SAS® Visual Data Mining and Machine Learning 8.1 The NETWORK Procedure

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2016. *SAS® Visual Data Mining and Machine Learning 8.1: The NETWORK Procedure*. Cary, NC: SAS Institute Inc.

SAS® Visual Data Mining and Machine Learning 8.1: The NETWORK Procedure

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

September 2016

Contents

Credits

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
Technical Support	Tonya Chapman

Chapter 1
Introduction

Contents

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.

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.

About This Book

This book assumes that you are familiar with Base SAS software and with the books *SAS Language Reference: Concepts* and *Base SAS 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.

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:

```
ods html style=HTMLBlue;
. . .
ods html close;

ods pdf style=PearlJ;
. . .
ods pdf close;
```

Most of the PDF tables are produced by using the following SAS System option:

```
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`.

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` for more information.

Chapter 2
The NETWORK Procedure

Contents

Overview: NETWORK Procedure

In order to support the myriad ways that 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

Algorithm Class	PROC NETWORK Statement
Biconnected components	BICONNECTEDCOMPONENTS
Centrality	CENTRALITY
Clique enumeration	CLIQUE
Community detection	COMMUNITY
Connected components	CONNECTEDCOMPONENTS
Core decomposition	CORE
Cycle enumeration	CYCLE
Reach (ego) networks	REACH
Shortest path	SHORTESTPATH
Graph summary	SUMMARY
Transitive closure	TRANSITIVECLOSURE

The NETWORK procedure expects as input *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.

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:

```
cas mysess terminate;
```

For more information about the CAS statement and the LIBNAME statement, see *SAS Cloud Analytic Services: Language Reference*. For general information about CAS and CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*.

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:

```
data mycas.Sample;
   input y x @@;
   datalines;
.46 1 .47 2 .57 3 .61 4 .62 5 .68 6 .69 7
;
```

  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:

```
data Sample;
   input y x @@;
   datalines;
.46 1 .47 2 .57 3 .61 4 .62 5 .68 6 .69 7 .78 8
;
data mycas.Sample;
   set Sample;
run;
```

- You can use the CASUTIL procedure as follows:

```
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:

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

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

```
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: Accessing and Manipulating Data*.

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 input formats expected and the various algorithms available, see the sections "Details: NETWORK Procedure" on page 39 and "Examples: NETWORK Procedure" on page 124.

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:

```
data mycas.LinkSetInRoadNC10am;
   input start_inter $1-20 end_inter $20-40 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:

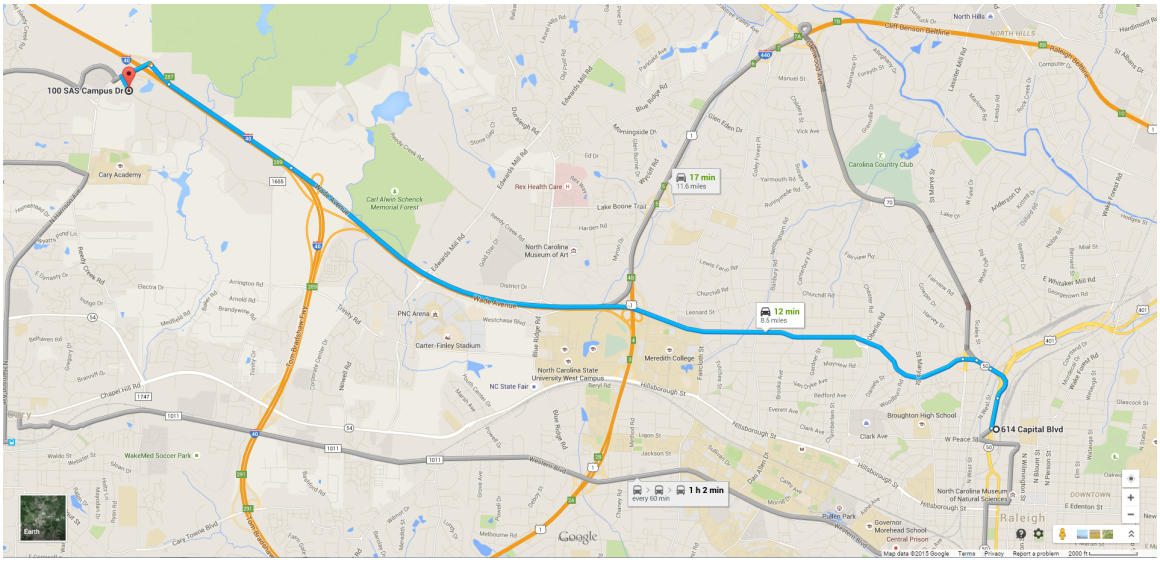
```
proc network
   links       = mycas.LinkSetInRoadNC10am;
   linksVar
      from     = start_inter
      to       = end_inter
      weight   = time_to_travel;
   shortestPath
      outPaths = mycas.ShortPath
      source   = "614CapitalBlvd"
      sink     = "SASCampusDrive";
run;
```

For more information about shortest path algorithms in PROC NETWORK, see the section "Shortest Path" on page 101. 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.

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

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

Figure 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 $20-40 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;
```

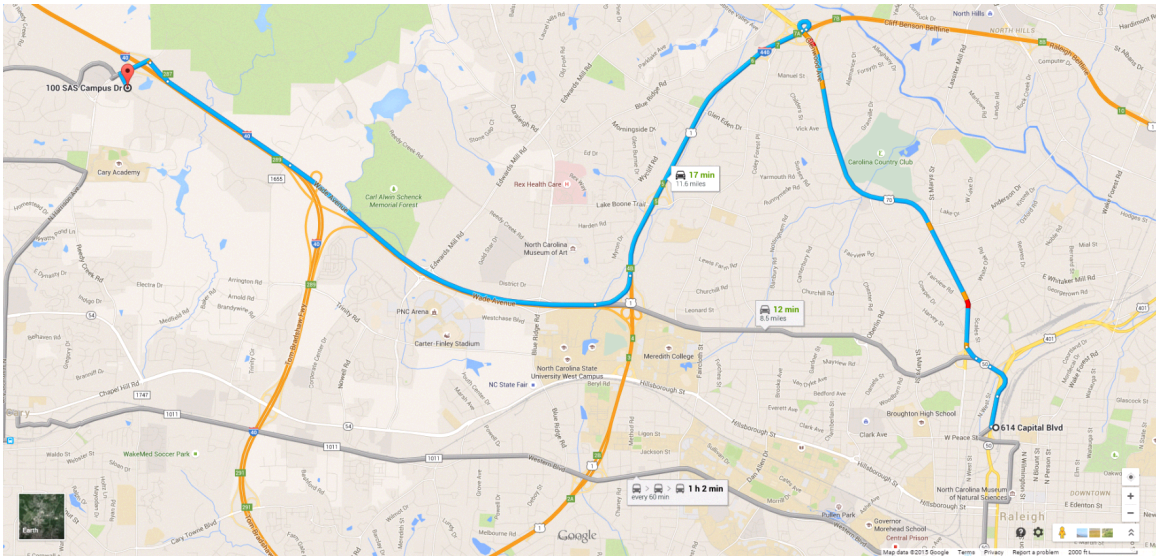
Now, the output data table mycas.ShortPath, shown in Figure 2.3, shows the best route for going home. Because the traffic on Wade Avenue is usually heavy at this time of day, the route home is different from the route to work.

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

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

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

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



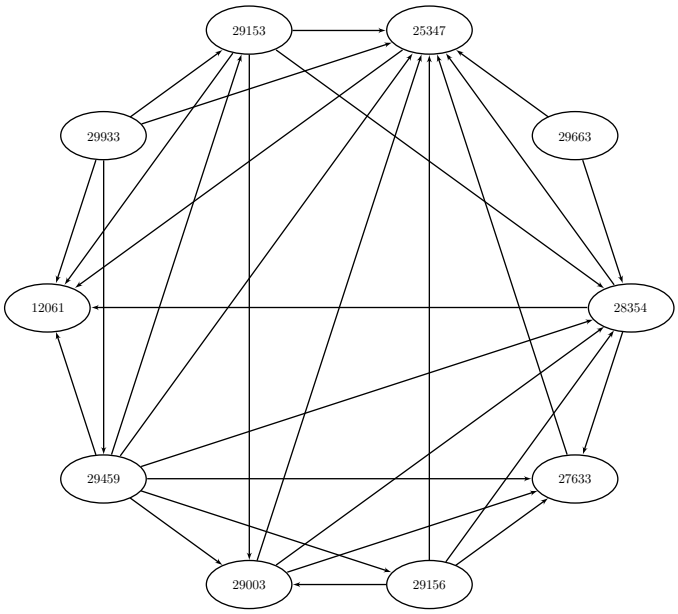
Authority in US Supreme Court Precedent

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 66. Figure 2.5 shows a small representative subset of the citation network for landmark abortion decisions from the example in Fowler and Joen (2008).

Figure 2.5 Citation Network for Some US Supreme Court Cases



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

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

The data table mycas.LinkSetInCourt provides the citation network between case identifiers:

```
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
;
```

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

