand subsequent inventions.

### Output 2.12.2 Ten Largest Components for US Patent Citations

<table>
<thead>
<tr>
<th>Obs</th>
<th>concomp</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 3764117</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>446</td>
<td>19</td>
</tr>
<tr>
<td>3</td>
<td>242</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>25</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>263</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>345</td>
<td>14</td>
</tr>
<tr>
<td>7</td>
<td>169</td>
<td>14</td>
</tr>
<tr>
<td>8</td>
<td>1431</td>
<td>14</td>
</tr>
<tr>
<td>9</td>
<td>239</td>
<td>13</td>
</tr>
<tr>
<td>10</td>
<td>158</td>
<td>13</td>
</tr>
</tbody>
</table>
Chapter 2: The NETWORK Procedure

Example 2.13: Shortest Paths of the New York Road Network

This example looks at the road networks in the state of New York (NY). The distance graph raw data are maintained at the DIMACS challenge website (Demetrescu 2010). The NY road network includes 264,346 intersections (nodes) and 365,050 roads (links). Although the input data table is not large, the computing power that you need to find all-pairs shortest paths is enormous. In addition, the storage space that you need to handle the results data can easily overwhelm the capacity of a single machine. In this example, a session of 130 machines (each with 32 cores) was configured to process this graph.

The following statements construct the links data table mycas.RoadNY from a local copy of the raw distance graph data:

```
filename in 'USA-road-d.NY.gr';
data mycas.RoadNY (drop=a);
infile in firstobs=8;
input a $ from $ to $ weight;
run;
```

The following statements find the all-pairs shortest paths of the NY road network (that have a total path weight of less than 20,000) by using a distributed algorithm. This algorithm takes advantage of all the machines and cores in your configured session.

```
proc network
   logFreqTime = 10
   logLevel = aggressive
   links = mycas.RoadNY;
   shortestPath
      maxPathWeight = 20000
      outWeights = mycas.shortPathSummary
      outPaths = mycas.shortPathPaths;
run;
%put &_NETWORK_;
```

The progress of the procedure is shown in Output 2.13.1.
Example 2.14: Shortest Path in a Road Network by Date and Time

This example reconsiders the road network between a SAS employee’s home in Raleigh, North Carolina, and SAS headquarters nearby in Cary introduced in the section “Road Network Shortest Path” on page 10. The following data provide a snapshot of the road network and travel times observed at three different times:

```sas
data mycas.LinkSetInRoadNC;
  input start_inter $1-20 end_inter $21-40 miles miles_per_hour date date11. time time10.;
  format date date11. time time10.;
  time_to_travel = miles * 1/miles_per_hour * 60;
datalines;
614CapitalBlvd Capital/WadeAve 0.6 25 15-APR-2013 10:30 am
```

Notice that the resulting output data tables, mycas.shortPathSummary and mycas.shortPathPaths, are large distributed data tables.

Example 2.14: Shortest Path in a Road Network by Date and Time

This example reconsiders the road network between a SAS employee’s home in Raleigh, North Carolina, and SAS headquarters nearby in Cary introduced in the section “Road Network Shortest Path” on page 10. The following data provide a snapshot of the road network and travel times observed at three different times:
The first snapshot (15-APR-2013 10:30 am) is a typical traffic pattern on a workday. The second snapshot (16-APR-2013 9:30 am) represents morning rush-hour traffic, and the third (18-APR-2013 8:30 am) represents rush-hour traffic where a major highway (US70W) has been closed for repairs.

The following statements find the route that yields the shortest path between home (614 Capital Boulevard) and SAS headquarters (SAS Campus Drive) for all three scenarios simultaneously by using the BY statement:

```sas
proc network
  links = mycas.LinkSetInRoadNC;
  linksVar
    from = start_inter
    to = end_inter
    weight = time_to_travel;
  shortestPath
    outPaths = mycas.ShortPathP
    outWeights = mycas.ShortPathW
    source = "614CapitalBlvd"
    sink = "SASCampusDrive";
  displayout
    ProblemSummary = ProblemSummary
    SolutionSummary = SolutionSummary;
  by date time;
run;
%put &_NETWORK_;```

The first snapshot (15-APR-2013 10:30 am) is a typical traffic pattern on a workday. The second snapshot (16-APR-2013 9:30 am) represents morning rush-hour traffic, and the third (18-APR-2013 8:30 am) represents rush-hour traffic where a major highway (US70W) has been closed for repairs.

The following statements find the route that yields the shortest path between home (614 Capital Boulevard) and SAS headquarters (SAS Campus Drive) for all three scenarios simultaneously by using the BY statement:
Example 2.14: Shortest Path in a Road Network by Date and Time

Assuming that your grid has a total of at least three cores, all three graphs are processed simultaneously through one call to PROC NETWORK. The progress of the procedure is shown in Output 2.14.1.

**Output 2.14.1** PROC NETWORK Log: Shortest Path in a Road Network by Date and Time

```
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 11.
NOTE: Processing the shortest paths problem using 32 threads.
NOTE: Processing the shortest paths problem between 1 source nodes and 1 sink nodes.
NOTE: Processing the shortest paths problem used 0.00 (cpu: 0.00) seconds.
NOTE: The above message was for the following BY group:
    date=15-APR-2013 time=10:30:00
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 11.
NOTE: Processing the shortest paths problem using 32 threads.
NOTE: Processing the shortest paths problem between 1 source nodes and 1 sink nodes.
NOTE: Processing the shortest paths problem used 0.02 (cpu: 0.00) seconds.
NOTE: The above message was for the following BY group:
    date=16-APR-2013 time=9:30:00
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 8.
NOTE: Processing the shortest paths problem using 32 threads.
NOTE: Processing the shortest paths problem between 1 source nodes and 1 sink nodes.
NOTE: Processing the shortest paths problem used 0.00 (cpu: 0.00) seconds.
NOTE: The above message was for the following BY group:
    date=18-APR-2013 time=8:30:00
```

Output 2.14.2 displays the output table mycas.ProblemSummary, which contains a summary of each graph that is processed by PROC NETWORK.

**Output 2.14.2** Problem Summary by Date and Time

```
<table>
<thead>
<tr>
<th>date</th>
<th>time</th>
<th>numNodes</th>
<th>numLinks</th>
<th>graphDirection</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-APR-2013</td>
<td>10:30:00</td>
<td>10</td>
<td>11</td>
<td>Undirected</td>
</tr>
<tr>
<td>16-APR-2013</td>
<td>9:30:00</td>
<td>10</td>
<td>11</td>
<td>Undirected</td>
</tr>
<tr>
<td>18-APR-2013</td>
<td>8:30:00</td>
<td>8</td>
<td>8</td>
<td>Undirected</td>
</tr>
</tbody>
</table>
```

Output 2.14.3 displays the output table mycas.SolutionSummary, which contains a solution summary for the processing on each graph.
Output 2.14.3 Solution Summary by Date and Time

<table>
<thead>
<tr>
<th>date</th>
<th>time</th>
<th>problemType</th>
<th>status</th>
<th>numPaths</th>
<th>cpuTime</th>
<th>realTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-APR-2013</td>
<td>10:30:00</td>
<td>Shortest Path</td>
<td>OK</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>16-APR-2013</td>
<td>9:30:00</td>
<td>Shortest Path</td>
<td>OK</td>
<td>1</td>
<td>0.00</td>
<td>0.02</td>
</tr>
<tr>
<td>18-APR-2013</td>
<td>8:30:00</td>
<td>Shortest Path</td>
<td>OK</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
</tr>
</tbody>
</table>

Output 2.14.4 displays the output data table mycas.ShortPathW, which shows the total time to travel on the best route for each time snapshot.

Output 2.14.4 Shortest Path Summary for Road Network at Each Date and Time

<table>
<thead>
<tr>
<th>date</th>
<th>time</th>
<th>source</th>
<th>sink</th>
<th>path_weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-APR-2013</td>
<td>10:30:00</td>
<td>614CapitalBlvd</td>
<td>SASCampusDrive</td>
<td>11.5582</td>
</tr>
<tr>
<td>16-APR-2013</td>
<td>9:30:00</td>
<td>614CapitalBlvd</td>
<td>SASCampusDrive</td>
<td>12.9582</td>
</tr>
<tr>
<td>18-APR-2013</td>
<td>8:30:00</td>
<td>614CapitalBlvd</td>
<td>SASCampusDrive</td>
<td>14.2582</td>
</tr>
</tbody>
</table>

Output 2.14.5 displays the output data table mycas.ShortPathP, which shows (by date and time) the best route for each time snapshot.

Output 2.14.5 Shortest Path for Road Network by Date and Time

<table>
<thead>
<tr>
<th>date=15-APR-2013 time=10:30:00</th>
<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>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Capital/WadeAve</td>
<td>WadeAve/RaleighExpy</td>
<td>4.5000</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>WadeAve/RaleighExpy</td>
<td>RaleighExpy/US40W</td>
<td>3.0000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>RaleighExpy/US40W</td>
<td>US40W/HarrisonAve</td>
<td>1.4182</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>US40W/HarrisonAve</td>
<td>SASCampusDrive</td>
<td>1.2000</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>date=16-APR-2013 time=9:30:00</th>
<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/US70W</td>
<td>1.4400</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>US70W/US440W</td>
<td>US440W/RaleighExpy</td>
<td>2.7000</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>US440W/RaleighExpy</td>
<td>RaleighExpy/US40W</td>
<td>3.0000</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>RaleighExpy/US40W</td>
<td>US40W/HarrisonAve</td>
<td>1.4182</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>US40W/HarrisonAve</td>
<td>SASCampusDrive</td>
<td>1.2000</td>
<td></td>
</tr>
</tbody>
</table>

| date=16-APR-2013 time=9:30:00 | time | 12.9582 |
| date                          | 12.9582 |
Example 2.15: Pattern Matching in a Social Network

This example considers a portion of a social network that conveys relationships between people (friends), residences (lives in), and preferences for particular restaurants (likes). The network, directed graph $G$, is shown in Figure 2.126.

**Figure 2.126** Social Network $G$

The following data provide a snapshot of the social connections between Matt and a few of his friends:

```plaintext
data mycas.NodesSocial;
  infile datalines dsd;
  length node $40. type $40. subtype $20. ;
  input node $ type $ subtype $;
  label=node;
  datalines;
  Matt, Person,
  Rob, Person,
  Chuck, Person,
  Stephen, Person,
  Manoj, Person,
  Bryan, Person,
  Jack, Person,
  Raleigh, City,
  Philadelphia, City,
```
data mycas.LinksSocial;
  infile datalines dsd;
  length from $40. to $40. connection $20.;
  input from $ to $ connection $;
  datalines;
  Matt, Rob, friends
  Rob, Matt, friends
  Matt, Chuck, friends
  Chuck, Matt, friends
  Chuck, Rob, friends
  Rob, Chuck, friends
  Jack, Rob, friends
  Rob, Jack, friends
  Matt, Stephen, friends
  Stephen, Matt, friends
  Matt, Manoj, friends
  Manoj, Matt, friends
  Matt, Bryan, friends
  Bryan, Matt, friends
  Matt, Jack, friends
  Jack, Matt, friends
  Matt, Philadelphia, lives in
  Stephen, Philadelphia, lives in
  Stephen, JimmyJs, likes
  Stephen, Cafe Luna, likes
  Rob, Raleigh, lives in
  Chuck, Raleigh, lives in
  Manoj, Raleigh, lives in
  Jack, Raleigh, lives in
  Bryan, Charlotte, lives in
  Rob, The Pit Authentic, likes
  Jack, Red Hot Blue, likes
  Chuck, The Pit Authentic, likes
  Chuck, Cafe Luna, likes
  Chuck, Second Empire, likes
  Jack, Vivo Rist, likes
  Manoj, Dumplings, likes
  Bryan, Red Hot Blue, likes
  Bryan, Vivo Rist, likes
  Rob, Moonlight, likes
;

The nodes in the nodes data table mycas.NodesSocial represent people, cities, and restaurants. The node attribute type defines the node type. In the case of a restaurant, the node attribute subtype defines the type of
restaurant.

The links in the links data table mycas.LinksSocial represent connections between the nodes. The type of connection is defined by the link attribute connection.

For these data, a typical social network pattern search might be to find “friends of Matt who like barbecue restaurants.” This pattern is shown in Figure 2.127.

![Query Graph Q](image)

In order to construct this pattern, the query graph can be represented using the data that are created by the following DATA steps:

``` SAS
data mycas.NodesSocialQuery;
  infile datalines dsd;
  length node $40. label $40. type $40. subtype $20.;
  input node $ label $ type $ subtype $;
  datalines;
  Matt, Matt, Person,
  X,, Person,
  BBQ,, Restaurant, BBQ
;*

data mycas.LinksSocialQuery;
  infile datalines dsd;
  length from $40. to $40. connection $20.;
  input from $ to $ connection $;
  datalines;
  Matt, X, friends
  X, Matt, friends
  X, BBQ, likes
;*
```

The query node Matt must be a person with the node attribute label=Matt. The query node X can be any person. The query node BBQ must be a barbecue restaurant (that is, type=Restaurant and subtype=BBQ). Person Matt and person X must be friends, and person X must have liked the restaurant that is assigned to node BBQ.

You can use the following statements to find all subgraphs that have the specified pattern:

``` SAS
proc network
  direction = directed
  links = mycas.LinksSocial
  nodes = mycas.NodesSocial
  linksQuery = mycas.LinksSocialQuery
  nodesQuery = mycas.NodesSocialQuery;
  nodesVar
    vars = (label type subtype);
  nodesQueryVar
```
The progress of the procedure is shown in Output 2.15.1.

### Output 2.15.1 PROC NETWORK Log: Pattern Matching in a Social Network

<table>
<thead>
<tr>
<th>NOTE:</th>
<th>Running NETWORK.</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOTE:</td>
<td>The number of nodes in the input graph is 18.</td>
</tr>
<tr>
<td>NOTE:</td>
<td>The number of links in the input graph is 35.</td>
</tr>
<tr>
<td>NOTE:</td>
<td>The number of nodes in the query graph is 3.</td>
</tr>
<tr>
<td>NOTE:</td>
<td>The number of links in the query graph is 3.</td>
</tr>
<tr>
<td>NOTE:</td>
<td>Processing the pattern matching query using 32 threads.</td>
</tr>
<tr>
<td>NOTE:</td>
<td>The Cloud Analytic Services server processed the request in 0.239177 seconds.</td>
</tr>
<tr>
<td>NOTE:</td>
<td>The data set MYCAS.OUTMATCHNODES has 15 observations and 6 variables.</td>
</tr>
<tr>
<td>NOTE:</td>
<td>The data set MYCAS.OUTMATCHLINKS has 15 observations and 4 variables.</td>
</tr>
<tr>
<td>STATUS=OK</td>
<td>PROBLEM_TYPE=_PATTERNMATCH SOLUTION_STATUS=OK NUM_MATCHES=5 CPU_TIME=0.20 REAL_TIME=0.24</td>
</tr>
</tbody>
</table>

Output 2.15.2 displays the output data table mycas.OutMatchNodes, which shows the mappings from nodes in the query graph to nodes in the input graph for each match. For this query, five friends (X) match the specified criteria: Bryan, Jack, Stephen, Chuck, and Rob.

### Output 2.15.2 Node Mappings for Friends Who Like Barbecue

<table>
<thead>
<tr>
<th>match</th>
<th>nodeQ</th>
<th>node</th>
<th>label</th>
<th>type</th>
<th>subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 BBQ</td>
<td>Red Hot Blue</td>
<td>Red Hot Blue</td>
<td>Restaurant BBQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Matt</td>
<td>Matt</td>
<td>Matt</td>
<td>Person</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 X</td>
<td>Bryan</td>
<td>Bryan</td>
<td>Person</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 BBQ</td>
<td>Red Hot Blue</td>
<td>Red Hot Blue</td>
<td>Restaurant BBQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 Matt</td>
<td>Matt</td>
<td>Matt</td>
<td>Person</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 X</td>
<td>Jack</td>
<td>Jack</td>
<td>Person</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 BBQ</td>
<td>JimmyJs</td>
<td>JimmyJs</td>
<td>Restaurant BBQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 Matt</td>
<td>Matt</td>
<td>Matt</td>
<td>Person</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 X</td>
<td>Stephen</td>
<td>Stephen</td>
<td>Person</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 BBQ</td>
<td>The Pit Authentic</td>
<td>The Pit Authentic</td>
<td>Restaurant BBQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 Matt</td>
<td>Matt</td>
<td>Matt</td>
<td>Person</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 X</td>
<td>Rob</td>
<td>Rob</td>
<td>Person</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 BBQ</td>
<td>The Pit Authentic</td>
<td>The Pit Authentic</td>
<td>Restaurant BBQ</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 Matt</td>
<td>Matt</td>
<td>Matt</td>
<td>Person</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 X</td>
<td>Chuck</td>
<td>Chuck</td>
<td>Person</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Output 2.15.3 displays the output data table mycas.OutMatchLinks, which shows the subgraphs for each match.

**Output 2.15.3** Subgraphs for Friends Who Like Barbecue

<table>
<thead>
<tr>
<th>match</th>
<th>from</th>
<th>to</th>
<th>connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Bryan</td>
<td>Matt</td>
<td>friends</td>
</tr>
<tr>
<td>1</td>
<td>Bryan</td>
<td>Red Hot Blue</td>
<td>likes</td>
</tr>
<tr>
<td>1</td>
<td>Matt</td>
<td>Bryan</td>
<td>friends</td>
</tr>
<tr>
<td>2</td>
<td>Jack</td>
<td>Matt</td>
<td>friends</td>
</tr>
<tr>
<td>2</td>
<td>Jack</td>
<td>Red Hot Blue</td>
<td>likes</td>
</tr>
<tr>
<td>2</td>
<td>Matt</td>
<td>Jack</td>
<td>friends</td>
</tr>
<tr>
<td>3</td>
<td>Matt</td>
<td>Stephen</td>
<td>friends</td>
</tr>
<tr>
<td>3</td>
<td>Stephen</td>
<td>JimmyJs</td>
<td>likes</td>
</tr>
<tr>
<td>3</td>
<td>Stephen</td>
<td>Matt</td>
<td>friends</td>
</tr>
<tr>
<td>4</td>
<td>Matt</td>
<td>Rob</td>
<td>friends</td>
</tr>
<tr>
<td>4</td>
<td>Rob</td>
<td>Matt</td>
<td>friends</td>
</tr>
<tr>
<td>4</td>
<td>Rob</td>
<td>The Pit Authentic</td>
<td>likes</td>
</tr>
<tr>
<td>5</td>
<td>Chuck</td>
<td>Matt</td>
<td>friends</td>
</tr>
<tr>
<td>5</td>
<td>Chuck</td>
<td>The Pit Authentic</td>
<td>likes</td>
</tr>
<tr>
<td>5</td>
<td>Matt</td>
<td>Chuck</td>
<td>friends</td>
</tr>
</tbody>
</table>

Next, you can find “friends of Matt who like barbecue restaurants and live in Raleigh,” shown in Figure 2.128, by using the data that are created by the following DATA steps and the same call to PROC NETWORK as before:

```plaintext
data mycas.NodesSocialQuery;
   infile datalines dsd;
   length node $40. label $40. type $40. subtype $20.;
   input node $ label $ type $ subtype $;
datalines;
Matt, Matt, Person,
X,, Person,
Raleigh, Raleigh, City,
BBQ,, Restaurant, BBQ
;
data mycas.LinksSocialQuery;
   infile datalines dsd;
   length from $40. to $40. connection $20.;
   input from $ to $ connection $;
datalines;
Matt, X, friends
X, Matt, friends
X, Raleigh, lives in
X, BBQ, likes
;```
Output 2.15.4 displays the output data table mycas.OutMatchNodes. For this query, three friends (X) match the specified criteria: Jack, Chuck, and Rob.

**Output 2.15.4 Node Mappings for Friends Who Like Barbecue and Live in Raleigh**

<table>
<thead>
<tr>
<th>match</th>
<th>nodeQ</th>
<th>node</th>
<th>label</th>
<th>type</th>
<th>subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BBQ</td>
<td>The Pit Authentic</td>
<td>The Pit Authentic</td>
<td>Restaurant</td>
<td>BBQ</td>
</tr>
<tr>
<td>1</td>
<td>Matt</td>
<td>Matt</td>
<td>Matt</td>
<td>Person</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Raleigh</td>
<td>Raleigh</td>
<td>Raleigh</td>
<td>City</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>Chuck</td>
<td>Chuck</td>
<td>Person</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BBQ</td>
<td>The Pit Authentic</td>
<td>The Pit Authentic</td>
<td>Restaurant</td>
<td>BBQ</td>
</tr>
<tr>
<td>2</td>
<td>Matt</td>
<td>Matt</td>
<td>Matt</td>
<td>Person</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Raleigh</td>
<td>Raleigh</td>
<td>Raleigh</td>
<td>City</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>Rob</td>
<td>Rob</td>
<td>Person</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>BBQ</td>
<td>Red Hot Blue</td>
<td>Red Hot Blue</td>
<td>Restaurant</td>
<td>BBQ</td>
</tr>
<tr>
<td>3</td>
<td>Matt</td>
<td>Matt</td>
<td>Matt</td>
<td>Person</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Raleigh</td>
<td>Raleigh</td>
<td>Raleigh</td>
<td>City</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>Jack</td>
<td>Jack</td>
<td>Person</td>
<td></td>
</tr>
</tbody>
</table>

Output 2.15.5 displays the output data table mycas.OutMatchLinks.
Example 2.15: Pattern Matching in a Social Network