```
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:

```
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

case_id	centr_auth_unwt	case_name
25347	1.00000	Roe vs. Wade, 410 U.S. 113 (1973)
28354	0.72262	Thornburgh vs. American College, 476 U.S. 747 (1986)
12061	0.61717	Jacobson v. Massachusetts, 197 U.S. 11 (1905)
27633	0.59831	Akron vs. Akron Cntr for Repro-Health, 462 U.S. 416 (1983)
29003	0.50930	Webster vs. Repro-Health Services, 492 U.S. 490 (1989)
29153	0.31742	Cruzan v. Director, MO Dept of Health, 497 U.S. 261 (1990)
29156	0.20968	Hodgson v. Minnesota, 497 U.S. 417 (1990)
29459	0.10775	Planned Parenthood of SE PA vs. Casey, 505 U.S. 833 (1992)
29663	0.00000	Madsen v. Women's Health Ctr., 512 U.S. 753 (1994)
29933	0.00000	Wash. v. Glucksberg, 521 U.S. 702 (1997)
29155	.	Georgia v. South Carolina, 497 U.S. 376 (1990)

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 >* **;**

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

Data Input Statements:

LINKSVAR *< options >* **;**
NODESVAR *< options >* **;**
NODESSUBSETVAR *< options >* **;**

These 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 >* **;**
REACH *< options >* **;**
SHORTESTPATH *< options >* **;**
SUMMARY *< options >* **;**
TRANSITIVECLOSURE *< options >* **;**

These statements determine which algorithm is run and set options for each individual algorithm.

Standard Statements:

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

These 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).

Functional Summary

Table 2.2 summarizes the statements and options available in the NETWORK procedure.

Table 2.2 Functional Summary of Statements and Options

Description	Statement	Option
Input		
Specifies the links data table	PROC NETWORK	LINKS=
Specifies the nodes data table	PROC NETWORK	NODES=
Specifies the nodes subset data table	PROC NETWORK	NODESSUBSET=
Output		
Specifies the links output data table	PROC NETWORK	OUTLINKS=
Specifies the nodes output data table	PROC NETWORK	OUTNODES=
Options		
Specifies the graph direction	PROC NETWORK	DIRECTION=
Includes self-links	PROC NETWORK	INCLUDESELFLINK
Specifies the index offset for identifiers	PROC NETWORK	INDEXOFFSET=
Specifies the desired frequency (in number of seconds) between log entries	PROC NETWORK	LOGFREQTIME=
Specifies the overall log level	PROC NETWORK	LOGLEVEL=
Specifies the maximum number of threads to use for multithreaded processing	PROC NETWORK	NTHREADS=
Specifies that the input graph data are in a standardized format	PROC NETWORK	STANDARDIZEDLABELS
Specifies whether time units are in CPU time or real time	PROC NETWORK	TIMETYPE=
Data Input Statements		
Specifies the data variable name for the auxiliary link weights	LINKSVAR	AUXWEIGHT=
Specifies the data variable name for the *from* nodes	LINKSVAR	FROM=

Table 2.2 (continued)

Description	Statement	Option
Specifies the data variable name for the *to* nodes	LINKSVAR	TO=
Specifies the data variable name for the link weights	LINKSVAR	WEIGHT=
Specifies the data variable name for the nodes	NODESVAR	NODE=
Specifies the data variable name for node weights	NODESVAR	WEIGHT=
Specifies the data variable name for the nodes	NODESSUBSETVAR	NODE=
Specifies the data variable name for the reach identifier	NODESSUBSETVAR	REACH=
Specifies the data variable name for the sink indicator	NODESSUBSETVAR	SINK=
Specifies the data variable name for the source indicator	NODESSUBSETVAR	SOURCE=
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
Specifies which type of eigenvector centrality to calculate	CENTRALITY	EIGEN=
Specifies the algorithm to use for eigenvector calculation	CENTRALITY	EIGENALGORITHM=
Specifies the maximum number of iterations for eigenvector calculation	CENTRALITY	EIGENMAXITERS=
Specifies which type of hub centrality to calculate	CENTRALITY	HUB=
Specifies which type of influence centrality to calculate	CENTRALITY	INFLUENCE=

Table 2.2 (continued)

Description	Statement	Option
CLIQUE Statement		
Specifies the maximum number of cliques to return during clique enumeration	CLIQUE	MAXCLIQUES=
Specifies the maximum amount of time to spend finding cliques	CLIQUE	MAXTIME=
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=
Specifies the maximum number of iterations for community detection	COMMUNITY	MAXITERS=
Specifies the output data table for intercommunity links	COMMUNITY	OUTCOMMLINKS=
Specifies the output data table for the community summary	COMMUNITY	OUTCOMMUNITY=
Specifies the output data table for the community level summary	COMMUNITY	OUTLEVEL=
Specifies the output data table for the community overlap	COMMUNITY	OUTOVERLAP=
Specifies the random factor in the parallel label propagation algorithm	COMMUNITY	RANDOMFACTOR=
Specifies the random seed for the parallel label propagation algorithm	COMMUNITY	RANDOMSEED=
Applies the recursive option to break large communities	COMMUNITY	RECURSIVE
Specifies the resolution list for community detection	COMMUNITY	RESOLUTIONLIST=
Specifies the modularity tolerance value for community detection	COMMUNITY	TOLERANCE=
CONNECTEDCOMPONENTS Statement		
Specifies the algorithm to use for connected components	CONNECTEDCOMPONENTS	ALGORITHM=
Specifies the internal graph format	CONNECTEDCOMPONENTS	INTERNALFORMAT=
CORE Statement		
Specifies the maximum amount of time to spend calculating the core decomposition	CORE	MAXTIME=
CYCLE Statement		
Specifies the algorithm to use for cycle enumeration	CYCLE	ALGORITHM=

Table 2.2 (continued)

Description	Statement	Option
Specifies the maximum number of cycles to return during cycle enumeration	CYCLE	MAXCYCLES=
Specifies the maximum length for the cycles found	CYCLE	MAXLENGTH=
Specifies the maximum link weight for the cycles found	CYCLE	MAXLINKWEIGHT=
Specifies the maximum node weight for the cycles found	CYCLE	MAXNODEWEIGHT=
Specifies the maximum amount of time to spend finding cycles	CYCLE	MAXTIME=
Specifies the minimum length for the cycles found	CYCLE	MINLENGTH=
Specifies the minimum link weight for the cycles found	CYCLE	MINLINKWEIGHT=
Specifies the minimum node weight for the cycles found	CYCLE	MINNODEWEIGHT=
Specifies the output data table for cycles	CYCLE	OUT=
REACH Statement		
Calculates the directed reach counts	REACH	DIGRAPH
Treats each node as a source in reach calculations	REACH	EACHSOURCE
Specifies the maximum number of links in the reach calculations	REACH	MAXREACH=
Specifies the output data table for reach counts	REACH	OUTCOUNTS=
Specifies the output data table for reach links	REACH	OUTREACHLINKS=
Specifies the output data table for reach nodes	REACH	OUTREACHNODES=
SHORTESTPATH Statement		
Specifies the maximum path weight	SHORTESTPATH	MAXPATHWEIGHT=
Specifies the output data table for shortest paths	SHORTESTPATH	OUTPATHS=
Specifies the output data table for shortest path summaries	SHORTESTPATH	OUTWEIGHTS=
Specifies the sink node for shortest paths calculations	SHORTESTPATH	SINK=
Specifies the source node for shortest paths calculations	SHORTESTPATH	SOURCE=
SUMMARY Statement		
Calculates information about biconnected components	SUMMARY	BICONNECTEDCOMPONENTS

Table 2.2 (continued)

Description	Statement	Option
Calculates information about connected components	SUMMARY	CONNECTEDCOMPONENTS
Calculates the approximate diameter and chooses the weight type	SUMMARY	DIAMETERAPPROX=
Specifies the output data table for summary results	SUMMARY	OUT=
Calculates information about shortest paths and chooses the weight type	SUMMARY	SHORTESTPATH=
TRANSITIVECLOSURE Statement		
Specifies the output data table for transitive closure results	TRANSITIVECLOSURE	OUT=

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

Table 2.3 Supported Input Formats by Statement

	DIRECTION	
Statement	UNDIRECTED	DIRECTED
BICONNECTEDCOMPONENTS	X	
CENTRALITY		
AUTH=, HUB=		X
BETWEEN=, CLOSE=, CLUSTERINGCOEF, DEGREE=, EIGEN=, INFLUENCE=,	X	X
CLIQUE	X	
COMMUNITY		
ALGORITHM=		
LOUVAIN, LABELPROP	X	
PARALLELLABELPROP	X	X
CONNECTEDCOMPONENTS		
ALGORITHM=		
DFS	X	X
PARALLEL, UNIONFIND	X	
CORE	X	X
CYCLE	X	X
REACH	X	X
SHORTESTPATH	X	X
SUMMARY	X	X
TRANSITIVECLOSURE	X	X

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

Statement	OUTNODES	OUTLINKS	Algorithm Statement Options
BICONNECTEDCOMPONENTS	X	X	
CENTRALITY AUTH=, CLOSE=, CLUSTERINGCOEF, DEGREE=, EIGEN=, HUB=, INFLUENCE= BETWEEN=	X X	 X	
CLIQUE			OUT=
COMMUNITY	X	X	OUTCOMMLINKS=, OUTCOMMUNITY=, OUTLEVEL=, OUTOVERLAP=
CONNECTEDCOMPONENTS	X	X	
CORE	X		
CYCLE			OUT=
REACH	X		OUTCOUNTS=, OUTREACHLINKS=, OUTREACHNODES=
SHORTESTPATH			OUTPATHS=, OUTWEIGHTS=
SUMMARY	X	X	OUT=
TRANSITIVECLOSURE			OUT=

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 (for example, information) can flow over that link. In link (i, j), the flow is from node i to node j ($i \rightarrow j$). The node i is called the *source* (*tail*) node, and the 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 39.

INCLUDESELFLINK
includes self-links—for example, (i, i)—when an input graph is read. By default, when PROC NETWORK reads the LINKS= 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 would be labeled cycles 1, 2, and 3 by default. If INDEXOFFSET=4, they would be 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 7.

For more information about this input table, see the section "Links Input Data" on page 40.

LOGFREQTIME=*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.

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 input, output, and 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.

NODES=*CAS-libref.data-table*
specifies the input data table that contains the graph node 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 7.

For more information about this input table, see the section "Nodes Input Data" on page 43.

NODESSUBSET=*CAS-libref.data-table*
specifies the input data table that contains the graph node subset 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 7.

For more information about this input table, see the section "Nodes Subset Input Data" on page 44.

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. Certain algorithms cannot take advantage of this option and use only one thread even if *number* > 1. The algorithms that can take advantage of multiple cores are listed in Table 2.6. 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 7.

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 7.

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 45.

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 51.

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. An example of this is shown in the section "Example 2.8: Centrality Metrics for a Simple Undirected Graph by Community" on page 145.

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 55.

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 based on the weighted graph.
UNWEIGHT	calculates authority centrality based on the unweighted graph.
BOTH	calculates authority centrality based on both weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (using 1.0 for each link weight). This centrality metric can be used only for directed graphs. For more

information about the authority centrality metric, see the section "Hub and Authority Scoring" on page 66.

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 based on the weighted graph.

UNWEIGHT calculates betweenness centrality based on the unweighted graph.

BOTH calculates betweenness centrality based on both weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (using 1.0 for each link weight). If the 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 62.

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 62. 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 based on the weighted graph.

UNWEIGHT calculates closeness centrality based on the unweighted graph.

BOTH calculates closeness centrality based on both weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (using 1.0 for each link weight). For more information about the closeness centrality metric, see the section "Closeness Centrality" on page 59.

CLOSENOPATH=NNODES | DIAMETER | ZERO | HARMONIC
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:

NNODES uses the number of nodes as the shortest path distance between disconnected nodes. You cannot specify this option when CLOSE=WEIGHT or CLOSE=BOTH.

DIAMETER uses the graph diameter (plus one) as the shortest path distance between disconnected nodes.

ZERO uses zero as the shortest path distance between disconnected nodes.

HARMONIC uses the harmonic formula for calculating closeness centrality.

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 59. By default, CLOSENOPATH=DIAMETER.

CLUSTERINGCOEF
calculates the node clustering coefficient. For more information about the clustering coefficient, see the section "Clustering Coefficient" on page 57.

DEGREE
calculates the degree centrality. For more information about the degree centrality metric, see the section "Degree Centrality" on page 55.

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

WEIGHT calculates eigenvector centrality based on the weighted graph.

UNWEIGHT calculates eigenvector centrality based on the unweighted graph.

BOTH calculates eigenvector centrality based on both weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (using 1.0 for each link weight). For more information about the eigenvector centrality metric, see the section "Eigenvector Centrality" on page 64.

EIGENALGORITHM=AUTOMATIC | JACOBIDAVIDSON | POWER
specifies the algorithm to use in calculating centrality metrics that require solving eigensystems—that is, when the EIGEN, 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 is used as the default for the eigenvector metric on undirected graphs and the hub and authority metrics.

POWER uses the power method to calculate eigenvectors. This is used as the default for the eigenvector metric on directed graphs.

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 based on the weighted graph.

UNWEIGHT calculates hub centrality based on the unweighted graph.

BOTH calculates hub centrality based on both weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (using 1.0 for each link weight). This centrality metric can be used only for directed graphs. For more information about the hub centrality metric, see the section "Hub and Authority Scoring" on page 66.

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

WEIGHT calculates influence centrality based on the weighted graph.

UNWEIGHT calculates influence centrality based on the unweighted graph.

BOTH calculates influence centrality based on both weighted and unweighted graphs.

If the input graph does not contain weights, then WEIGHT and UNWEIGHT both give the same results (using 1.0 for each link or node weight). For more information about the influence centrality metric, see the section "Influence Centrality" on page 56.

CLIQUE Statement

CLIQUE < *options* > **;**

The CLIQUE 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 68.

You can specify the following *options*:

MAXCLIQUES=*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 you can specify ALL (which represents the maximum that can be represented by a 32-bit integer). By default, MAXCLIQUES=1.

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 value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

OUT=*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 7.

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 71.

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 algorithm is experimental in this release.

By default, ALGORITHM=LOUVAIN.

INTERNALFORMAT=FULL | THIN
specifies which 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.

For more information, see the section "Graph Input Data" on page 39.

By default, INTERNALFORMAT=THIN.

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 $A \rightarrow B$ has weight of 100, and link $A \rightarrow C$ has weight of 1. When nodes are grouped into communities, link $A \rightarrow B$ is much more important than link $A \rightarrow C$ because it contributes much more to the overall modularity value. Therefore, link $A \rightarrow C$ can be dropped from the network if dropping it does not disconnect node C from the network. If you specify this option, then the links that are incident to each node are examined. If the weight of any link is less than (*number*/100)**max_link_weight*, where *max_link_weight* is the maximum link weight among all links incident to this node, 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, LINKREMOVALRATIO=10.

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

OUTCOMMLINKS=*CAS-libref.data-table*
specifies the output data table that describes the links between communities. *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 7.

OUTCOMMUNITY=*CAS-libref.data-table*
specifies the output data table that contains the number of nodes in each community. *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 7.

OUTLEVEL=*CAS-libref.data-table*
specifies the output data table that contains community information at different resolution levels. *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 7.

OUTOVERLAP=*CAS-libref.data-table*
specifies the output data table that describes the intensity of each node. *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 7.

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

RANDOMSEED=*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 RANDOMFACTOR= option. To choose a different set of random samples, specify a *number* in the RANDOMSEED= option. By default, RANDOMSEED=1234.

RECURSIVE (*options***)**
breaks down large communities into smaller ones until the specified conditions are satisfied. This option starts with the keyword RECURSIVE followed by any combination of three suboptions enclosed in parentheses—for example, RECURSIVE (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 positive number that has the largest absolute value 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 positive

number that has the largest absolute value 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 74.

TOLERANCE=*number*

MODULARITY=*number*

specifies the tolerance value for when to stop iterations. When you specify ALGORITHM=LOUVAIN, the algorithm stops iterations when the percentage modularity gain between two consecutive iterations 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=0.01.

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 80. 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. You cannot specify this value when you specify INTERNALFORMAT=THIN.

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.

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 which 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).

For more information, see the section "Graph Input Data" on page 39.

By default, INTERNALFORMAT=FULL when ALGORITHM=DFS or INTERNALFORMAT=THIN for any other value of the ALGORITHM= option.

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 84.

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 value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

CYCLE Statement

CYCLE < *options* > **;**

The CYCLE statement invokes an algorithm that finds the cycles (or the existence of a cycle) in the input graph. For more information about cycles, see the section "Cycle Enumeration" on page 89.

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).

BUILD uses a building algorithm based on Liu and Wang (2006).

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 you can specify 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 positive number that has the largest absolute value 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 positive number that has the largest absolute value that can be represented in your operating environment, 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 positive number that has the largest absolute value that can be represented in your operating environment, 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 value of *number* can be any positive number; the default value is the positive number that has the largest absolute value that can be represented in your operating environment.

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 negative number that has the largest absolute value that can be represented in your operating environment, 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 negative number that has the largest absolute value that can be represented in your operating environment, 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 7.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of ODS 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, all ODS tables are sent to the client and then the client creates a subset. If both DISPLAY and ODS statements are used together, the DISPLAY statement takes precedence over the ODS statements. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

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

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to ODS 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 ODS tables except those specified in the *table-list*.

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

TRACE
displays the ODS table names, labels, and paths.

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

A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that is produced by a procedure during a selection routine might have the path

Bygroup1.Summary.SelectionSummary. A partial pathname does not include all groups; for example, *Selection-Summary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all ODS 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 starts with a "/" or a "!". 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.

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* or a *key=value* 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.

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 (/):

NOREPLACE
does not replace an existing CAS output table of the same name.

REPEATED
replicates the CAS output tables on all nodes.

The output tables produced by the NETWORK procedure when using the DISPLAYOUT statement are a transposed version of the displayed tables. This allows for easier post-analysis, especially when used together with BY-group processing. An example of using the DISPLAYOUT statement is shown in the section "Example 2.8: Centrality Metrics for a Simple Undirected Graph by Community" on page 145.

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 is specified 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 40.

You can specify the following *options*:

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

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

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

WEIGHT=*column*
specifies the data variable name for the link weights. 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 is specified 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 43.

You can specify the following *options*:

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

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

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 is specified 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 43.

You can specify the following *options*:

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

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

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

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

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 94.

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 7.

OUTREACHLINKS=*CAS-libref.data-table*
specifies the output data table to contain the links 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 7.

OUTREACHNODES=*CAS-libref.data-table*
specifies the output data table to contain the nodes 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 7.

SHORTESTPATH Statement

SHORTESTPATH < *options* > **;**

The 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 101.

You can specify the following *options*:

MAXPATHWEIGHT=*number*
specifies the maximum path weight. Any shortest path whose sum of link weights is greater than *number* is removed from the results. The default is the positive number that has the largest absolute value that can be represented in your operating environment, which causes no paths to be removed from the results.

OUTPATHS=*CAS-libref.data-table*

OUT=*CAS-libref.data-table*
specifies the output data table to contain the shortest paths. *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 7.

OUTWEIGHTS=*CAS-libref.data-table*
specifies the output data table to contain the shortest path summaries. *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 7.

SINK=*sink-node*
specifies the sink node for shortest paths calculations. This setting overrides the use of the variable sink in the data table that is specified in the NODESSUBSET= option in the PROC NETWORK statement.

SOURCE=*source-node*
specifies the source node for shortest paths calculations. This setting overrides the use of the variable source in the data table that is specified 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 112.

You can specify the following *options*:

BICONNECTEDCOMPONENTS
calculates information about biconnected components. The graph must be undirected to use this option.

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 expensive. You can specify the following values:

WEIGHT	calculates the approximate diameter based on the weighted graph.
UNWEIGHT	calculates the approximate diameter based on the unweighted graph.
BOTH	calculates the approximate diameter based on both weighted and unweighted graphs.

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

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 7.

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 based on the weighted graph.
UNWEIGHT	calculates shortest paths based on the unweighted graph.
BOTH	calculates shortest paths based on both weighted and unweighted graphs.

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

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 119.

You can specify the following *option*:

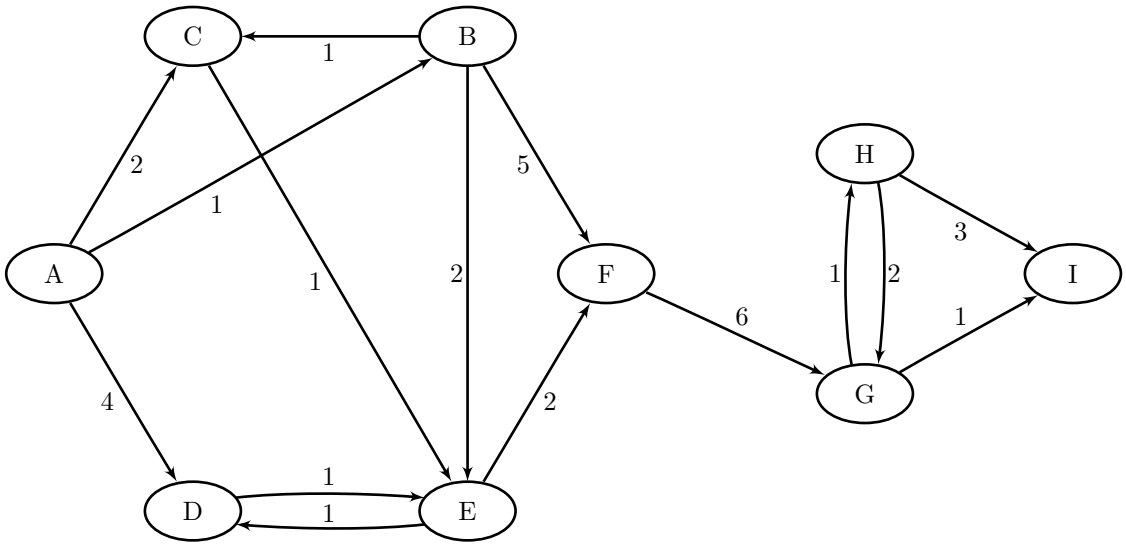
OUT=*CAS-libref.data-table*
specifies the output data table to contain the transitive closure results. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section "Using CAS Sessions and CAS Engine Librefs" on page 7.

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 some combination of the following possible variables:

- 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 call for 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, you must identify the variables by using the LINKSVAR statement, as described in the section "LINKSVAR Statement" on page 34.

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

```
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:

```
proc network
   links = mycas.LinkSetInA;
run;

proc network
   links     = mycas.LinkSetInB;
   linksVar
      from   = source_node
      to     = sink_node
      weight = value;
run;
```

The directed graph *G* shown in Figure 2.7 can be represented by the following links data table, mycas.LinkSetIn:

```
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

node
A
B
C
D
E
F
G
H
I

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.

Figure 2.9 Links Data Table of a Simple Directed Graph

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

If you define this graph as undirected, then reciprocal links (for example, $D \rightarrow E$ and $D \leftarrow E$) are treated as the same link, and duplicates are removed. PROC NETWORK aggregates the attributes of each duplicate link by taking the minimum value (for each attribute). By default, DIRECTION=UNDIRECTED, so you can just 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:

```
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 PROC NETWORK Log: Links Data Table of a Simple Undirected Graph

```
NOTE: ---------------------------------------------------------------------------------------
NOTE: Running NETWORK.
NOTE: ---------------------------------------------------------------------------------------
WARNING: The graph contains 2 duplicate links that are ignored.
NOTE: The number of nodes in the input graph is 9.
NOTE: The number of links in the input graph is 13.
NOTE: The Cloud Analytic Services server processed the request in 0.367296 seconds.
NOTE: The data set MYCAS.NODESETOUT has 9 observations and 1 variables.
NOTE: The data set MYCAS.LINKSETOUT has 13 observations and 3 variables.
```

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.

Figure 2.11 Links Data Table of a Simple Undirected Graph

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

Thin Internal Format

Certain algorithms can perform more efficiently when you specify INTERNALFORMAT=THIN in their respective algorithmic statement. However, when you specify this option, PROC NETWORK does not remove duplicate links. Instead, you should use 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 some combination of the following possible variables:

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

You can specify any value that you want for the data table variable name. If you use a nonstandard name, you must identify the variable by using the NODESVAR statement, as described in the section "NODESVAR Statement" on page 35.

The data table that is specified 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 this node must be defined in the NODES= data table. The following statements produce a graph that has three links but four nodes, including a singleton node D:

```
data mycas.NodeSetIn;
   input label $ @@;
   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 SHORTESTPATH or REACH statement. (See the sections "Shortest Path" on page 101 and "Reach (Ego) Network" on page 94, 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 (must be numeric)
- sink: whether to process this node as a sink node in shortest path algorithms (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

Algorithm Type	Variable Designations	Example Shown In:
Shortest path	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.	The section "Shortest Path" on page 101

Table 2.5 (continued)

Algorithm Type	Variable Designations	Example Shown In:
Reach	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.	The section "Reach (Ego) Network" on page 94

The following example of a nodes subset data table might be used with the graph in Figure 2.7:

```
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 for the source-sink pairs in $\{A, E\} \times \{F\}$ (the crossproduct of subsets $\{A, E\}$ and $\{F\}$)

Standardized Labels

For large-scale graphs, the processing stage that reads the nodes and links into memory can be time-consuming. Under the following assumptions, you can use the 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:

```
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:

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

The log is shown in Figure 2.12.

Figure 2.12 PROC NETWORK Log: A Simple Undirected Graph

```
NOTE: ------------------------------------------------------------------------------------------
NOTE: Running NETWORK.
NOTE: ------------------------------------------------------------------------------------------
NOTE: The number of nodes in the input graph is 4.
NOTE: The number of links in the input graph is 3.
NOTE: The Cloud Analytic Services server processed the request in 0.245527 seconds.
NOTE: The data set MYCAS.NODESETOUT has 4 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 2.13, contains the unique numeric node labels, $\{0, 1, 3, 5\}$.

Figure 2.13 Nodes Data Table of a Simple Directed Graph

Obs	node
1	0
2	1
3	3
4	5

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

```
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 2.14.

Figure 2.14 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 number of singleton nodes in the input graph is 2.
NOTE: The Cloud Analytic Services server processed the request in 0.240239 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 2.15, now contains all node labels from 0 to 5, based on the assumptions when you use the STANDARDIZEDLABELS option.

Figure 2.15 Nodes Data Table of a Simple Directed Graph

Obs	node
1	0
2	1
3	2
4	3
5	4
6	5

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 the algorithm that you select.

For a single-machine CAS server, there is no data movement. The algorithm runs on the same machine where the data are present. 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 and merges results at the end of the processing. The resulting output tables end up being distributed across the grid.

Some other algorithms use multiple machines and require only a portion of the data. However, because the original data are typically 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 end up being distributed across the grid.

In addition, on each machine, some of these algorithms (as well as the input phase) take advantage of multicore chip technology by executing 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 result in reduced performance. Specifying a high *number* does not guarantee shorter solution time; the actual change in solution time depends on the computing hardware and the scalability of the underlying algorithms. In 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 the same 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 setting for the NTHREADS= option exceeds the number of groups assigned to the processing machine. In this case, the resulting output tables end up being distributed across the grid (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 2.6 Execution Modes and Data Movement

Statement (and Options)	Data Movement	Processing Mode
BICONNECTEDCOMPONENTS	Moved to SM	SM
CENTRALITY		
BETWEEN=, CLOSE=	Repeated on MM	MM (MT)
CLUSTERINGCOEF	Moved to SM	SM (MT)
AUTH=, HUB=, DEGREE=, EIGEN=, INFLUENCE=	Moved to SM	SM
CLIQUE	Moved to SM	SM
COMMUNITY ALGORITHM=		
LOUVAIN, LABELPROP	Moved to SM	SM
PARALLELLABELPROP	Shuffled across MM	MM (MT)
CONNECTEDCOMPONENTS ALGORITHM=		
DFS, UNIONFIND	Moved to SM	SM
PARALLEL	Shuffled across MM	MM
CORE	Moved to SM	SM
CYCLE ALGORITHM=		
BACKTRACK	Moved to SM	SM
BUILD	Moved to SM	SM (MT)

Table 2.6 (continued)

Statement (and Options)	Data Movement	Processing Mode
REACH	Repeated on MM	MM (MT)
SHORTESTPATH	Repeated on MM	MM (MT)
SUMMARY (other than shortest path)	Moved to SM	SM
SHORTESTPATH=	Moved to SM	SM (MT)
TRANSITIVECLOSURE	Moved to SM	SM

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 absolute value representable in your operating environment.

Table 2.7 Threshold Limits by Statement

	Graph Links		Graph Nodes
Statement (and Options)	weight	auxweight	weight
CENTRALITY			
AUTH=, EIGEN=, HUB=	1E20		
BETWEEN=, CLOSE=	$\sqrt{M}$	$\sqrt{M}$	
INFLUENCE=	$\sqrt{M}$		$\sqrt{M}$
COMMUNITY	$\sqrt{M}$		
CYCLE	$\sqrt{M}$		$\sqrt{M}$
REACH			$\sqrt{M}$
SHORTESTPATH	$\sqrt{M}$	$\sqrt{M}$	
SUMMARY			
DIAMETERAPPROX=, SHORTESTPATH=	$\sqrt{M}$		

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:

```
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 a negative link weight is provided:

- 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 can be represented by a 32-bit integer). 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 73.