Output 2.15.5 Subgraphs for Friends Who Like Barbecue and Live in Raleigh

<table>
<thead>
<tr>
<th>match</th>
<th>from</th>
<th>to</th>
<th>connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Chuck</td>
<td>Matt</td>
<td>friends</td>
</tr>
<tr>
<td>1</td>
<td>Chuck</td>
<td>Raleigh</td>
<td>lives in</td>
</tr>
<tr>
<td>1</td>
<td>Chuck</td>
<td>The Pit Authentic</td>
<td>likes</td>
</tr>
<tr>
<td>1</td>
<td>Matt</td>
<td>Chuck</td>
<td>friends</td>
</tr>
<tr>
<td>2</td>
<td>Matt</td>
<td>Rob</td>
<td>friends</td>
</tr>
<tr>
<td>2</td>
<td>Rob</td>
<td>Matt</td>
<td>friends</td>
</tr>
<tr>
<td>2</td>
<td>Rob</td>
<td>Raleigh</td>
<td>lives in</td>
</tr>
<tr>
<td>2</td>
<td>Rob</td>
<td>The Pit Authentic</td>
<td>likes</td>
</tr>
<tr>
<td>3</td>
<td>Jack</td>
<td>Matt</td>
<td>friends</td>
</tr>
<tr>
<td>3</td>
<td>Jack</td>
<td>Raleigh</td>
<td>lives in</td>
</tr>
<tr>
<td>3</td>
<td>Jack</td>
<td>Red Hot Blue</td>
<td>likes</td>
</tr>
<tr>
<td>3</td>
<td>Matt</td>
<td>Jack</td>
<td>friends</td>
</tr>
</tbody>
</table>

Finally, you can find “a pair of friends of Matt who like the same barbecue restaurant, live in Raleigh, and are friends of each other,” shown in Figure 2.129, by using the data that are created by the following DATA steps and the same call to PROC NETWORK as before:

```plaintext
data mycas.NodesSocialQuery;
    infile datalines dsd;
    length node $40. label $40. type $40. subtype $20.;
    input node $ label $ type $ subtype $;
    datalines;
    Matt, Matt, Person,
    X,, Person,
    Y,, Person,
    Raleigh, Raleigh, City,
    BBQ,, Restaurant, BBQ
;
data mycas.LinksSocialQuery;
    infile datalines dsd;
    length from $40. to $40. connection $20.;
    input from $ to $ connection $;
    datalines;
    Matt, X, friends
    X, Matt, friends
    Matt, Y, friends
    Y, Matt, friends
    X, Raleigh, lives in
    Y, Raleigh, lives in
    X, BBQ, likes
    Y, BBQ, likes
    X, Y, friends
    Y, X, friends
;```
Figure 2.129 Query Graph $Q$

Output 2.15.6 displays the output data table `mycas.OutMatchNodes`. For this query, only one pair of friends (Chuck and Rob) match the specified criteria. There are two isomorphic mappings: one where $X=Chuck$, $Y=Rob$ and one where $X=Rob$, $Y=Chuck$.

### Output 2.15.6 Node Mappings for a Pair of Friends

<table>
<thead>
<tr>
<th>match</th>
<th>nodeQ</th>
<th>node</th>
<th>label</th>
<th>type</th>
<th>subtype</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>BBQ</td>
<td>The Pit Authentic</td>
<td>The Pit Authentic</td>
<td>Restaurant</td>
<td>BBQ</td>
</tr>
<tr>
<td>1</td>
<td>Matt</td>
<td>Matt</td>
<td>Matt</td>
<td>Person</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Raleigh</td>
<td>Raleigh</td>
<td>Raleigh</td>
<td>City</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>Chuck</td>
<td>Chuck</td>
<td>Person</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Y</td>
<td>Rob</td>
<td>Rob</td>
<td>Person</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>BBQ</td>
<td>The Pit Authentic</td>
<td>The Pit Authentic</td>
<td>Restaurant</td>
<td>BBQ</td>
</tr>
<tr>
<td>2</td>
<td>Matt</td>
<td>Matt</td>
<td>Matt</td>
<td>Person</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Raleigh</td>
<td>Raleigh</td>
<td>Raleigh</td>
<td>City</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>Rob</td>
<td>Rob</td>
<td>Person</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Y</td>
<td>Chuck</td>
<td>Chuck</td>
<td>Person</td>
<td></td>
</tr>
</tbody>
</table>

Output 2.15.7 displays the output data table `mycas.OutMatchLinks`.
Example 2.16: Node Similarity for Link Prediction

The goal of link prediction is to recover missing links—connections that are likely to exist between nodes but have not been recorded because of incomplete data. This example considers a network of 168 nodes and 219 links, where nodes represent individuals and links represent wiretap records. The data set is based on information from operation Oversize, an Italian criminal case against a mafia group that was involved in international drug trafficking, homicides, and robberies (Berlusconi et al. 2016). This network, an undirected graph $G$, is shown in Figure 2.130.

Output 2.15.7 Subgraphs for a Pair of Friends

<table>
<thead>
<tr>
<th>match</th>
<th>from</th>
<th>to</th>
<th>connection</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Chuck</td>
<td>Matt</td>
<td>friends</td>
</tr>
<tr>
<td>1</td>
<td>Chuck</td>
<td>Raleigh</td>
<td>lives in</td>
</tr>
<tr>
<td>1</td>
<td>Chuck</td>
<td>Rob</td>
<td>friends</td>
</tr>
<tr>
<td>1</td>
<td>Chuck</td>
<td>The Pit Authentic</td>
<td>likes</td>
</tr>
<tr>
<td>1</td>
<td>Matt</td>
<td>Chuck</td>
<td>friends</td>
</tr>
<tr>
<td>1</td>
<td>Matt</td>
<td>Rob</td>
<td>friends</td>
</tr>
<tr>
<td>1</td>
<td>Rob</td>
<td>Chuck</td>
<td>friends</td>
</tr>
<tr>
<td>1</td>
<td>Rob</td>
<td>Matt</td>
<td>friends</td>
</tr>
<tr>
<td>1</td>
<td>Rob</td>
<td>Raleigh</td>
<td>lives in</td>
</tr>
<tr>
<td>1</td>
<td>Rob</td>
<td>The Pit Authentic</td>
<td>likes</td>
</tr>
<tr>
<td>2</td>
<td>Chuck</td>
<td>Matt</td>
<td>friends</td>
</tr>
<tr>
<td>2</td>
<td>Chuck</td>
<td>Raleigh</td>
<td>lives in</td>
</tr>
<tr>
<td>2</td>
<td>Chuck</td>
<td>Rob</td>
<td>friends</td>
</tr>
<tr>
<td>2</td>
<td>Chuck</td>
<td>The Pit Authentic</td>
<td>likes</td>
</tr>
<tr>
<td>2</td>
<td>Matt</td>
<td>Chuck</td>
<td>friends</td>
</tr>
<tr>
<td>2</td>
<td>Matt</td>
<td>Rob</td>
<td>friends</td>
</tr>
<tr>
<td>2</td>
<td>Rob</td>
<td>Chuck</td>
<td>friends</td>
</tr>
<tr>
<td>2</td>
<td>Rob</td>
<td>Matt</td>
<td>friends</td>
</tr>
<tr>
<td>2</td>
<td>Rob</td>
<td>Raleigh</td>
<td>lives in</td>
</tr>
<tr>
<td>2</td>
<td>Rob</td>
<td>The Pit Authentic</td>
<td>likes</td>
</tr>
</tbody>
</table>
The data that are created by the following DATA step provide the connections that were obtained through wiretapping:

```sas
  data mycas.LinkSetInWR;
    input from $ to $ @@;
  datalines;
  n8 n5 n9 n1 n9 n7 n10 n9 n11 n9 n17 n9 n19 n13 n20 n9
  n21 n3 n21 n13 n21 n18 n24 n13 n24 n19 n26 n6 n26 n15 n26 n25
```

Figure 2.130  Wiretap Network $G$
The persons of interest—the two main traffickers within the criminal network, $n_{49}$ and $n_{27}$; the drug dealer $n_{13}$; $n_{49}$'s brother and drug dealer, $n_{50}$; and $n_{118}$, the wife of a known drug dealer—are selected as a node subset to probe what other people in the network they are likely to be connected to.

The following statements define the nodes that are the sources of the new predicted links, with all other nodes as candidates:

```plaintext
data mycas.NodeSubSetIn;
  input node $ source;
  datalines;
  n27  1
  n49  1
  n13  1
  n50  1
  n118 1
;
```

The following statements produce the output data table `mycas.NodeSim`, which contains the Jaccard similarity between the nodes in the input subset and all other nodes; higher similarity values indicate a higher probability of connection.

```plaintext
proc network
  links = mycas.LinkSetInWR
  nodesSubset = mycas.NodeSubSetIn;
  nodeSimilarity
    outSimilarity = mycas.NodeSim;
```
run;

To select the best candidate links for investigators, the following steps determine five links that have the highest Jaccard coefficient for each of the nodes in the subset:

```
data NodeSim;
   set mycas.NodeSim;
run;

data LinkSetInWR;
   set mycas.LinkSetInWR;
run;

proc sql;
create table PredictedLinks as select
   a.* from NodeSim as a left join
   LinkSetInWR as b on
   (a.source=b.from and a.sink=b.to) or
   (a.source=b.to and a.sink=b.from)
   where b.from="" and a.source ne a.sink;
quit;
proc sort data=PredictedLinks;
   by source descending jaccard;
run;

data PredictedLinksTop5;
   retain count 0;
   set PredictedLinks;
   by source;
   if first.source then count=0;
   count=count+1;
   if count <= 5;
run;
```

Output 2.16.1 displays the output data table PredictedLinksTop5.
Output 2.16.1  Top Predicted Links between Suspected Criminals

<table>
<thead>
<tr>
<th>count</th>
<th>source</th>
<th>sink</th>
<th>jaccard</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>n118</td>
<td>n36</td>
<td>0.75000</td>
</tr>
<tr>
<td>2</td>
<td>n118</td>
<td>n16</td>
<td>0.66667</td>
</tr>
<tr>
<td>3</td>
<td>n118</td>
<td>n28</td>
<td>0.50000</td>
</tr>
<tr>
<td>4</td>
<td>n118</td>
<td>n103</td>
<td>0.33333</td>
</tr>
<tr>
<td>5</td>
<td>n118</td>
<td>n114</td>
<td>0.33333</td>
</tr>
<tr>
<td>1</td>
<td>n13</td>
<td>n117</td>
<td>0.40000</td>
</tr>
<tr>
<td>2</td>
<td>n13</td>
<td>n89</td>
<td>0.40000</td>
</tr>
<tr>
<td>3</td>
<td>n13</td>
<td>n43</td>
<td>0.25000</td>
</tr>
<tr>
<td>4</td>
<td>n13</td>
<td>n101</td>
<td>0.20000</td>
</tr>
<tr>
<td>5</td>
<td>n13</td>
<td>n104</td>
<td>0.20000</td>
</tr>
<tr>
<td>1</td>
<td>n27</td>
<td>n49</td>
<td>0.08333</td>
</tr>
<tr>
<td>2</td>
<td>n27</td>
<td>n137</td>
<td>0.03571</td>
</tr>
<tr>
<td>3</td>
<td>n27</td>
<td>n141</td>
<td>0.03571</td>
</tr>
<tr>
<td>4</td>
<td>n27</td>
<td>n25</td>
<td>0.03571</td>
</tr>
<tr>
<td>5</td>
<td>n27</td>
<td>n6</td>
<td>0.03571</td>
</tr>
<tr>
<td>1</td>
<td>n49</td>
<td>n35</td>
<td>0.23077</td>
</tr>
<tr>
<td>2</td>
<td>n49</td>
<td>n1</td>
<td>0.18182</td>
</tr>
<tr>
<td>3</td>
<td>n49</td>
<td>n92</td>
<td>0.18182</td>
</tr>
<tr>
<td>4</td>
<td>n49</td>
<td>n84</td>
<td>0.16667</td>
</tr>
<tr>
<td>5</td>
<td>n49</td>
<td>n48</td>
<td>0.14286</td>
</tr>
<tr>
<td>1</td>
<td>n50</td>
<td>n160</td>
<td>0.27273</td>
</tr>
<tr>
<td>2</td>
<td>n50</td>
<td>n76</td>
<td>0.22222</td>
</tr>
<tr>
<td>3</td>
<td>n50</td>
<td>n23</td>
<td>0.20000</td>
</tr>
<tr>
<td>4</td>
<td>n50</td>
<td>n63</td>
<td>0.18182</td>
</tr>
<tr>
<td>5</td>
<td>n50</td>
<td>n26</td>
<td>0.14286</td>
</tr>
</tbody>
</table>

The predicted links are highlighted as bold and colored (by source node) in the illustration in Figure 2.131. Many of these predicted links have additional support from other sources of information, according to Berlusconi et al. (2016). For example, \( n_{49} \) and \( n_{48} \) lived in the same area and had key roles in the drug distribution; the husband of \( n_{118} \) used to buy cocaine from \( n_{36} \); the link between \( n_{13} \) and \( n_{43} \) had been confirmed by further investigation; and so on.
References


Index

ALGORITHM= option
  COMMUNITY statement, 33
  CONNECTEDCOMPONENTS statement, 36
  CYCLE statement, 37
AUTH= option
  CENTRALITY statement, 28
AUXWEIGHT= option
  LINKSVAR statement, 40
BETWEEN= option
  CENTRALITY statement, 28
BETWEENNORM= option
  CENTRALITY statement, 29
BICONNECTEDCOMPONENTS option
  SUMMARY statement, 48
BICONNECTEDCOMPONENTS statement
  statement options, 27
BY statement
  NETWORK procedure, 28
CASESENSITIVE option
  DISPLAY statement (NETWORK), 39
CENTRALITY statement
  statement options, 28
CLIQUE statement
  statement options, 32
CLOSE= option
  CENTRALITY statement, 29
CLOSENOPATH= option
  CENTRALITY statement, 29
CLUSTERINGCOEF option
  CENTRALITY statement, 30
COMMUNITY statement
  statement options, 33
CONNECTEDCOMPONENTS option
  SUMMARY statement, 48
CONNECTEDCOMPONENTS statement
  statement options, 36
CORE statement
  statement options, 36
CYCLE statement
  statement options, 37
DEGREE option
  CENTRALITY statement, 30
DIAMETERAPPROX= option
  SUMMARY statement, 48
DIGRAPH option
  REACH statement, 46
DIRECTION= option
  PROC NETWORK statement, 25
DISPLAY statement
  NETWORK procedure, 38
DISPLAYOUT statement
  NETWORK procedure, 39
DISTRIBUTED option
  PROC NETWORK statement, 25
EACHSOURCE option
  REACH statement, 46
EIGEN= option
  CENTRALITY statement, 30
EIGENALGORITHM= option
  CENTRALITY statement, 30
EIGENMAXITERS= option
  CENTRALITY statement, 30
EXCLUDE option
  DISPLAY statement (NETWORK), 39
EXCLUDEALL option
  DISPLAY statement (NETWORK), 39
FROM= option
  LINKSVAR statement, 40
  LINKSVAR statement, 40
FROMVAR= option
  LINKSVAR statement, 41
HUB= option
  CENTRALITY statement, 30
INCLUDEALL option
  DISPLAYOUT statement (NETWORK), 39
INCLUDESELFLINK option
  PROC NETWORK statement, 25
INDEXOFFSET= option
  PROC NETWORK statement, 25
INFLUENCE= option
  CENTRALITY statement, 31
INTERNALFORMAT= option
  CENTRALITY statement, 31
  COMMUNITY statement, 33
  CONNECTEDCOMPONENTS statement, 36