If the data from your problem require a graph that contains more than two billion nodes (or links), there is typically a heuristic way to break the network into smaller networks based on problem-specific attributes. Then, using DATA steps (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 2.8 Formulas for Number of Links in $K(N)$

Graph Direction	Default	INCLUDESELFLINK
Directed	$\|N\|^2 - \|N\|$	$\|N\|^2$
Undirected	$\frac{\|N\|^2-\|N\|}{2}$	$\frac{\|N\|^2+\|N\|}{2}$

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.

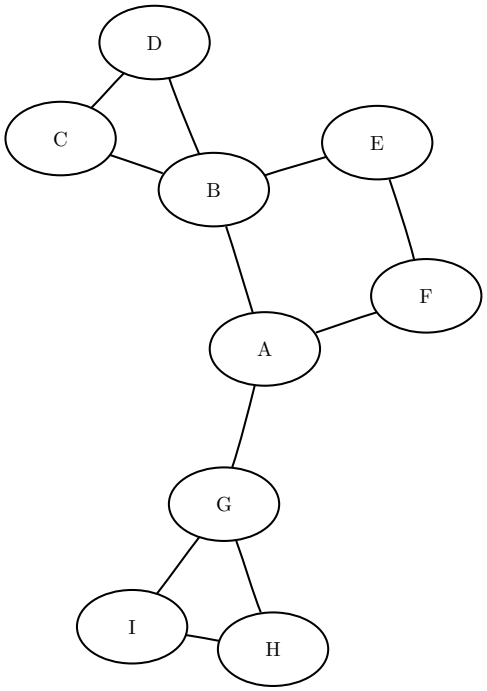
The results of the biconnected components algorithm are written to the output links data table that is specified in the OUTLINKS= option in the PROC NETWORK statement. For each link in the links data table, the variable biconcomp identifies its component. The component identifiers are numbered sequentially, 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 is specified 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 depth-first search (Tarjan 1972). This algorithm runs in time $O(|N| + |A|)$ and therefore should scale to very large graphs.

Biconnected Components of a Simple Undirected Graph

This section illustrates the use of the biconnected components algorithm on the simple undirected graph G shown in Figure 2.16.

Figure 2.16 A Simple Undirected Graph *G*



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

```
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:

```
proc network
   links    = mycas.LinkSetInBiCC
   outLinks = mycas.LinkSetOut
   outNodes = mycas.NodeSetOut;
   biconnectedComponents;
run;
```

The output data table mycas.LinkSetOut contains the biconnected components of the input graph, as shown in Figure 2.17.

Figure 2.17 Biconnected Components of a Simple Undirected Graph

from	to	biconcomp
A	F	1
A	B	1
A	G	2
E	F	1
B	E	1
B	D	3
B	C	3
G	H	4
G	I	4
H	I	4
C	D	3

The output data table mycas.NodeSetOut contains the articulation points of the input graph, as shown in Figure 2.18.

Figure 2.18 Articulation Points of a Simple Undirected Graph

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

The biconnected components are shown graphically in Figure 2.19 and Figure 2.20.

Figure 2.19 Biconnected Components C^1 and C^2

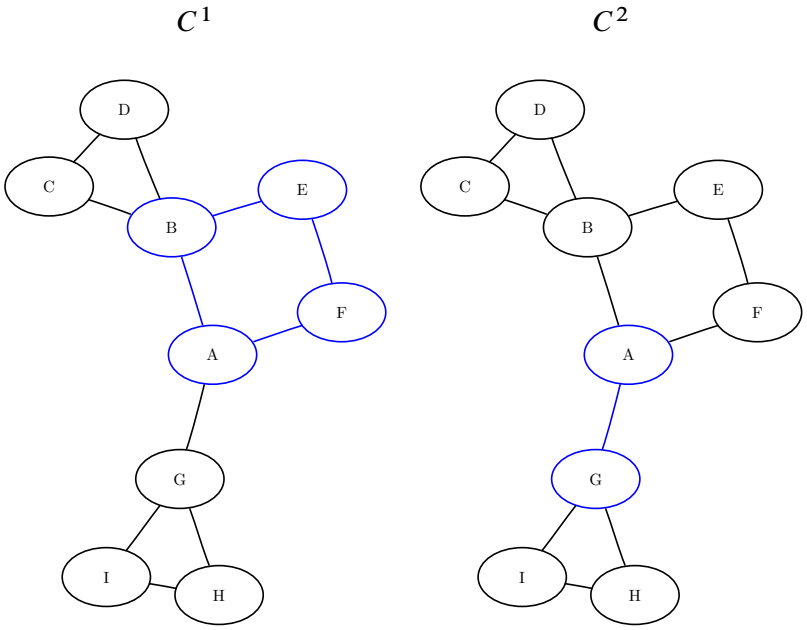
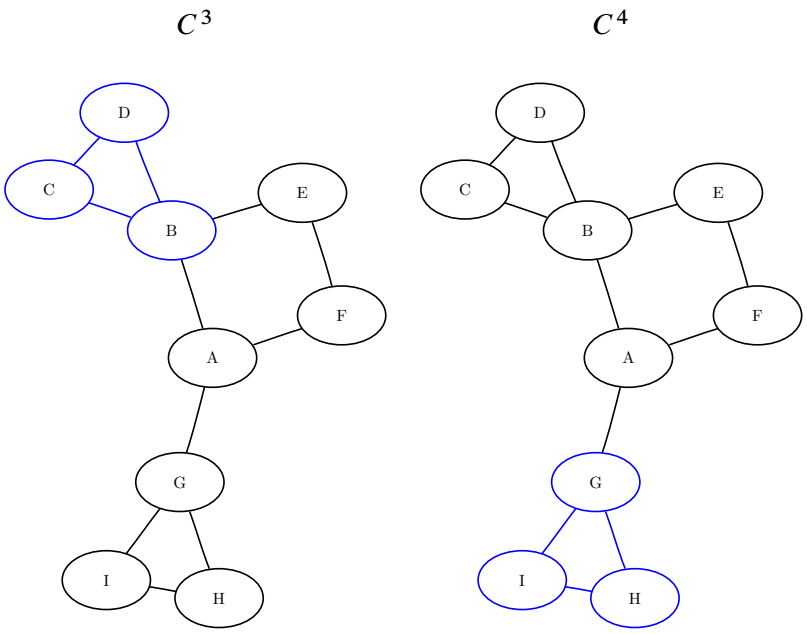


Figure 2.20 Biconnected Components C^3 and C^4



For a more detailed example, see “Example 2.1: Articulation Points in a Terrorist Network” on page 124.

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 24.

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

Degree Centrality

The *degree* of a node v in an undirected graph is the number of links that are incident to node v. The *out-degree* of a node in a directed graph is the number of out-links incident to that node; the *in-degree* is the number of in-links incident 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 is specified in the OUTNODES= option in 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 40. The following statements calculate the degree centrality for both in- and out-degrees:

```
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 2.21.

Figure 2.21 Degree Centrality of a Simple Directed Graph

node	centr_degree_in	centr_degree_out	centr_degree
A	0	3	3
B	1	3	4
C	2	1	3
D	2	1	3
E	3	2	5
F	2	1	3
G	2	2	4
H	1	2	3
I	2	0	2

Influence Centrality

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

Let w_{uv} define the link weight of link (u, v), and let w_u define the node weight of node u. Let δ_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 is specified 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:

```
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 2.22.

Figure 2.22 Influence Centrality of a Simple Directed Graph

node	centr_influence1_unwt	centr_influence2_unwt
A	0.33333	0.55556
B	0.33333	0.44444
C	0.11111	0.22222
D	0.11111	0.22222
E	0.22222	0.22222
F	0.11111	0.22222
G	0.22222	0.22222
H	0.22222	0.22222
I	0.00000	0.00000

For a more detailed example, see "Example 2.2: Influence Centrality for Project Groups in a Research Department" on page 126.

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

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

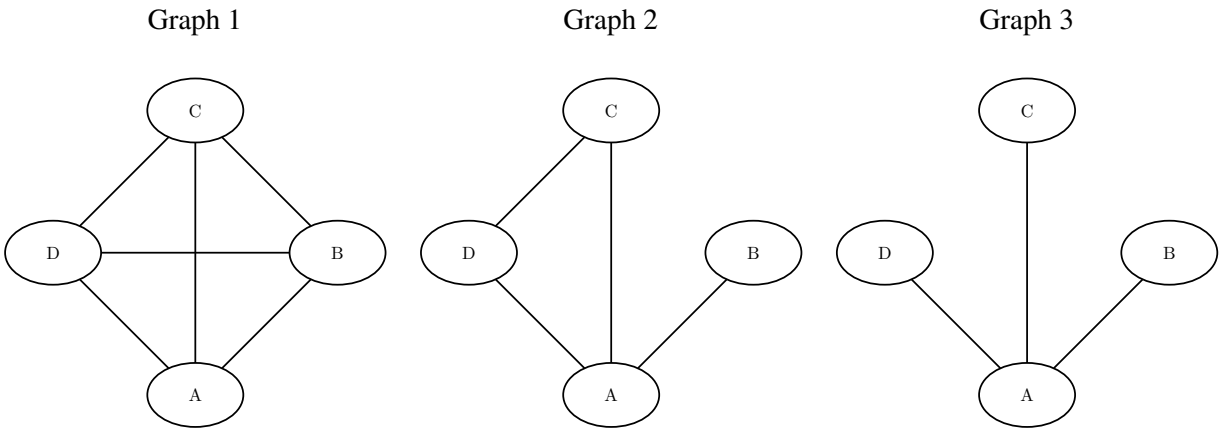
For a particular node i, the clustering coefficient determines how close the subgraph induced by its neighbor set N_i is to being a clique (complete subgraph). In social networks, a high clustering coefficient can help predict relationships that might not be known, confirmed, or realized yet. The fact 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 OUTNODES= 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 2.23.

Figure 2.23 Three Undirected Graphs



Define the three links data tables as follows:

```
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:

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

proc network
   links     = mycas.LinkSetInCC2
   outNodes  = mycas.NodeSetOut2;
   centrality
      clusteringCoef;
run;
```

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

The nodes data tables provide the clustering coefficients of each graph (the centr_cluster variable), as shown in Figure 2.24 through Figure 2.26.

Figure 2.24 Clustering Coefficient of a Simple Undirected Graph 1

node	centr_cluster
A	1
B	1
C	1
D	1

Figure 2.25 Clustering Coefficient of a Simple Undirected Graph 2

node	centr_cluster
A	0.33333
B	0.00000
C	1.00000
D	1.00000

Figure 2.26 Clustering Coefficient of a Simple Undirected Graph 3

node	centr_cluster
A	0
B	0
C	0
D	0

Closeness Centrality

Closeness centrality is the reciprocal of the average of the shortest 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 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

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

where p defines a penalty parameter for unreachable nodes, $n(u)$ defines the number of nodes that are considered in calculating the average, and $s(u)$ is a scaling factor, as shown in Table 2.9.

Table 2.9 Formulas for CLOSENOPATH= Option for Undirected Graphs

CLOSENOPATH=	p	$n(u)$	$s(u)$
DIAMETER	$\max_{(i,j) \in A} \{d_{ij} : d_{ij} < \infty\} + 1$	$\|N\| - 1$	1
NNODES	$\|N\|$	$\|N\| - 1$	1
ZERO	0	$\|R(u)\| - 1$	$\frac{\|R(u)\|-1}{\|N\|-1}$

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

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

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

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

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.

Table 2.10 Formulas for CLOSENOPATH= Option for Directed Graphs

CLOSENOPATH=	$n^{\text{out}}(u)$	$s^{\text{out}}(u)$	$n^{\text{in}}(u)$	$s^{\text{in}}(u)$
DIAMETER	$\|N\|-1$	1	$\|N\|-1$	1
NNODES	$\|N\|-1$	1	$\|N\|-1$	1
ZERO	$\|R^{\text{out}}(u)\|-1$	$\frac{\|R^{\text{out}}(u)\|-1}{\|N\|-1}$	$\|R^{\text{in}}(u)\|-1$	$\frac{\|R^{\text{in}}(u)\|-1}{\|N\|-1}$

The overall closeness centrality for directed graphs is calculated as

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

Harmonic Centrality

Harmonic centrality, as described in Rochat (2009), is a variant of closeness centrality that attempts to simplify the treatment of unreachable nodes by calculating the average of the reciprocal of the shortest path distances from a particular node to all 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 40. The following statements calculate the closeness centrality for both the weighted and unweighted graphs:

```
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 2.27.

Figure 2.27 Closeness Centrality of a Simple Directed Graph

node	centr_close_wt	centr_close_in_wt	centr_close_out_wt	centr_close_unwt	centr_close_in_unwt	centr_close_out_unwt
A	0.38835	0.00000	0.77670	0.22222	0.00000	0.44444
B	0.55134	0.30000	0.80268	0.33333	0.22222	0.44444
C	0.38260	0.34043	0.42478	0.27885	0.25000	0.30769
D	0.40113	0.41202	0.39024	0.29178	0.30769	0.27586
E	0.45962	0.43439	0.48485	0.32000	0.32000	0.32000
F	0.44357	0.50314	0.38400	0.30725	0.34783	0.26667
G	0.56402	0.79470	0.33333	0.32500	0.40000	0.25000
H	0.41663	0.48290	0.35036	0.27885	0.30769	0.25000
I	0.30227	0.60453	0.00000	0.18182	0.36364	0.00000

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

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

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

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

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

By default, this metric is normalized by dividing 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 is specified in the OUTNODES= option in the PROC NETWORK statement. The link betweenness results are provided in the link output data table that is specified 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 $O(|N| \times (|N| \log |N| + |A|))$. 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 40. The following statements calculate the betweenness centrality for both the weighted and unweighted graphs:

```
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 2.28.

Figure 2.28 Node Betweenness Centrality of a Simple Directed Graph

node	centr_between_wt	centr_between_unwt
A	0.00000	0.00000
B	0.07143	0.07738
C	0.00000	0.00595
D	0.01786	0.00595
E	0.17857	0.17857
F	0.26786	0.26786
G	0.21429	0.21429
H	0.00000	0.00000
I	0.00000	0.00000

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 2.29.

Figure 2.29 Link Betweenness Centrality of a Simple Directed Graph

from	to	weight	centr_between_wt	centr_between_unwt
A	B	1	0.08929	0.09524
A	C	2	0.01786	0.02381
A	D	4	0.03571	0.02381
B	C	1	0.01786	0.01786
B	E	2	0.03571	0.04167
B	F	5	0.14286	0.14286
C	E	1	0.10714	0.11310
D	E	1	0.10714	0.09524
E	D	1	0.05357	0.05357
E	F	2	0.21429	0.21429
F	G	6	0.32143	0.32143
G	H	1	0.12500	0.12500
H	G	2	0.01786	0.01786
G	I	1	0.12500	0.12500
H	I	3	0.01786	0.01786

For more detailed examples, see "Example 2.3: Betweenness and Closeness Centrality for Computer Network Topology" on page 130 and "Example 2.4: Betweenness and Closeness Centrality for Project Groups in a Research Department" on page 133.

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, λ is a constant, δ_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

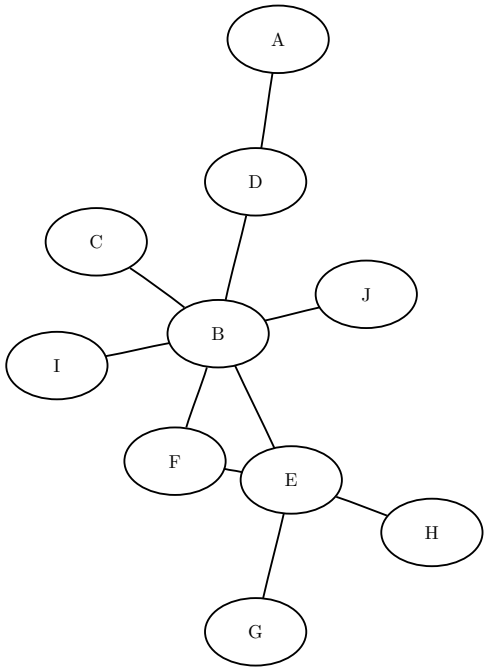
$$Ax = \lambda x$$

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

Eigenvector centrality is calculated 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 2.30.

Figure 2.30 Eigenvector Centrality Example of a Simple Undirected Graph



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

```
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:

```
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 2.31.

Figure 2.31 Eigenvector Centrality Output

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

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

For a more detailed example, see "Example 2.5: Eigenvector Centrality for Word Sense Disambiguation" on page 136.

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 12 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,

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

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 α and β are constants.

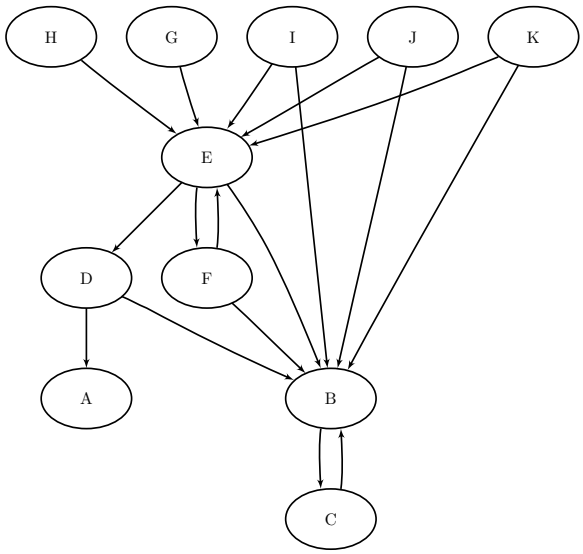
The definition can be written in matrix form as follows:

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

Thus, the authority and hub centralities are the principal eigenvectors of $A^T A$ and AA^T, respectively. To solve this eigenvector problem, PROC 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. JACOBIDAVIDSON, 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 2.32. 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 2.32 Hub and Authority Centrality Example of a Simple Directed Graph



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

```
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:

```
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 2.33.

Figure 2.33 Hub and Authority Centrality Output

node	centr_hub_unwt	centr_auth_unwt
B	0.00000	1.00000
C	0.54135	0.00000
D	0.59703	0.11466
A	0.00000	0.10287
E	0.66549	0.84725
F	1.00000	0.11466
G	0.45865	0.00000
H	0.45865	0.00000
I	1.00000	0.00000
J	1.00000	0.00000
K	1.00000	0.00000

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

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 27. The clique algorithm works only with undirected graphs.

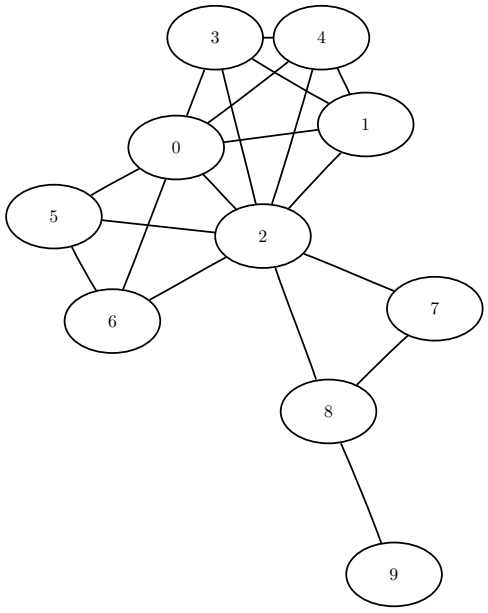
The results of the clique algorithm are written to the output data table that is specified in the OUT= option in the CLIQUE statement. Each node of each clique is listed in the output data table along with the variable clique to identify the clique to which it belongs. The clique identifiers are numbered 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 typically does not scale to very large graphs.

Maximal Cliques of a Simple Undirected Graph

This section illustrates the use of the clique algorithm on the simple undirected graph *G* shown in Figure 2.34.

Figure 2.34 A Simple Undirected Graph *G*



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

```
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:

```
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 2.35.

Figure 2.35 Maximal Cliques of a Simple Undirected Graph

clique	node
1	0
1	2
1	3
1	4
1	1
2	0
2	2
2	5
2	6
3	2
3	8
3	7
4	8
4	9

In addition, the output data table mycas.CliqueSizes contains the number of nodes in each clique, as shown in Figure 2.36.

Figure 2.36 Sizes of Maximal Cliques of a Simple Undirected Graph

clique	size
1	5
2	4
3	3
4	2

The maximal cliques are shown graphically in Figure 2.37 and Figure 2.38.

Figure 2.37 Maximal Cliques C^1 and C^2

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

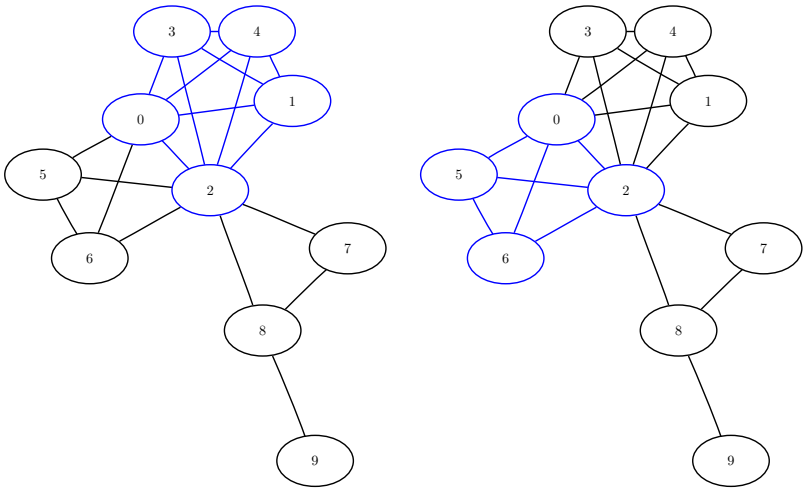
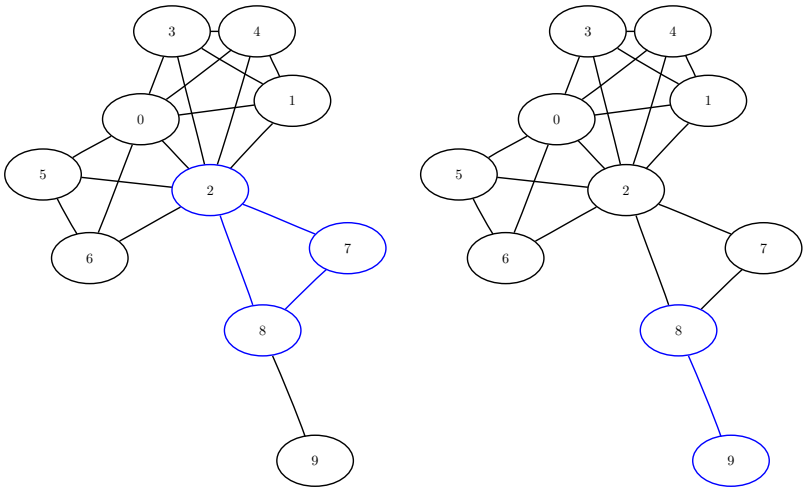


Figure 2.38 Maximal Cliques C^3 and C^4

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



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 27.

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 both undirected and directed graphs. 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. Using 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

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

where Q is the modularity, w_{uv} is the link weight between nodes u and v (or 0 if $(u, v) \notin A$), δ_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=PARALLELLABELPROP, 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=PARALLELLABELPROP.

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=PARALLELLABELPROP in the COMMUNITY statement, community detection supports both undirected and directed graphs. However, you should be careful in deciding whether to model your problem as an undirected or a directed graph. For an undirected graph, the algorithm finds communities based on the density of the subgraphs. For a directed graph, the algorithm finds communities based on the information flow along the directed links. That is, the algorithm propagates the community 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.

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=PARALLELLABELPROP 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=PARALLELLABELPROP, 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=PARALLELLABELPROP 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 139 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.

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

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:

- 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.
- from: the *from* node label
- to: the *to* node label

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, this data table is not produced if you specify ALGORITHM=PARALLELLABELPROP together with multiple resolution values in the RESOLUTIONLIST= option. However, if you specify ALGORITHM=LOUVAIN, the data table is produced and will contain only results corresponding to the smallest value of the RESOLUTIONLIST= option. This data table contains the following columns:

- node: 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:

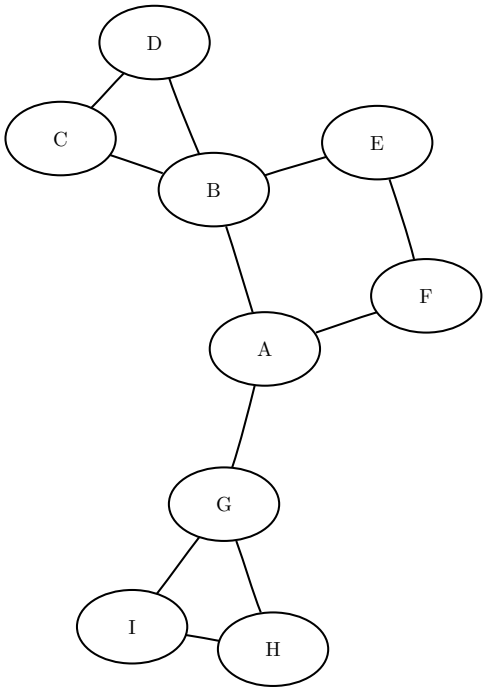
- level: resolution level number
- resolution: resolution value
- from_community: community identifier of the *from* community
- to_community: community identifier of the *to* community
- link_weight: sum of link weights of all links between from_community and to_community

This data table is not produced if you specify ALGORITHM=PARALLELLABELPROP together with multiple resolution values in the RESOLUTIONLIST= option.

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 2.39.

Figure 2.39 A Simple Undirected Graph *G*



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

```
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.

```
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 2.40.

Figure 2.40 Community Detection on an Undirected Simple Graph: Nodes Output

node	community_1	community_2
A	1	1
B	2	1
F	1	1
G	3	2
C	2	1
D	2	1
E	1	1
H	3	2
I	3	2

The output data table mycas.CommLevelOut contains summary information at each resolution level, as shown in Figure 2.41.

Figure 2.41 Community Detection on an Undirected Simple Graph: Level Output

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

The output data table mycas.CommOut contains the number of nodes in each community, as shown in Figure 2.42.

Figure 2.42 Community Detection on an Undirected Simple Graph: Community Summary

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

The output data table mycas.CommOverlapOut contains community overlap information, as shown in Figure 2.43.

Figure 2.43 Community Detection on an Undirected Simple Graph: Community Overlap

node	community	intensity
A	1	0.66667
A	2	0.33333
B	1	1.00000
F	1	1.00000
G	1	0.33333
G	2	0.66667
C	1	1.00000
D	1	1.00000
E	1	1.00000
H	2	1.00000
I	2	1.00000

The output data table mycas.CommLinksOut describes how the communities are connected, as shown in Figure 2.44.

Figure 2.44 Community Detection on an Undirected Simple Graph: Intercommunity Links

level	resolution	from_community	to_community	link_weight
1	1.0	1	2	2
1	1.0	1	3	1
2	0.5	1	2	1

Connected Components

A *connected component* of a graph is a set of nodes that are all reachable from each other. That is, if two nodes are in the same component, then there 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 30.

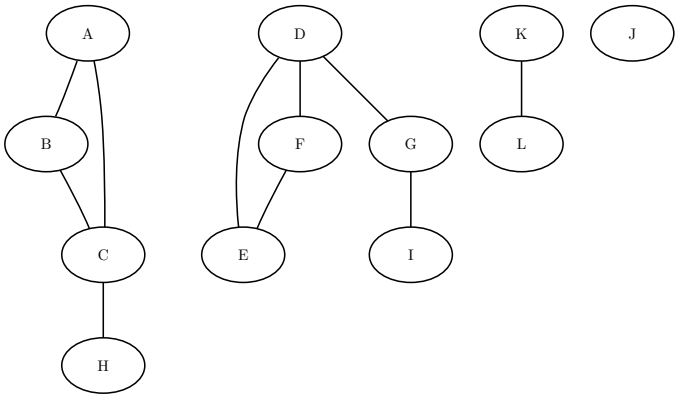
There are three algorithms for finding connected components in an undirected graph: a depth-first search algorithm (ALGORITHM=DFS), a union-find algorithm (ALGORITHM=UNIONFIND), and a distributed parallel union-find algorithm (ALGORITHM=PARALLEL). For a graph $G = (N, A)$, each algorithm runs in time $O(|N| + |A|)$ 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 is available (ALGORITHM=DFS).

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 2.45.

Figure 2.45 A Simple Undirected Graph G



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

```
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:

```
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 2.46.

Figure 2.46 Connected Components of a Simple Undirected Graph

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

Notice that the graph is defined by using only the links data table. As seen in Figure 2.45, this graph also contains a singleton node labeled J, which has no associated links. By definition, this node defines its own component. But because the input graph was defined by using only the links data table, it did 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:

```
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:

```
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 2.47.

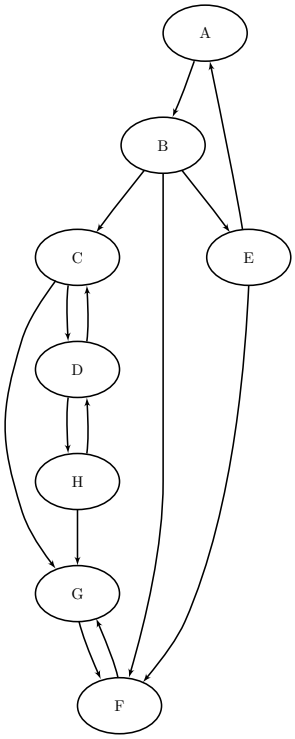
Figure 2.47 Connected Components of a Simple Undirected Graph

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

Connected Components of a Simple Directed Graph

This section illustrates the use of the connected components algorithm on the simple directed graph *G* shown in Figure 2.48.

Figure 2.48 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;
```

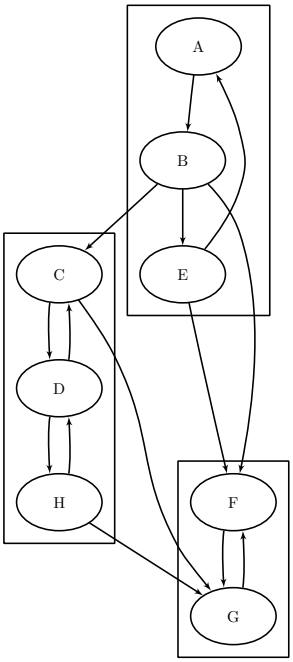
The output data table mycas.NodeSetOut, shown in Figure 2.49, now contains the connected components of the input graph.

Figure 2.49 Connected Components of a Simple Directed Graph

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

The connected components are represented graphically in Figure 2.50.

Figure 2.50 Strongly Connected Components of Graph G



Core Decomposition

An alternative to community detection for detecting 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 give a coarse approximation of cohesive structure at a very low computational cost. Let $G = (N, A)$ define a graph with nodes N and links A, and let $G_S = (S, A_S)$ be an induced subgraph on nodes S. The subgraph G_S is a *k-core* if and only if for every node $v \in S$, the degree of v is greater than or equal to k and G_S is the maximum subgraph with this property. By definition, the cores are nested. That is, if G_{S_k} is a k-core of size k, then $G_{S_{k+1}}$ is contained in G_{S_k}.

In PROC 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 31.

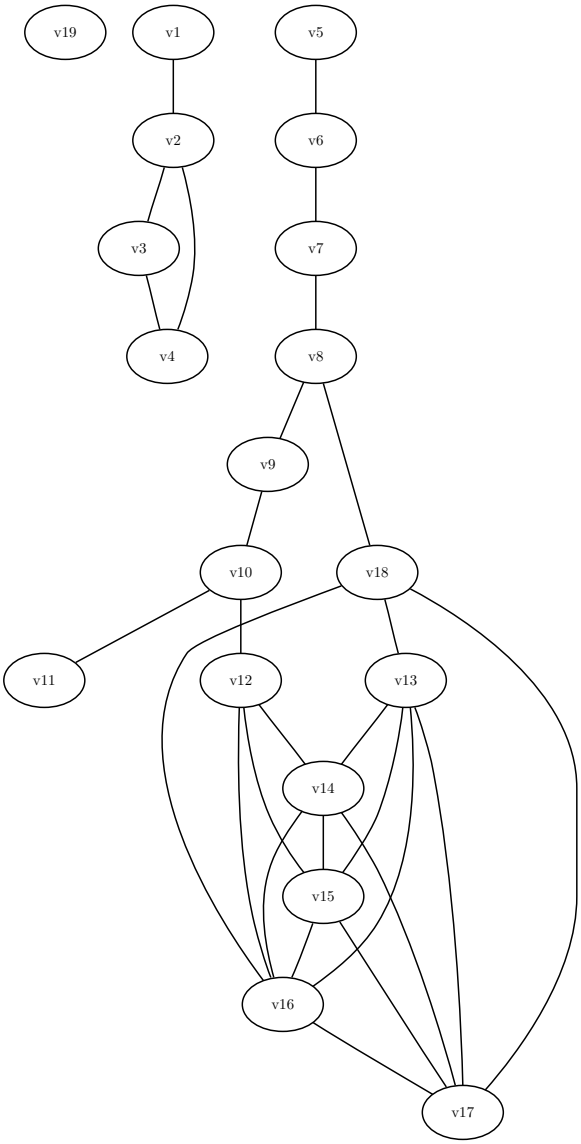
The results of the core decomposition algorithm are given in the output nodes data table that is specified in the OUTNODES= option in the PROC NETWORK statement. For each node in the nodes data table, the variable core_out identifies its *core number*, the highest-order core that contains this node.

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 2.51.