  NODESIMILARITY statement, 41
  PATH statement, 44
JACCARD= option
  NODESIMILARITY statement, 41
LINKREMOVALRATIO= option
COMMUNITY statement, 33
LINKS= option
PROC NETWORK statement, 25
LINKSQUERY= option
PROC NETWORK statement, 26
LINKSQUERYVAR statement
statement options, 41
LINKSVAR statement
statement options, 40
LOGFREQTIME= option
PROC NETWORK statement, 26
LOGFREQUENCYTIME= option
PROC NETWORK statement, 26
LOGLEVEL= option
PROC NETWORK statement, 26
MAXCLIQUES= option
CLIQUE statement, 32
MAXCYCLES= option
CYCLE statement, 37
MAXITERS= option
COMMUNITY statement, 34
MAXLENGTH= option
CYCLE statement, 37
PATH statement, 44
MAXLINKWEIGHT= option
CLIQUE statement, 32
CYCLE statement, 37
PATH statement, 45
MAXNODEWEIGHT= option
CLIQUE statement, 32
CYCLE statement, 37
PATH statement, 45
MAXPATHWEIGHT= option
SHORTESTPATH statement, 47
MAXREACH= option
REACH statement, 46
MAXSCORE= option
NODESIMILARITY statement, 42
MAXSIZE= option
CLIQUE statement, 32
CYCLE statement, 38
PATH statement, 45
MINCLIQUES= option
CLIQUE statement, 32
CYCLE statement, 38
PATH statement, 45
MINSIZE= option
CLIQUE statement, 32
MINSIZE= option
CLIQUE statement, 32
MODULARITY= option
COMMUNITY statement, 35
NDIMENSIONS= option
NODESIMILARITY statement, 42
NEGATIVESAMPLEFACTOR= option
NODESIMILARITY statement, 42
NETWORK procedure, 16
NETWORK procedure, DISPLAY statement
CASESENSITIVE option, 39
EXCLUDE option, 39
EXCLUDEALL option, 39
TRACE option, 39
NETWORK procedure, DISPLAYOUT statement
INCLUDEALL option, 39
NOREPLACE option, 39
REPEATED option, 40
NODE= option
NODEQUERYVAR statement, 44
NODESUBSETVAR statement, 43
NODESVAR statement, 43
NODES= option
PROC NETWORK statement, 26
NODESIMILARITY statement
statement options, 41
NODESQUERY= option
PROC NETWORK statement, 26
NODESQUERYVAR statement
statement options, 44
NODESSUBSET= option
PROC NETWORK statement, 26
NODESSUBSETVAR statement
statement options, 43
NODESVAR statement
statement options, 43
NOREPLACE option
DISPLAYOUT statement (NETWORK), 39
NSAMPLES= option
NODESIMILARITY statement, 42
NTHREADS= option
PROC NETWORK statement, 27
OUT= option
CLIQUE statement, 33
CYCLE statement, 38
SHORTESTPATH statement, 47
SUMMARY statement, 48
Index ♦ 215

TRANSITIVECLOSURE statement, 49
OUTCOMMLINKS= option 
COMMUNITY statement, 34
OUTCOMMUNITY= option 
COMMUNITY statement, 34
OUTCOUNTS= option 
REACH statement, 46
OUTLEVEL= option 
COMMUNITY statement, 34
OUTLINKS= option 
PROC NETWORK statement, 27
OUTMATCHLINKS= option 
PATTERNMATCH statement, 46
OUTMATCHNODES= option 
PATTERNMATCH statement, 46
OUTNODES= option 
PROC NETWORK statement, 27
OUTOVERLAP= option 
COMMUNITY statement, 34
OUTPATHS= option 
PATH statement, 45
OUTPATHSLINKS= option 
PATH statement, 45
OUTPATHSNODES= option 
PATH statement, 45
OUTREACHLINKS= option 
REACH statement, 47
OUTREACHNODES= option 
REACH statement, 47
OUTSIMILARITY= option 
NODESIMILARITY statement, 42
OUTWEIGHTS= option 
SHORTESTPATH statement, 47
PAGERANK= option 
CENTRALITY statement, 31
PAGERANKALPHA= option 
CENTRALITY statement, 31
PAGERANKTOL= option 
CENTRALITY statement, 31
PATH statement 
statement options, 44
PATTERNMATCH statement 
statement options, 46
PROC NETWORK statement 
statement options, 25
PROXIMITYORDER= option 
NODESIMILARITY statement, 42
RANDOMFACTOR= option 
COMMUNITY statement, 34
RANDOMSEED= option 
COMMUNITY statement, 34

REACH statement 
statement options, 46
REACH= option 
NODESUBSETVAR statement, 43
RECURSIVE (options) 
COMMUNITY statement, 34
REPEATED option 
DISPLAYOUT statement (NETWORK), 40
RESOLUTIONLIST= option 
COMMUNITY statement, 35
SHORTESTPATH statement 
statement options, 47
SHORTESTPATH= option 
SUMMARY statement, 48
SINK= option 
NODESIMILARITY statement, 42
NODESUBSETVAR statement, 43
PATH statement, 45
SHORTESTPATH statement, 47
SOURCE= option 
NODESIMILARITY statement, 42
NODESUBSETVAR statement, 43
PATH statement, 45
SHORTESTPATH statement, 47
STANDARDIZEDLABELS option 
PROC NETWORK statement, 27
SUMMARY statement 
statement options, 48
TIMETYPE= option 
PROC NETWORK statement, 27
TO= option 
LINKSQUERYVAR statement, 41
LINKSVAR statement, 40
TOLERANCE= option 
COMMUNITY statement, 35
TOVAR= option 
LINKSQUERYVAR statement, 41
TRACE option 
DISPLAY statement (NETWORK), 39
TRANSITIVECLOSURE statement 
statement options, 49
VARS= option 
LINKSQUERYVAR statement, 41
LINKSVAR statement, 40
NODEQUERYVAR statement, 44
NODESVAR statement, 43
VECTOR= option 
NODESIMILARITY statement, 42
WEIGHT= option 
LINKSVAR statement, 40
NODESVAR statement, 43
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.