Figure 2.51 Simple Undirected Graph



The undirected graph G can be represented using the following nodes data table, mycas.NodeSetIn, and links data table, mycas.LinkSetIn:

```
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;
v1   v2   v5  v6   v6  v7   v7  v8  v10 v11
v2   v3   v3  v4   v2  v4   v8  v9   v9 v10
v8  v18  v10 v12  v13 v14  v13 v15  v13 v16
v13 v17  v14 v15  v14 v16  v14 v17  v15 v16
v15 v17  v16 v17  v18 v13  v18 v17  v18 v16
v12 v14  v12 v15  v12 v16
;
```

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

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

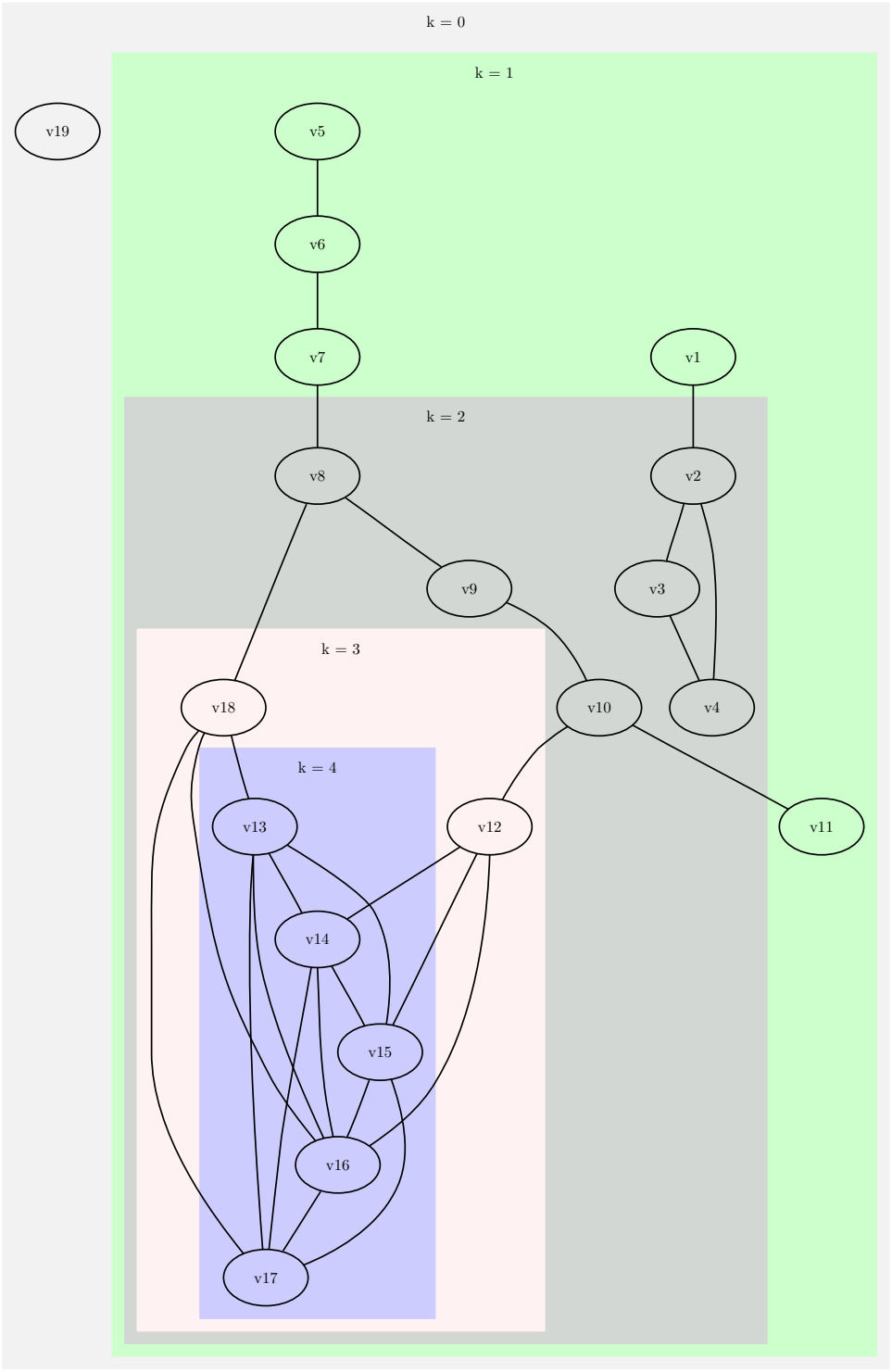
The nodes output data table mycas.NodeSetOut contains the core number (variable core_out) for each node, as shown in Figure 2.52.

Figure 2.52 Core Decomposition of a Simple Undirected Graph

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

Figure 2.53 shows the graph layered by its core number.

Figure 2.53 Core Decomposition



Cycle Enumeration

A *path* in a graph is a sequence of nodes, each of which has a link to the next node in the sequence. An *elementary cycle* is a path in which the start node and the end node are the same and no other node appears more than once 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 32. To find the cycles and report them in an output data table, use the OUT= option. To simply count the cycles, do not use the OUT= option.

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

The results 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.

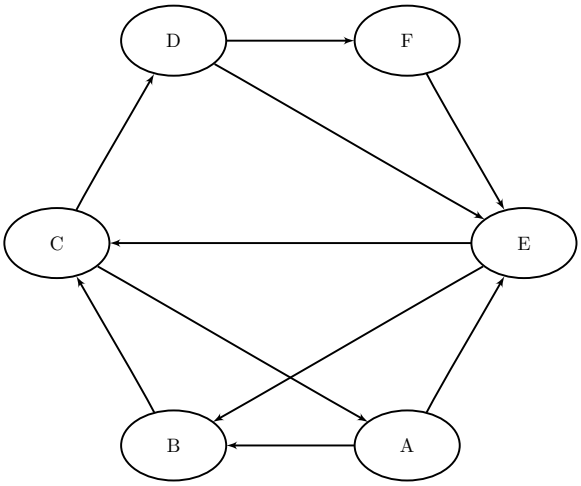
The algorithm that PROC NETWORK uses to compute all cycles is a variant of the algorithm in Johnson (1975). This algorithm runs in time $O((|N| + |A|)(c + 1))$, where c is the number of elementary cycles in the graph. So the algorithm should scale to large graphs that contain few cycles. However, some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

If 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 32.

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 2.54. For a more detailed example involving both cycle enumeration and transitive closure, see "Example 2.9: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System" on page 151.

Figure 2.54 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  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):

```
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 2.55.

Figure 2.55 PROC NETWORK Log: Count the Number of Cycles in a Simple Directed Graph

```
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 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 0.041392 seconds.

STATUS=OK  PROBLEM_TYPE=CYCLE  SOLUTION_STATUS=OK  NUM_CYCLES=7  CPU_TIME=0.07  REAL_TIME=0.04
```

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

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

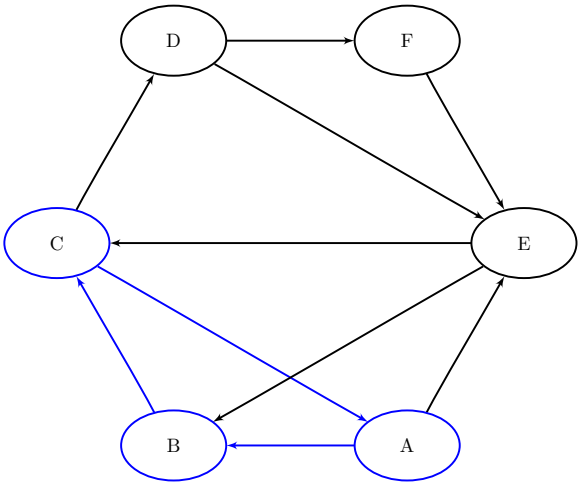
The output data table mycas.Cycles now contains the first cycle found in the input graph, as shown in Figure 2.56.

Figure 2.56 First Cycle Found in a Simple Directed Graph

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

The first cycle that is found in the input graph is shown graphically in Figure 2.57.

Figure 2.57 $A \rightarrow B \rightarrow C \rightarrow A$



The following statements return all the cycles in the graph:

```
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 2.58.

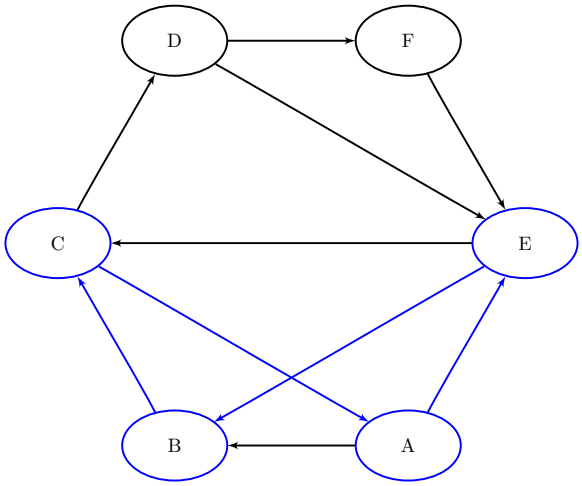
Figure 2.58 All Cycles in a Simple Directed Graph

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

The six additional cycles are shown graphically in Figure 2.59 through Figure 2.61.

Figure 2.59 Cycles

$A \to E \to B \to C \to A$



$A \to E \to C \to A$

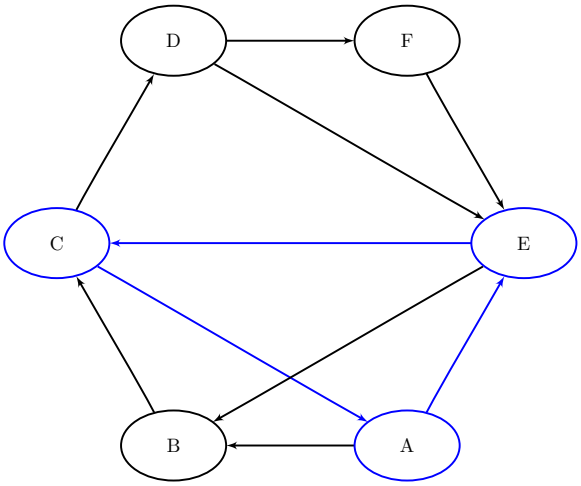
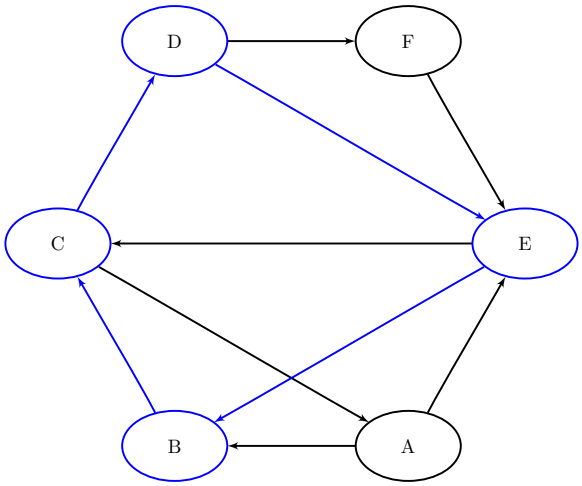


Figure 2.60 Cycles

$B \to C \to D \to E \to B$



$B \to C \to D \to F \to E \to B$

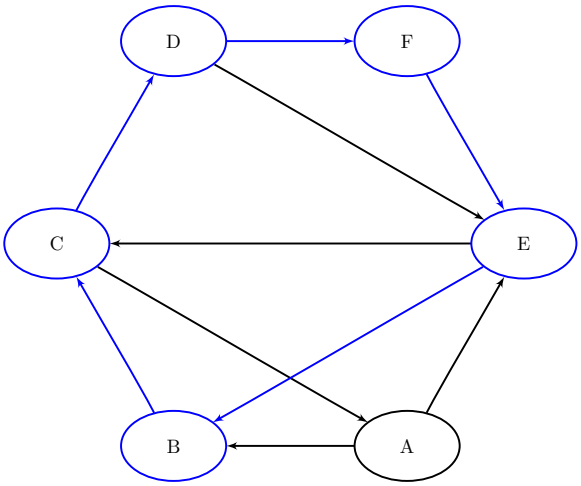
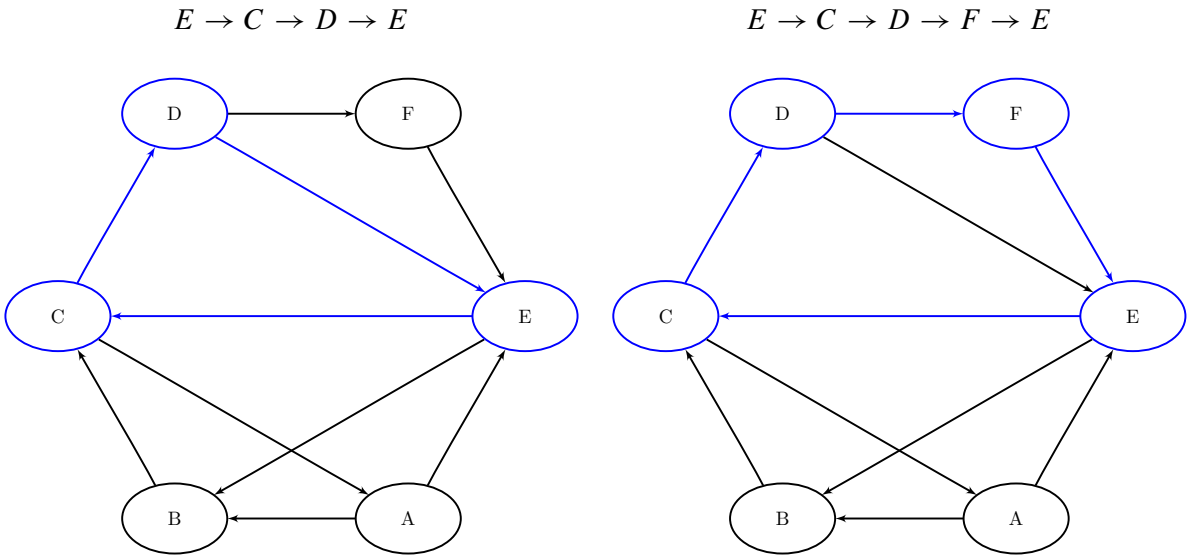


Figure 2.61 Cycles



Reach (Ego) Network

The *reach network* of a graph $G = (N, A)$ is a graph $G_L^R = (N_L^R, A_L^R)$ that is defined as the induced subgraph over the set of nodes N_L^R that are reachable in L steps (or hops) from a set S of nodes, called the *source nodes*. Reach networks are often referred to as *ego networks* in the context of social networks, 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 36.

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 44. 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)
- 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

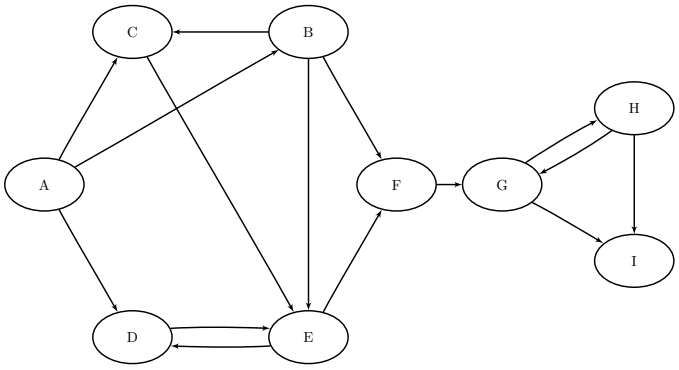
If node weights are present, the OUTCOUNTS= data table contains the following additional columns:

- 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
- 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 2.62.

Figure 2.62 Simple Directed Graph G



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

```
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:

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

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 2.63.

Figure 2.63 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

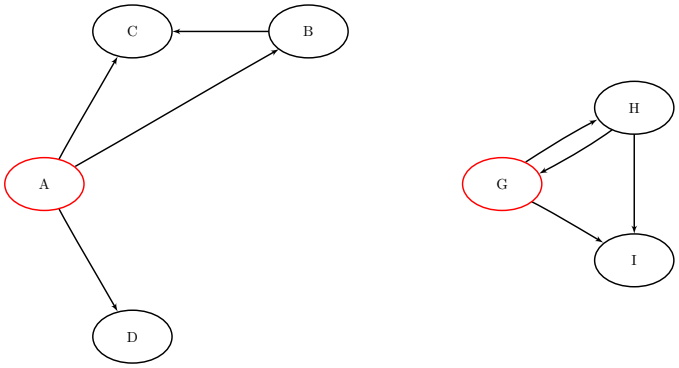
reach	from	to
1	A	B
1	A	C
1	A	D
1	B	C
1	G	H
1	H	G
1	G	I
1	H	I

ReachCounts1

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

The results are displayed graphically in Figure 2.64.

Figure 2.64 Reach Network for $S_1 = \{A, G\}$ with Hop Limit of 1



For the second set of source nodes, you can use the following statements to find the reach network that is restricted by a hop limit of 2:

```
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 2.65.

Figure 2.65 Reach Network for $S_2 = \{B\}$ with Hop Limit of 2

ReachNodes2

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

Figure 2.65 *continued*

ReachLinks2

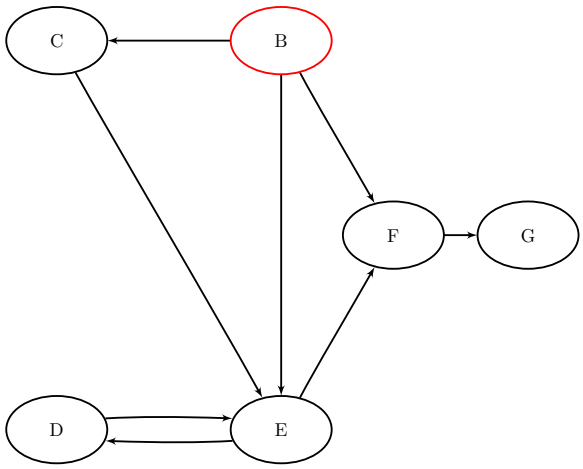
reach	from	to
1	B	C
1	B	E
1	B	F
1	C	E
1	D	E
1	E	D
1	E	F
1	F	G

ReachCounts2

reach	node	count	count_not
1	B	6	3

The results are displayed graphically in Figure 2.66.

Figure 2.66 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 2.62, 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:

```
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 2.67.

Figure 2.67 Reach Networks for $S_1 = \{C\}$ and $S_2 = \{A, H\}$ with Hop Limit of 1

ReachNodes

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

ReachLinks

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

Figure 2.67 *continued*

ReachCounts

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

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 37.

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 44. The following sections show examples of the use of 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 2.11 Algorithms for Shortest Paths

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

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

For weighted graphs, the algorithm uses the 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 2.11: Shortest Paths of the New York Road Network" on page 156.

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.

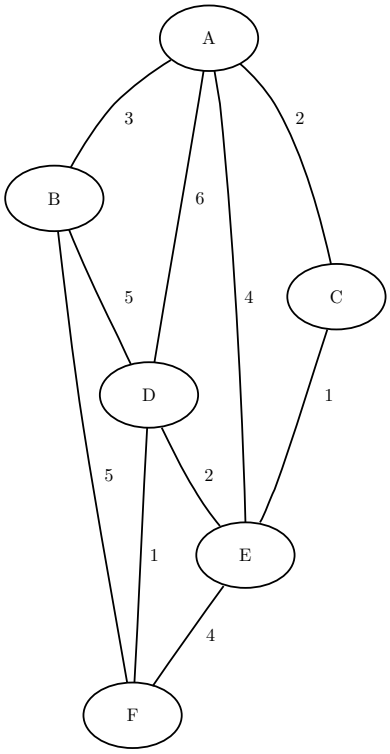
This data table contains the following columns:

- 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 the AUXWEIGHT= option is defined 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 2.68.

Figure 2.68 A Simple Undirected Graph G



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

```
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:

```
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 2.69.

Figure 2.69 All-Pairs Shortest Paths

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

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

The output data table mycas.ShortPathW contains the path weights of the shortest paths of each source-sink pair, as shown in Figure 2.70.

Figure 2.70 All-Pairs Shortest Paths Summary

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

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

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 $S \times T = \{A, C\} \times \{B, F\}$:

```
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:

```
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 2.71.

Figure 2.71 Shortest Paths for a Subset of Source-Sink Pairs

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

Shortest Paths for a Subset of Source or Sink Pairs

This section illustrates the use of the shortest path algorithm 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 pairs from a subset of source nodes (or to calculate all pairs to a subset of sink nodes).

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

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

You can use the same PROC NETWORK call as is used in the section "Shortest Paths for a Subset of Source-Sink Pairs" on page 105 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 2.72.

Figure 2.72 Shortest Paths for a Subset of Source Pairs

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

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

```
data 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 2.73.

Figure 2.73 Shortest Paths for a Subset of Sink Pairs

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

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*:

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

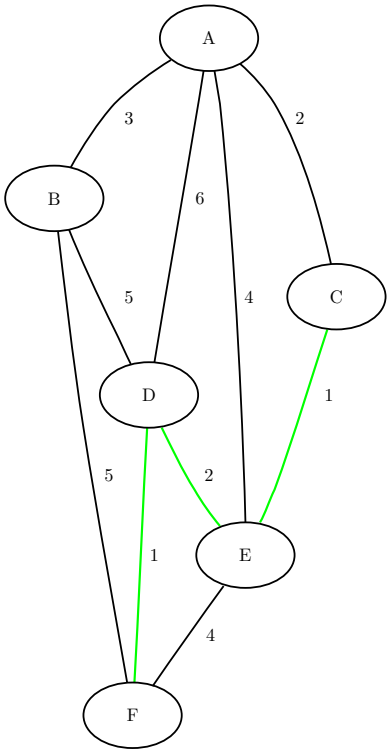
The output data table mycas.ShortPath contains this shortest path, as shown in Figure 2.74.

Figure 2.74 Shortest Paths for One Source-Sink Pair

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

The shortest path is shown graphically in Figure 2.75.

Figure 2.75 Shortest Path between Nodes C and F



Shortest Paths with Auxiliary Weight Calculation

This section illustrates the use of the shortest path algorithm with auxiliary weights 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:

```
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:

```
proc network
   links         = mycas.LinkSetIn;
   linksVar
      auxWeight  = count;
   shortestPath
      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 2.76. 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 2.76 Shortest Paths Including Auxiliary Weights in Calculation

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

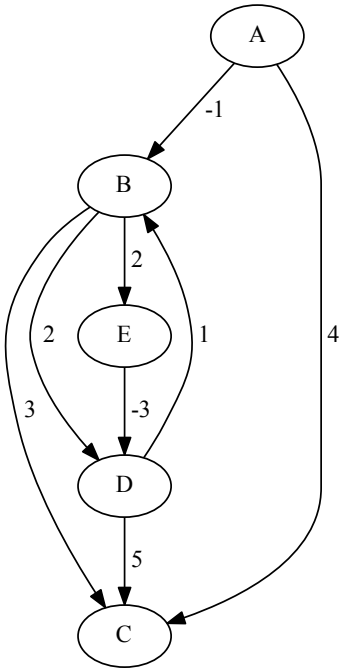
source	sink	path_weight	path_auxweight
D	A	5	3
D	B	5	1
D	C	3	2
D	E	2	1
D	F	1	1
B	A	3	1
B	C	5	2
B	D	5	1
B	E	6	3
B	F	5	1
E	A	3	2
E	B	6	3
E	C	1	1
E	D	2	1
E	F	3	2

The section "Road Network Shortest Path" on page 9 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 2.77.

Figure 2.77 A Simple Directed Graph *G* with Negative Link Weights



You can represent the directed graph *G* by using the following links data table, mycas.LinkSetIn:

```
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*:

```
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 2.78.

Figure 2.78 Shortest Paths with Negative Link Weights

source	sink	order	from	to	weight
E	B	1	E	D	-3
E	B	2	D	B	1

Now, consider the following adjustment to the weight of link (B, E):

```
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 this and produces an error message, as shown in Figure 2.79.

Figure 2.79 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 0.07 (cpu: 0.00) seconds.
ERROR: The action stopped due to errors.
NOTE: The Cloud Analytic Services server processed the request in 0.148603 seconds.
NOTE: The SAS System stopped processing this step because of errors.
STATUS=ERROR  PROBLEM_TYPE=SHORTESTPATH  CPU_TIME=0.11  REAL_TIME=0.15
```

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 38.

Output Data Tables

The summary statistics that are produced are broken into two categories: statistics on the entire graph and statistics on the nodes and links of the graph. The latter statistics are appended to the output nodes and links data 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 $\left(\frac{|A|}{|K(N)|}\right)$
- 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 80 and "Biconnected Components and Articulation Points" on page 51, 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 *diameter* of a graph is the longest possible shortest path distance of all source-sink pairs that can occur in the graph. For more information about shortest paths, see the section "Shortest Path" on page 101. If you use the SHORTESTPATH= option, the following columns also appear in the summary output data table:

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

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 $O(\sqrt{|N|})$. If you use the DIAMETERAPPROX= option, the following columns also appear in the summary output data table:

- diameter_approx_wt: approximate longest weighted shortest path in the graph
- diameter_approx_unwt: approximate longest unweighted shortest path in the graph

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: 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 options, 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 (.)

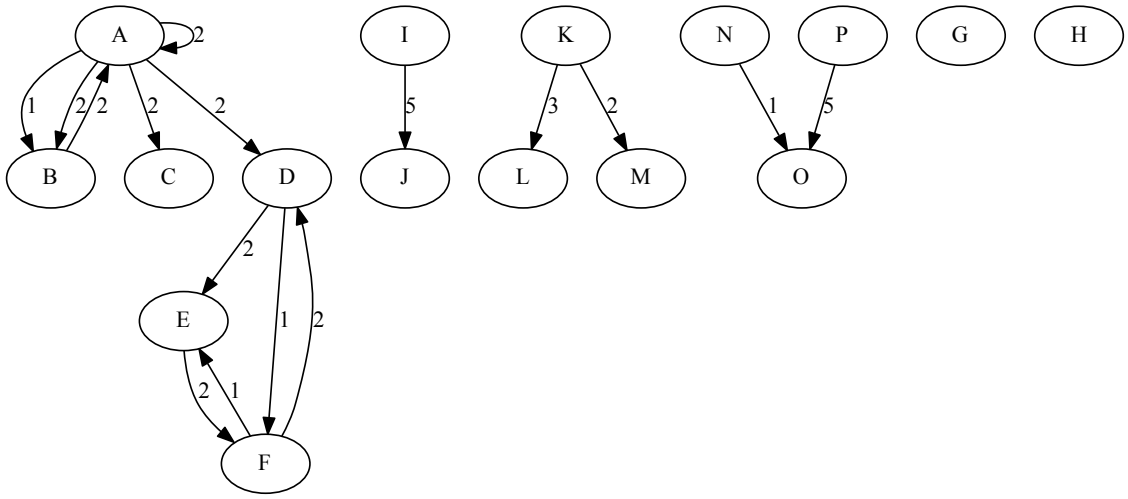
The following columns are appended for directed graphs:

- 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 2.80.

Figure 2.80 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:

```
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:

```
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 2.81.

Figure 2.81 Graph Summary Statistics of a Simple Directed Graph

nodes	links	avg_links_per_node	density	self_links_ignored	dup_links_ignored	leaf_nodes	singleton_nodes
16	14	0.875	0.058333	1	1	5	2

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.

```
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 2.82.

Figure 2.82 Graph Summary and Connectedness Statistics of a Simple Directed Graph

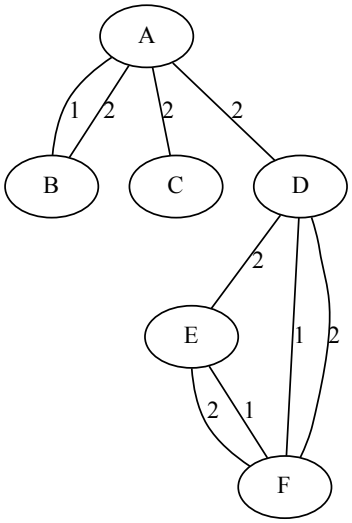
nodes	links	avg_links_per_node	density	self_links_ignored	dup_links_ignored	leaf_nodes	singleton_nodes
16	14	0.875	0.058333	1	1	5	2

concomp	isolated_pairs	isolated_stars_out	isolated_stars_in
13	1	1	1

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 2.83.

Figure 2.83 A Simple Undirected Graph G



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 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 2.84.

Figure 2.84 Graph Summary and Shortest Path Statistics of a Simple Undirected Graph

nodes	links	avg_links_per_node	density	self_links_ignored	dup_links_ignored	leaf_nodes	singleton_nodes	diameter_wt	avg_shortpath_wt
6	6	1	0.4	0	3	2	0	6	3.06667

Figure 2.84 *continued*

node	leaf_node	singleton_node	neighbor_leaf_nodes	sum_in_and_out_wt	eccentr_wt_out
A	0	0	2	5	4
B	1	0	0	1	5
C	1	0	0	2	6
D	0	0	0	5	4
E	0	0	0	3	6
F	0	0	0	2	5

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 39.

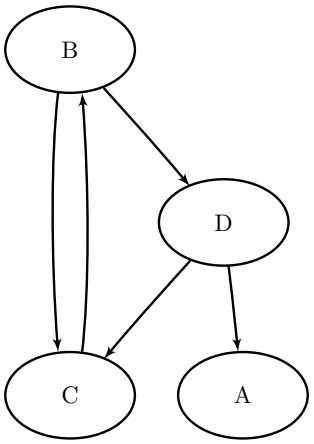
The links that define the transitive closure of the input graph are written to the output data table that is specified in the OUT= option in the TRANSITIVECLOSURE statement.

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 2.85.

Figure 2.85 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;
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:

```
proc network
   direction       = directed
   links           = mycas.LinkSetIn;
   transitiveClosure
      out          = mycas.TransClosure;
run;
```

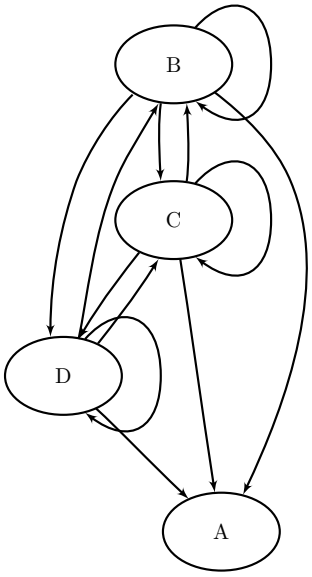
The output data table mycas.TransClosure contains the transitive closure of *G*, as shown in Figure 2.86.

Figure 2.86 Transitive Closure of a Simple Directed Graph

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

The transitive closure of *G* is shown graphically in Figure 2.87.

Figure 2.87 Transitive Closure of G



For a more detailed example, see "Example 2.9: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System" on page 151.

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

CONNECTEDCOMPONENTS	Connected components
CORE	Core decomposition
CYCLE	Cycle enumeration
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.
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) used by PROC NETWORK.

REAL_TIME
indicates the elapsed time (in seconds) used by PROC NETWORK.

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 found by the algorithm.

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

The CLIQUE algorithm provides the following additional information:

NUM_CLIQUES
indicates the number of cliques found by the algorithm.

The CONNECTEDCOMPONENTS algorithm provides the following additional information:

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

The CYCLE algorithm provides the following additional information:

NUM_CYCLES
indicates the number of cycles found by the algorithm.

The SHORTESTPATH algorithm provides the following additional information:

NUM_PATHS
indicates the number of shortest paths found by the algorithm.

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 2.12 ODS Tables Produced by PROC NETWORK

Table Name	Description
ProblemSummary	Summary of the graph
SolutionSummary	Summary of the solution status, timing, and results

The following statements use the example in the section "Shortest Paths for All Pairs" on page 102 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
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 2.88 provides a basic summary of the graph input.

Figure 2.88 Problem Summary Table

The NETWORK Procedure

Problem Summary	
Number of Nodes	6
Number of Links	10
Graph Direction	Undirected

The solution summary table in Figure 2.89 provides a basic solution summary for the algorithm that is processed. The information in these tables matches the information that is provided in the macro variable _NETWORK_, described in the section "Macro Variable _NETWORK_" on page 121.

Figure 2.89 Solution Summary Table

Solution Summary	
Problem Type	Shortest Path
Solution Status	OK
Number of Paths	30
CPU Time	0.01
Real Time	0.03

Examples: NETWORK Procedure

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 2.90 shows this network, which was constructed after the attacks, based on collected intelligence information.

Figure 2.90 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:

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

```
Jerome Courtaillier        David Courtaillier
Zacarias Moussaoui         David Courtaillier
Zacarias Moussaoui         Ahmed Ressam
Zacarias Moussaoui         Abu Qatada
Zacarias Moussaoui         Ramzi Bin al-Shibh
Zacarias Moussaoui         Mahamed Atta
Ahmed Ressam               Haydar Abu Doha
Mehdi Khammoun             Haydar Abu Doha
Essid Sami Ben Khemais     Haydar Abu Doha
Mehdi Khammoun             Essid Sami Ben Khemais
Mehdi Khammoun             Mohamed Bensakhria
...
;
```

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

To find the articulation points, use the following statements:

```
proc 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 formulating the attack.

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

node	artpoint
Djamal Beghal	1
Mamoun Darkazanli	1
Zacarias Moussaoui	1
Nawaf Alhazmi	1
Essid Sami Ben Khemais	1
Mohamed Atta	1

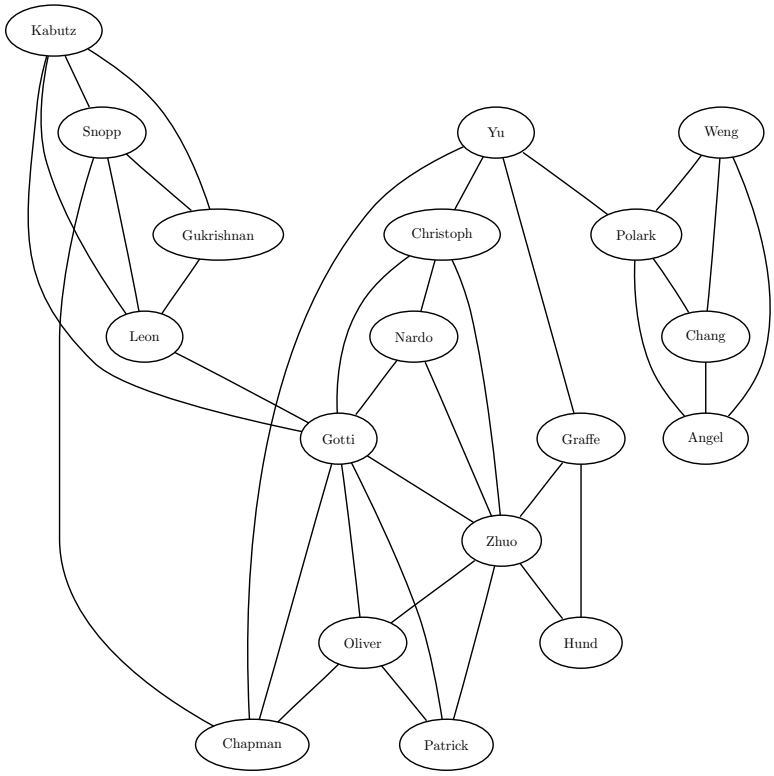
Example 2.2: Influence Centrality for Project Groups in a Research Department

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 2.91.

Figure 2.91 Project Groups in a Research Department



The link weights measure the reporting magnitude. In general, the higher the weight, the higher the contribution to the influence metric. Chapman is the director of the overall department, and Yu is the manager of a subgroup. The leads for 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:

```
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
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):

```
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: ------------------------------------------------------------------------------------------
NOTE: Running NETWORK.
NOTE: ------------------------------------------------------------------------------------------
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 0.344802 seconds.
NOTE: The data set MYCAS.NODESETOUT has 18 observations and 5 variables.

STATUS=OK  PROBLEM_TYPE=CENTRALITY  SOLUTION_STATUS=OK  CPU_TIME=0.10  REAL_TIME=0.34
```

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

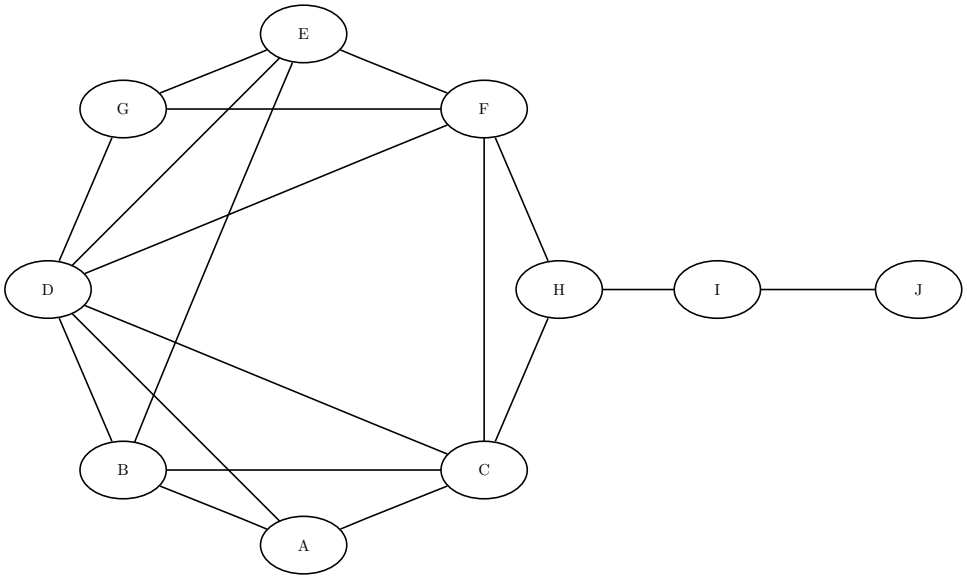
node	weight	centr_degree_out	centr_influence1_wt	centr_influence2_wt
Gotti	2	8	0.35714	1.57143
Zhuo	1	7	0.25000	1.17857
Oliver	2	4	0.21429	1.14286
Chapman	4	4	0.42857	1.10714
Christoph	2	4	0.17857	1.03571
Yu	3	4	0.32143	0.92857
Kabutz	1	4	0.14286	0.82143
Leon	1	4	0.14286	0.82143
Patrick	1	3	0.10714	0.82143
Snopp	2	4	0.21429	0.82143
Nardo	1	3	0.10714	0.78571
Graffe	1	3	0.14286	0.64286
Polark	2	4	0.17857	0.64286
Gukrishnan	1	3	0.10714	0.50000
Angel	1	3	0.10714	0.39286
Chang	1	3	0.10714	0.39286
Hund	1	2	0.07143	0.39286
Weng	1	3	0.10714	0.39286

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 2.92. This is an example of the well-known *kite network*, which was popularized by David Krackhardt (1990) for better understanding of social networks in the workplace.

Figure 2.92 Office Computer Network



Define the links data table as follows:

```
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.

```
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;
   merge mycas.NodeSetOutCentr mycas.NodeSetOutBiCC;
   by node;
run;
%put &_NETWORK_;
```

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

node	centr_degree_out	centr_close_unwt	centr_between_unwt	artpoint
C	5	0.64286	0.23148	0
F	5	0.64286	0.23148	0
D	6	0.60000	0.10185	0
H	3	0.60000	0.38889	1
E	4	0.52941	0.02315	0
B	4	0.52941	0.02315	0
A	3	0.50000	0.00000	0
G	3	0.50000	0.00000	0
I	2	0.42857	0.22222	1
J	1	0.31034	0.00000	0

Output 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

node	centr_degree_out	centr_close_unwt	centr_between_unwt	artpoint
H	3	0.60000	0.38889	1
C	5	0.64286	0.23148	0
F	5	0.64286	0.23148	0
I	2	0.42857	0.22222	1
D	6	0.60000	0.10185	0
E	4	0.52941	0.02315	0
B	4	0.52941	0.02315	0
A	3	0.50000	0.00000	0
G	3	0.50000	0.00000	0
J	1	0.31034	0.00000	0

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

from	to	centr_between_unwt
H	I	0.44444
C	H	0.29167
F	H	0.29167
I	J	0.25000
B	C	0.12963
E	F	0.12963
A	C	0.12500
F	G	0.12500
C	D	0.09259
D	F	0.09259
A	D	0.08333
D	G	0.08333
B	E	0.07407
C	F	0.07407
D	E	0.05093
B	D	0.05093
A	B	0.04167
E	G	0.04167

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 126, 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 59, 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: ------------------------------------------------------------------------------------------
NOTE: Running NETWORK.
NOTE: ------------------------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------------------------
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%     0.24
NOTE: Processing between/close centrality metrics used 0.24 seconds.
NOTE: Processing centrality metrics used 0.27 (cpu: 0.02) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.986986 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=0.18  REAL_TIME=0.99
```

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

node	centr_close_wt	centr_close_unwt	centr_between_wt	centr_between_unwt
Chang	0.44156	0.29310	0.00000	0.00000
Angel	0.44156	0.29310	0.00000	0.00000
Christoph	0.68456	0.48571	0.05882	0.11275
Gotti	0.81600	0.51515	0.20956	0.28444
Nardo	0.51777	0.42500	0.00000	0.00000
Yu	0.87179	0.50000	0.50000	0.41262
Zhuo	0.58286	0.47222	0.06618	0.15172
Chapman	0.88696	0.50000	0.44118	0.23235
Oliver	0.73913	0.44737	0.04044	0.02230
Patrick	0.50000	0.37778	0.00000	0.00000
Graffe	0.67105	0.43590	0.08088	0.06642
Hund	0.45133	0.36957	0.00000	0.00000
Gukrishnan	0.46575	0.32692	0.00000	0.00000
Leon	0.50746	0.38636	0.00000	0.03885
Kabutz	0.50746	0.38636	0.00000	0.03885
Snopp	0.75556	0.38636	0.16176	0.08088
Polark	0.69388	0.38636	0.30882	0.30882
Weng	0.44156	0.29310	0.00000	0.00000

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

from	to	weight	centr_between_wt	centr_between_unwt
Chang	Angel	1	0.00735	0.00735
Polark	Chang	1	0.11029	0.11029
Weng	Chang	1	0.00735	0.00735
Polark	Angel	1	0.11029	0.11029
Weng	Angel	1	0.00735	0.00735
Christoph	Gotti	1	0.02574	0.09620
Christoph	Nardo	1	0.04779	0.04412
Christoph	Yu	2	0.13603	0.15870
Christoph	Zhuo	1	0.03309	0.05147
Nardo	Gotti	1	0.05515	0.05147
Zhuo	Gotti	1	0.05515	0.10184
Gotti	Chapman	3	0.20221	0.09767
Gotti	Oliver	1	0.00000	0.03431
Gotti	Patrick	1	0.05882	0.06066
Leon	Gotti	1	0.07353	0.12586
Kabutz	Gotti	1	0.07353	0.12586
Nardo	Zhuo	1	0.02206	0.02941
Yu	Chapman	3	0.39706	0.25576
Graffe	Yu	2	0.18015	0.12402
Polark	Yu	2	0.41176	0.41176
Zhuo	Oliver	1	0.02574	0.03885
Zhuo	Patrick	1	0.02941	0.04412
Graffe	Zhuo	1	0.03676	0.08578
Zhuo	Hund	1	0.05515	0.07696
Oliver	Chapman	3	0.14338	0.07623
Snopp	Chapman	3	0.26471	0.16005
Oliver	Patrick	1	0.03676	0.02022
Graffe	Hund	1	0.06985	0.04804
Gukrishnan	Leon	1	0.00735	0.03431
Kabutz	Gukrishnan	1	0.00735	0.03431
Snopp	Gukrishnan	1	0.11029	0.05637
Kabutz	Leon	1	0.00735	0.00735
Snopp	Leon	1	0.03676	0.03517
Kabutz	Snopp	1	0.03676	0.03517
Weng	Polark	1	0.11029	0.11029

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

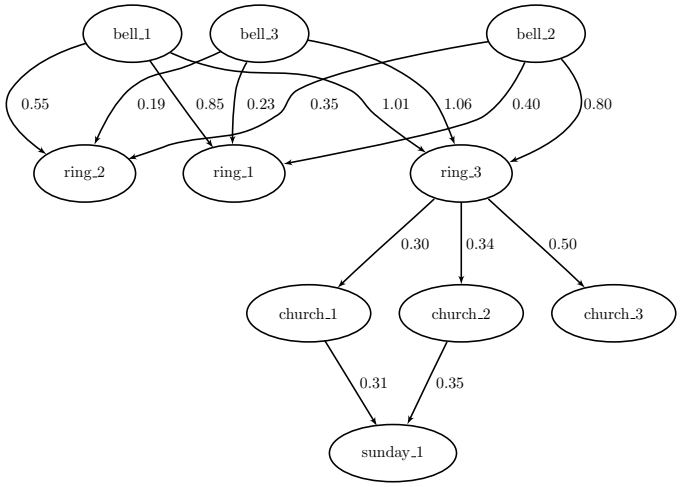
word "bass" as an example. It can mean either *a type of fish* (as in the sentence "I went fishing for sea bass") or *tones of low frequency* (as in the sentence "The bass part of the song is very moving").

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 2.93.

Figure 2.93 Eigenvector Centrality for Word Sense Disambiguation



To identify the correct senses, you run eigenvector centrality on the graph and select the highest-ranking sense for each word:

```
data 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.

Output 2.5.1 Eigenvector Centrality for Word Sense Disambiguation

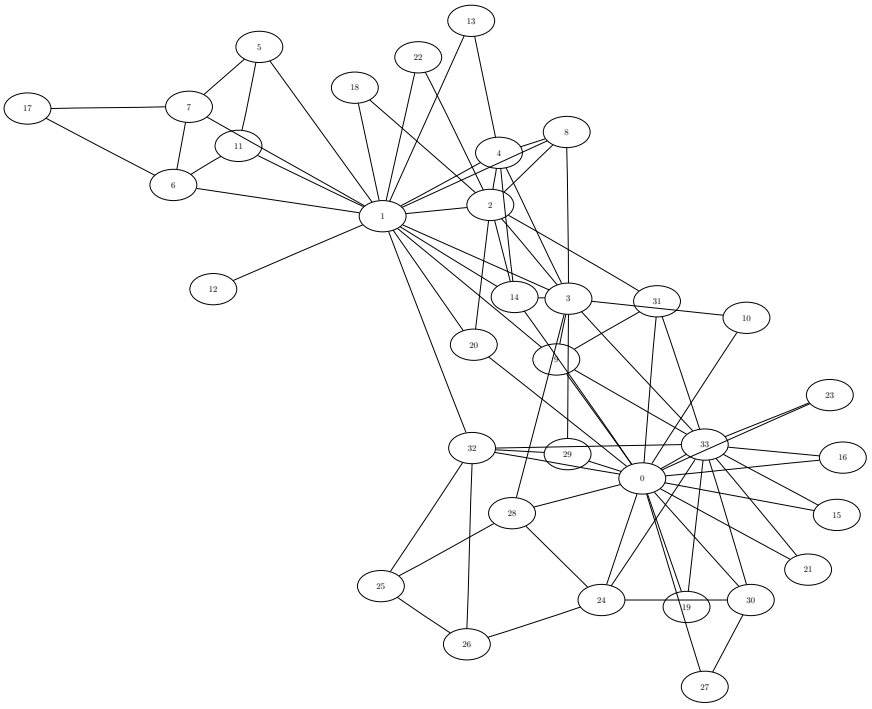
node	centr_eigen_wt
ring_3	1.00000
bell_1	0.77997
bell_3	0.59692
bell_2	0.53889
ring_1	0.48924
ring_2	0.35207
church_3	0.24081
church_2	0.17248
church_1	0.15222
sunday_1	0.05180

word	sense	node
bell	1	bell_1
church	3	church_3
ring	3	ring_3
sunday	1	sunday_1

Example 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 2.94.

Figure 2.94 Zachary's Karate Club Graph



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

```
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 74.

```
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

node	community_1	community_2
0	1	1
9	1	1
10	2	2
14	2	2
15	1	1
16	1	1
19	1	1
20	2	2
21	1	1
23	1	1
24	3	1
27	1	1
28	3	1
29	3	1
30	1	1
31	1	1
32	3	1

node	community_1	community_2
33	1	1
2	2	2
1	2	2
3	2	2
4	2	2
5	4	2
6	4	2
7	4	2
8	2	2
11	4	2
12	2	2
13	2	2
17	4	2
18	2	2
22	2	2
26	3	1
25	3	1

The column community_1 contains the community identifier of each node when the resolution value is 1.0; the column community_2 contains the community identifier of each node when the resolution value is 0.5. Different node colors are used to represent different communities in Figure 2.95 and Figure 2.96. As you can see from the figures, four communities at resolution 1.0 are merged into two communities at resolution 0.5.

Figure 2.95 Karate Club Communities (Resolution = 1.0)

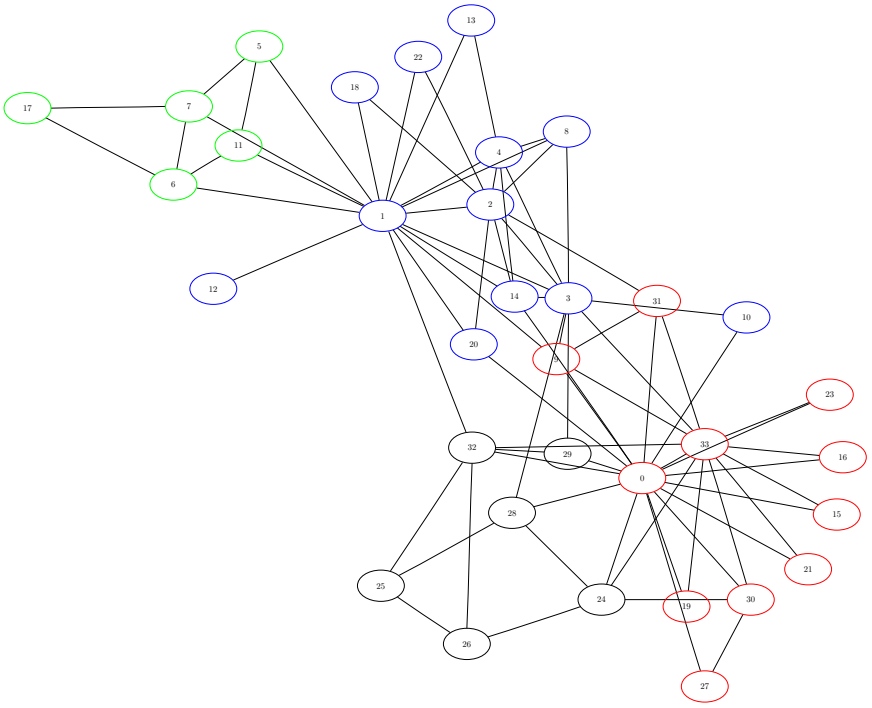
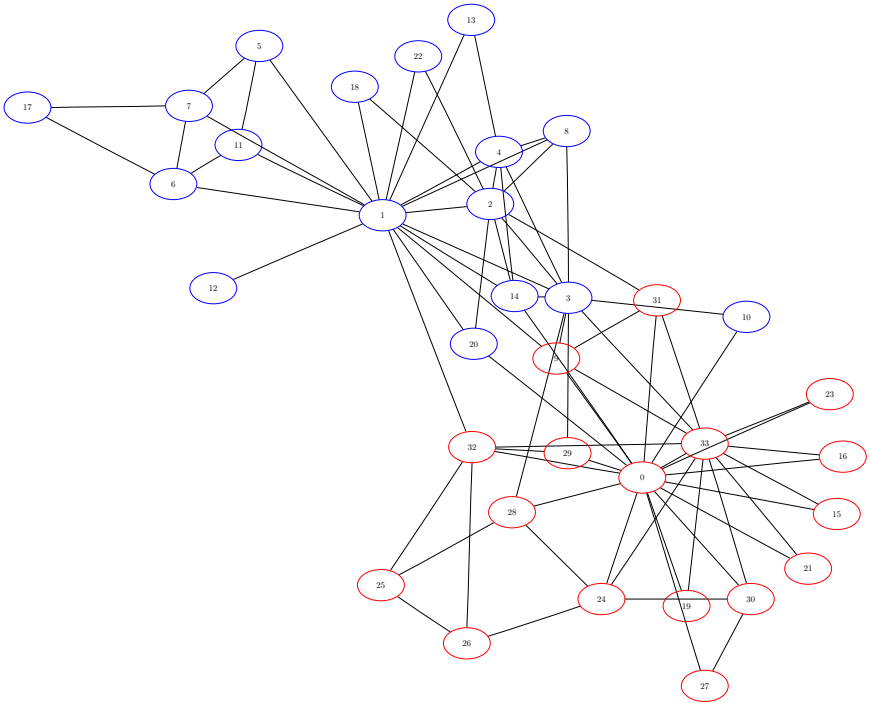


Figure 2.96 Karate Club Communities (Resolution = 0.5)



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

level	resolution	communities	modularity
1	1.0	4	0.41880
2	0.5	2	0.37179

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

level	resolution	community	nodes
1	1.0	1	11
1	1.0	2	12
1	1.0	3	6
1	1.0	4	5
2	0.5	1	17
2	0.5	2	17

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.

Output 2.6.4 Community Overlap Output

node	community	intensity
0	1	0.82353
0	2	0.17647
9	1	0.60000
9	2	0.40000
10	1	0.50000
10	2	0.50000
14	1	0.20000
14	2	0.80000
15	1	1.00000
16	1	1.00000
19	1	1.00000
20	1	0.33333
20	2	0.66667
21	1	1.00000
23	1	1.00000
24	1	1.00000
27	1	1.00000
28	1	0.75000
28	2	0.25000
29	1	0.66667
29	2	0.33333
30	1	1.00000
31	1	0.75000
31	2	0.25000
32	1	0.83333
32	2	0.16667
33	1	0.91667
33	2	0.08333
2	1	0.11111
2	2	0.88889
1	1	0.12500
1	2	0.87500
3	1	0.40000
3	2	0.60000
4	2	1.00000
5	2	1.00000
6	2	1.00000
7	2	1.00000
8	2	1.00000
11	2	1.00000
12	2	1.00000
13	2	1.00000
17	2	1.00000
18	2	1.00000
22	2	1.00000
26	1	1.00000
25	1	1.00000

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

level	resolution	from_community	to_community	link_weight
1	1.0	1	2	7
1	1.0	1	3	7
1	1.0	2	3	3
1	1.0	2	4	4
2	0.5	1	2	10

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 139. 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.

Output 2.7.1 Community Nodes Output

node	community_1
0	4
9	2
10	6
14	6
15	4
16	4
19	4
20	7
21	4
23	4
24	9
27	3
28	9
29	10
30	3
31	2
32	10

node	community_1
33	4
2	7
1	8
3	6
4	5
5	1
6	1
7	1
8	5
11	1
12	8
13	5
17	1
18	7
22	8
26	9
25	9

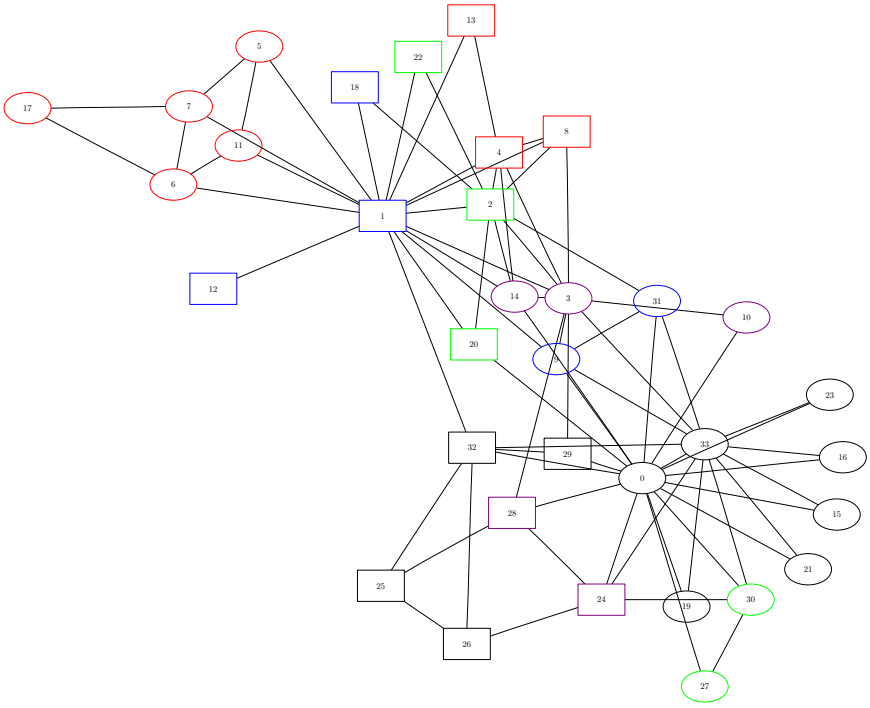
The output data table mycas.CommOut contains the number of nodes in each community, as shown in Output 2.7.2.

Output 2.7.2 Community Number of Nodes Output

level	resolution	community	nodes
1	1	1	5
1	1	2	2
1	1	3	2
1	1	4	7
1	1	5	3
1	1	6	3
1	1	7	3
1	1	8	3
1	1	9	4
1	1	10	2

The community graph is shown in Figure 2.97, with different node shapes and colors representing different communities.

Figure 2.97 Karate Club Recursive Communities



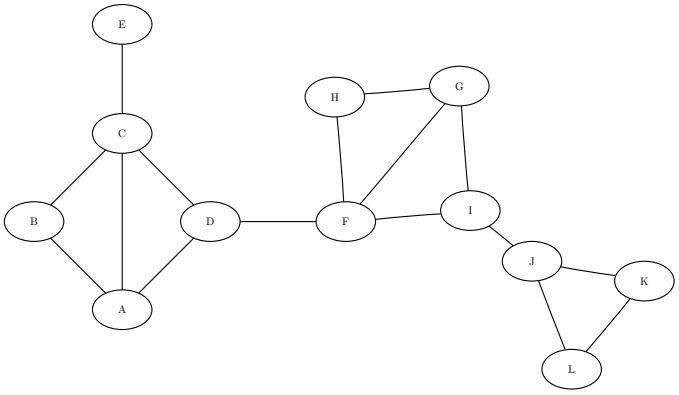
As you can see from Output 2.7.2, Community 4, whose nodes are drawn as black ellipses in Figure 2.97, contains seven nodes even though the maximum number of nodes in any community is set to 5. This is because Community 4 has a symmetric shape: Nodes 0 and 33 are in the center, and they symmetrically 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 2.8.1.

Output 2.8.1 Undirected Graph



The following statements create the data table mycas.LinkSetIn:

```
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
;
```

First, call the community detection method as follows:

```
proc network
   links     = mycas.LinkSetIn
   outNodes  = mycas.OutNodesComms
   outLinks  = mycas.OutLinksComms;
   community;
run;
```

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 2.8.2.

Output 2.8.2 Nodes for the Communities of a Simple Undirected Graph

node	community_1
A	1
B	1
C	1
D	1
E	1
F	2
G	2
H	2
I	2
J	3
K	3
L	3

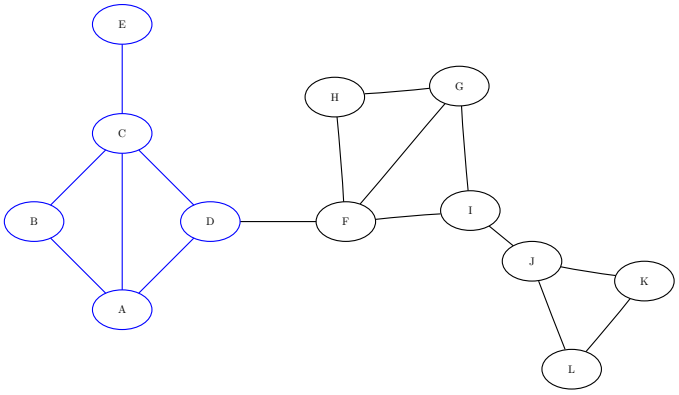
The data table that contains the assignment of links to communities, mycas.OutLinksComms, is shown in Output 2.8.3.

Output 2.8.3 Links for the Communities of a Simple Undirected Graph

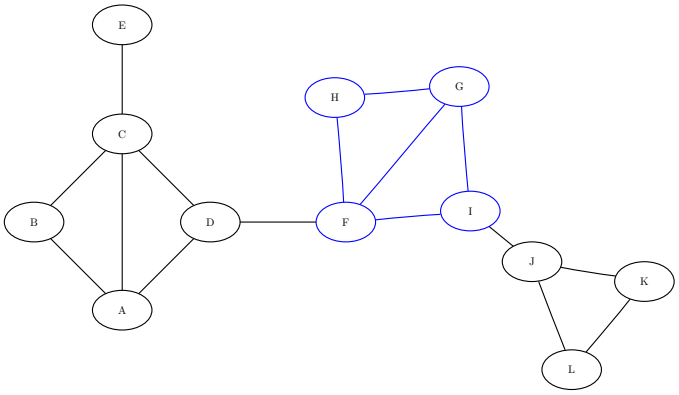
from	to	community_1
A	B	1
A	C	1
A	D	1
B	C	1
C	D	1
C	E	1
D	F	.
F	G	2
F	H	2
F	I	2
G	H	2
G	I	2
I	J	.
J	K	3
J	L	3
K	L	3

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 2.8.4 through Figure 2.8.6.

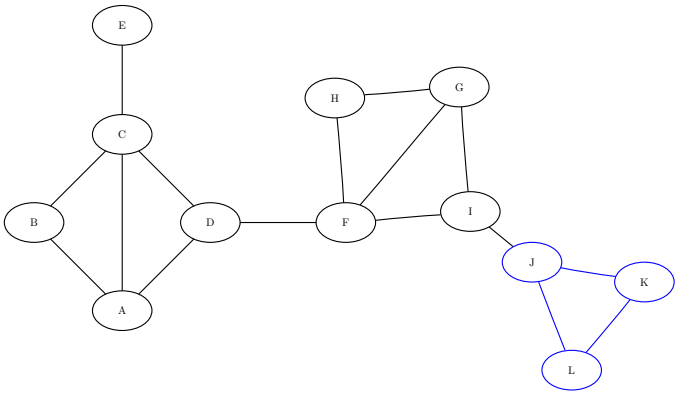
Output 2.8.4 Subgraph $C^1 = \{A, B, C, D, E\}$



Output 2.8.5 Subgraph $C^2 = \{F, G, H, I\}$



Output 2.8.6 Subgraph $C^3 = \{J, K, L\}$



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 will be done in parallel across machines and threads (depending on your server configuration).

```
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 2.8.7.

Output 2.8.7 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 0.00 (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 0.00 (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 0.00 (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(magala)' has 3 rows and 4 columns.
NOTE: The CAS Table 'SOLUTIONSUMMARY' in caslib 'CASUSERHDFS(magala)' has 3 rows and 5 columns.
NOTE: The Cloud Analytic Services server processed the request in 0.438229 seconds.
NOTE: The data set MYCAS.NODESETOUT has 12 observations and 8 variables.

STATUS=OK  PROBLEM_TYPE=CENTRALITY  CPU_TIME=0.28  REAL_TIME=0.44
```

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 2.8.8 Problem Summary by Community

community_1	numNodes	numLinks	graphDirection
1	5	6	Undirected
2	4	5	Undirected
3	3	3	Undirected

The output table mycas.SolutionSummary contains a solution summary for the processing on each of the induced subgraphs.

Output 2.8.9 Solution Summary by Community

community_1	problemType	status	cpuTime	realTime
1	Centrality	OK	0	.001883984
2	Centrality	OK	0	.001708984
3	Centrality	OK	0	.001863956

The centrality results (by community) are shown in Output 2.8.10.

Output 2.8.10 Centrality for All Induced Subgraphs

community_1=1

node	centr_degree_out	centr_eigen_unwt	centr_close_unwt	centr_between_unwt	centr_influence1_unwt	centr_influence2_unwt
B	2	0.70711	0.66667	0.00000	0.4	1.4
C	4	1.00000	1.00000	0.58333	0.8	1.6
D	2	0.70711	0.66667	0.00000	0.4	1.4
E	1	0.37236	0.57143	0.00000	0.2	0.8
A	3	0.89897	0.80000	0.08333	0.6	1.6

community_1=2

node	centr_degree_out	centr_eigen_unwt	centr_close_unwt	centr_between_unwt	centr_influence1_unwt	centr_influence2_unwt
G	3	1.00000	1.00	0.16667	0.75	1.75
H	2	0.78078	0.75	0.00000	0.50	1.50
I	2	0.78078	0.75	0.00000	0.50	1.50
F	3	1.00000	1.00	0.16667	0.75	1.75

community_1=3

node	centr_degree_out	centr_eigen_unwt	centr_close_unwt	centr_between_unwt	centr_influence1_unwt	centr_influence2_unwt
K	2	1	1	0	0.66667	1.33333
L	2	1	1	0	0.66667	1.33333
J	2	1	1	0	0.66667	1.33333

Example 2.9: 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. Then, a second user declares that B is a DUP of C, and a third user declares that D is a DUP of A. You now have a circular dependency, and no primary bug is defined that the development team should 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 89. 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. The existence of these chains can be identified by calculating the transitive closure of the directed graph that is defined by the DUP links.

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

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

```
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
S0350596 S0346582 DUPTO 21MAR06:00:00:00
S0539744 S0643230 DUPTO 10MAY10:00:00:00
S0575748 S0643230 DUPTO 15JUN10:00:00:00
S0629984 S0643230 DUPTO 01JUN10:00:00:00
;
```

The following statements calculate cycles in addition to the transitive closure of the graph G that is defined by the duplicated defects in 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.

```
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 2.9.1.

Output 2.9.1 PROC NETWORK Log: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System

```
NOTE: ------------------------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------------------------
NOTE: Running NETWORK.
NOTE: ------------------------------------------------------------------------------------------
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 16.
NOTE: The number of links in the input graph is 12.
NOTE: Processing cycle detection.
NOTE: Processing cycle detection using the backtrack algorithm.
NOTE: The algorithm found 1 cycles.
NOTE: Processing cycle detection used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.061105 seconds.
NOTE: The data set MYCAS.CYCLES has 4 observations and 3 variables.

STATUS=OK  PROBLEM_TYPE=CYCLE  SOLUTION_STATUS=OK  NUM_CYCLES=1  CPU_TIME=0.08  REAL_TIME=0.06

NOTE: ------------------------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------------------------
NOTE: Running NETWORK.
NOTE: ------------------------------------------------------------------------------------------
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 16.
NOTE: The number of links in the input graph is 12.
NOTE: Processing the transitive closure.
NOTE: Processing the transitive closure used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.057024 seconds.
NOTE: The data set MYCAS.TRANSCLOSURE has 20 observations and 2 variables.

STATUS=OK  PROBLEM_TYPE=TRANSITIVECLOSURE  SOLUTION_STATUS=OK  CPU_TIME=0.09  REAL_TIME=0.06
NOTE: Table WORK.CHAINS created, with 5 rows and 2 columns.
```

The output data table mycas.Cycles contains one case of a circular dependency in which the DUPs start and end at S0152674.

Output 2.9.2 Cycle in Bug Tracking System

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

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

Output 2.9.3 Chains in Bug Tracking System

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

Example 2.10: 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:

```
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.

```
proc network
   links        = mycas.Patents
   outNodes     = mycas.OutNodes;
   connectedComponents
      algorithm = parallel;
run;
%put &_NETWORK_;
```

The progress of the procedure is shown in Output 2.10.1.

Output 2.10.1 PROC NETWORK Log: Connected Components for US Patent Citations

```
NOTE: ---------------------------------------------------------------------------------------
NOTE: Running NETWORK.
NOTE: ---------------------------------------------------------------------------------------
WARNING: The graph contains 1 self links that are ignored.
NOTE: The number of nodes in the input graph is 3774768.
NOTE: The number of links in the input graph is 16518947.
NOTE: The number of singleton nodes in the input graph is 1.
NOTE: Processing connected components using 4 machines.
NOTE: The graph has 3627 connected components.
NOTE: Processing connected components used 7.59 (cpu: 5.39) seconds.
NOTE: The Cloud Analytic Services server processed the request in 10.018979 seconds.
NOTE: The data set MYCAS.OUTNODES has 3774768 observations and 2 variables.

STATUS=OK  PROBLEM_TYPE=CONNECTEDCOMPONENTS  SOLUTION_STATUS=OK  NUM_COMPONENTS=3627
CPU_TIME=47.99  REAL_TIME=10.02
```

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 2.10.2. 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 2.10.2 Ten Largest Components for US Patent Citations

Obs	concomp	count
1	1	3764117
2	446	19
3	242	16
4	25	15
5	345	14
6	169	14
7	263	14
8	1830	14
9	239	13
10	158	13

Example 2.11: 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 needed to find all-pairs shortest paths is enormous. In addition, the storage space that is needed 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 2.11.1.

Output 2.11.1 PROC NETWORK Log: Shortest Paths of the NY Road Network

```
NOTE: ------------------------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------------------------
NOTE: Running NETWORK.
NOTE: ------------------------------------------------------------------------------------------
NOTE: ------------------------------------------------------------------------------------------
WARNING: The graph contains 368796 duplicate links that are ignored.
NOTE: The number of nodes in the input graph is 264346.
NOTE: The number of links in the input graph is 365050.
NOTE: Processing the shortest paths problem using 32 threads on each of 130 machines.
NOTE: Processing the shortest paths problem between 264346 source nodes and 264346 sink nodes.
                                                 Real
      Algorithm              Sources   Complete     Time
      shortestpath               320         0%    10.63
      shortestpath             41045        15%    20.19
      shortestpath             82560        31%    30.05
      shortestpath            124225        46%    40.25
      shortestpath            163990        62%    50.01
      shortestpath            202910        76%    60.05
      shortestpath            241356        91%    70.06
      shortestpath            264151        99%    80.74
      shortestpath            264346       100%    91.96
NOTE: Processing the shortest paths problem used 93.14 (cpu: 231586.67) seconds.
NOTE: The Cloud Analytic Services server processed the request in 99.501308 seconds.
NOTE: The data set MYCAS.SHORTPATHSUMMARY has 104263396 observations and 3 variables.
NOTE: The data set MYCAS.SHORTPATHPATHS has 1419295895 observations and 6 variables.

STATUS=OK  PROBLEM_TYPE=SHORTESTPATH  SOLUTION_STATUS=OK  NUM_PATHS=104263396
CPU_TIME=232788.78  REAL_TIME=99.50
```

Notice that the resulting output data tables, mycas.shortPathSummary and mycas.shortPathPaths, are large distributed data tables.

References

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

Batagelj, V., and Zaversnik, M. (2003). "An O(m) Algorithm for Cores Decomposition of Networks." *Computing Research Repository* cs.DS/0310049.

Blondel, V. D., Guillaume, J. L., Lambiotte, R., and Lefebvre, E. (2008). "Fast Unfolding of Communities in Large Networks." *Journal of Statistical Mechanics: Theory and Experiment* 10:10000–10014.

Boitmanis, K., Freivalds, K., Ledins, P., and Opmanis, R. (2006). "Fast and Simple Approximation of the Diameter and Radius of a Graph." In *Experimental Algorithms*, vol. 4007, edited by C. Alvarez and M. Serna, 98–108. Berlin: Springer-Verlag. http://dx.doi.org/10.1007/11764298_9.

Bron, C., and Kerbosch, J. (1973). "Algorithm 457: Finding All Cliques of an Undirected Graph." *Communications of the ACM* 16:48–50.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to Algorithms*. Cambridge, MA, and New York: MIT Press and McGraw-Hill.

Demetrescu, C. (2010). "9th DIMACS Implementation Challenge—Shortest Paths." The data are available at http://www.dis.uniroma1.it/challenge9/download.shtml.

Fowler, J. H., and Joen, S. (2008). "The Authority of Supreme Court Precedent." *Social Networks* 30:16–30. http://jhfowler.ucsd.edu/judicial.htm.

Google (2011). "Google Maps." Accessed March 16, 2011. http://maps.google.com.

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

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

Kleinberg, J. (1998). "Authoritative Sources in a Hyperlinked Environment." In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, 668–677. Philadelphia: SIAM.

Krackhardt, D. (1990). "Assessing the Political Landscape: Structure, Cognition, and Power in Organizations." *Administrative Science Quarterly* 35:342–369.

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

Lancichinetti, A., and Fortunato, S. (2009). "Community Detection Algorithms: A Comparative Analysis." *Physical Review E* 80:056117–056128.

Landes, W. M., and Posner, R. A. (1976). "Legal Precedent: A Theoretical and Empirical Analysis." *Journal of Law and Economics* 19:249–307.

Leskovec, J. (2014). "SNAP: Stanford Network Analysis Project." The data are available at https://snap.stanford.edu/data/index.html.

Liu, H., and Wang, J. (2006). "A New Way to Enumerate Cycles in Graph." In *Proceedings of the Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services*, 57–59.

Mihalcea, R. (2005). "Unsupervised Large-Vocabulary Word Sense Disambiguation with Graph-Based Algorithms for Sequence Data Labeling." In *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*, 411–418. Vancouver.

Newman, M. E. J. (2010). *Networks: An Introduction*. Oxford: Oxford University Press.

Raghavan, U. N., Albert, R., and Kumara, S. (2007). "Near Linear Time Algorithm to Detect Community Structures in Large-Scale Networks." *Physical Review E* 76:36106–36117.

Rochat, Y. (2009). "Closeness Centrality Extended to Unconnected Graphs: The Harmonic Centrality Index." Paper presented at Sixth Applications of Social Network Analysis Conference, Zurich. http://infoscience.epfl.ch/record/200525/files/[EN]ASNA09.pdf.

Ronhovde, P., and Nussinov, Z. (2010). "Local Resolution-Limit-Free Potts Model for Community Detection." *Physical Review E* 81:46114–46129.

Sinha, R., and Mihalcea, R. (2007). "Unsupervised Graph-Based Word Sense Disambiguation Using Measures of Word Semantic Similarity." In *Proceedings of the IEEE International Conference on Semantic Computing*, 363–369. Los Alamitos, CA: IEEE Computer Society Press.

Sleijpen, G. L. G., and van der Vorst, H. A. (2000). "A Jacobi-Davidson Iteration Method for Linear Eigenvalue Problems." *SIAM Review* 42:267–293.

Tarjan, R. E. (1972). "Depth-First Search and Linear Graph Algorithms." *SIAM Journal on Computing* 1:146–160.

Traag, V. A., Van Dooren, P., and Nesterov, Y. (2011). "Narrow Scope for Resolution-Limit-Free Community Detection." *Physical Review E* 84:016114 (1–9). http://dx.doi.org/10.1103/PhysRevE.84.016114.

Zachary, W. W. (1977). "An Information Flow Model for Conflict and Fission in Small Groups." *Journal of Anthropological Research* 33:452–473.

Index