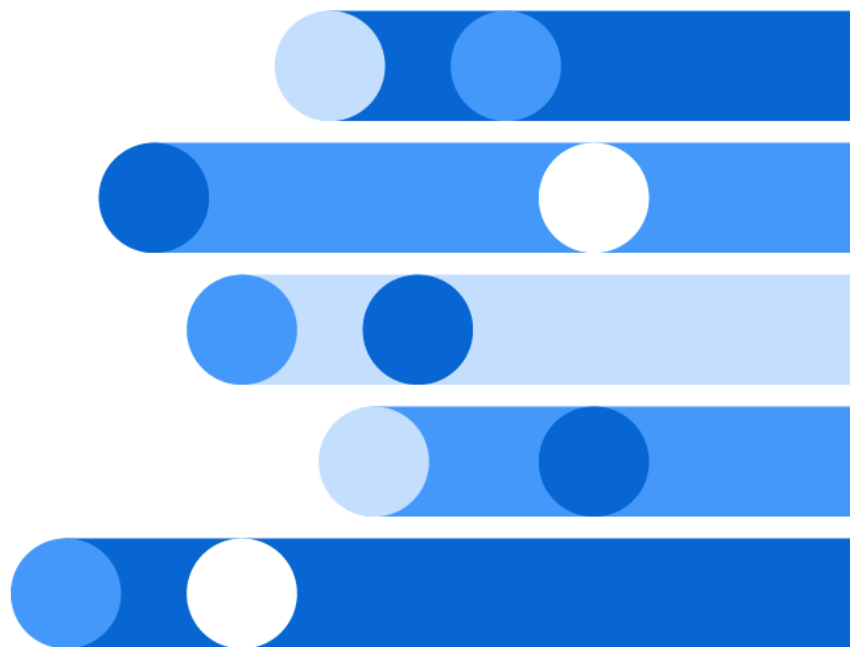




SAS[®] Viya[®] NETWORK Procedure

2025.11*



* This document might apply to additional versions of the software. Open this document in SAS Help Center and click on the version in the banner to see all available versions.



The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2025. *SAS® Viya®: NETWORK Procedure*. Cary, NC: SAS Institute Inc.

SAS® Viya®: NETWORK Procedure

Copyright © 2025, SAS Institute Inc., Cary, NC, USA

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

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

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

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

November 2025

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

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

SAS software may be provided with certain third-party software, including but not limited to open source software, which is licensed under its applicable third-party software license agreement. For license information about third-party software distributed with SAS software, refer to [Third-Party Software Reference | SAS Support](#).

Contents

Chapter 1.	Introduction	1
Chapter 2.	NETWORK Procedure	7

Chapter 1

Introduction

Contents

Overview of the NETWORK Procedure	1
About This Book	2
Chapter Organization	2
Using CAS Sessions and CAS Engine Librefs	2
Loading a SAS Data Set onto a CAS Server	3
Typographical Conventions	4
Options Used in Examples	5
Where to Turn for More Information	5
Online Documentation	5
SAS Technical Support Services	5

Overview of the NETWORK Procedure

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

About This Book

This book assumes that you are familiar with Base SAS software and with the books *SAS Programmers Guide: Essentials* 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.

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 mylib cas sessref=mysess;
```

The CAS statement creates the CAS session named `mysess`, and the LIBNAME statement creates the mylib 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: User's Guide*. 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, `mylib` 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 mylib.Sample;
  input from $ to $ @@;
  datalines;
A B A C B C
;
```

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

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

```
data Sample;
  input from $ to $ @@;
  datalines;
A B A C B C
;
data mylib.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: User's Guide*.

The mylib 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 mylib caslib:

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

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

```
proc delete data = mylib.Sample;
run;
```

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

Typographical Conventions

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

roman	is the standard type style used for most text.
UPPERCASE ROMAN	is used for SAS statements, options, and other SAS language elements when they appear in text. However, you can enter these elements in your own SAS programs in lowercase, uppercase, or a mixture of the two.
UPPERCASE BOLD	is used in the “Syntax” sections’ initial lists of SAS statements and options.
<i>oblique</i>	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.
<i>italic</i>	is used for terms that are defined in text, for emphasis, and for references to publications.
monospace bold	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 for an HTML ODS destination as follows:

```
ods html style=HTMLBlue;
```

You can also specify a style template for a PDF ODS destination as follows:

```
ods pdf style=PearlJ;
```

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

NETWORK Procedure

Contents

Overview: NETWORK Procedure	9
Getting Started: NETWORK Procedure	10
Road Network Shortest Path	11
Authority in US Supreme Court Precedent	14
Syntax: NETWORK Procedure	16
Functional Summary	17
PROC NETWORK Statement	25
BICONNECTEDCOMPONENTS Statement	29
BY Statement	30
CENTRALITY Statement	30
CLIQUE Statement	34
COMMUNITY Statement	35
CONNECTEDCOMPONENTS Statement	38
CORE Statement	39
CYCLE Statement	39
DISPLAY Statement	41
DISPLAYOUT Statement	42
LINKSQUERYVAR Statement	43
LINKSVAR Statement	44
LOADGRAPH Statement	45
NODESIMILARITY Statement	45
NODESQUERYVAR Statement	48
NODESSUBSETVAR Statement	49
NODESVAR Statement	49
PATH Statement	50
PATTERNMATCH Statement	51
PROJECTION Statement	54
REACH Statement	56
SHORTESTPATH Statement	57
SUMMARY Statement	59
TOPOLOGICALSORT Statement	60
TRANSITIVECLOSURE Statement	61
UNLOADGRAPH Statement	61
Details: NETWORK Procedure	61
Graph Input and Output Data	61
Determinism	78

Execution Modes and Data Movement	78
Persistent Data Structures (In-Memory Graphs)	81
Numeric Limitations	87
Size Limitations	88
Common Notation and Assumptions	89
Examples Statement and Option Cross-Reference	90
Biconnected Components and Articulation Points	94
Centrality	100
Clique Enumeration	123
Community Detection	128
Connected Components	137
Core Decomposition	142
Cycle Enumeration	146
Node Similarity	152
Path Enumeration	164
Pattern Matching	168
Network Projection	212
Reach (Ego) Network	219
Shortest Path	226
Summary Statistics	246
Topological Sort	254
Transitive Closure	256
Macro Variable <code>_NETWORK_</code>	258
ODS Table Names	261
The <code>%PlotGraph</code> Visualization Macro	262
Examples: NETWORK Procedure	268
Example 2.1: Articulation Points in a Terrorist Network	268
Example 2.2: Influence Centrality for Project Groups in a Research Department	271
Example 2.3: Betweenness and Closeness Centrality for Computer Network Topology	275
Example 2.4: Betweenness and Closeness Centrality for Project Groups in a Research Department	277
Example 2.5: Eigenvector Centrality for Word Sense Disambiguation	281
Example 2.6: Community Detection on Zachary's Karate Club Data	283
Example 2.7: Recursive Community Detection on Zachary's Karate Club Data	289
Example 2.8: Centrality Metrics for an Undirected Graph by Community	290
Example 2.9: Cycle Enumeration for Kidney Donor Exchange	295
Example 2.10: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System	300
Example 2.11: Reach Networks for Computing the Market Coverage of a Terrorist Network	303
Example 2.12: Connected Components for US Patent Citations	305
Example 2.13: Shortest Paths of the New York Road Network	307
Example 2.14: Shortest Path in a Road Network by Date and Time	309
Example 2.15: Pattern Matching in a Social Network	312

Example 2.16: Detection of Value-Added Tax Carousel Fraud	321
Example 2.17: Node Similarity for Link Prediction	335
Example 2.18: A Node Similarity Recipe Recommendation Engine	341
Example 2.19: Node Embeddings for Zachary’s Karate Club Data	344
Example 2.20: Node Embeddings for European Roads Data	349
Example 2.21: Learning Word Embeddings from Synonym Pairs	355
Example 2.22: Using Projection to Recommend Recipe Ingredient Pairings	359
References	361

Overview: NETWORK Procedure

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

Networks also appear explicitly and implicitly in many other application contexts. Networks are often constructed from certain relationships that are based on natural co-occurrence; examples are 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 each interaction is modeled as a weight on the corresponding link of the resulting network.

To support the myriad ways in which networks appear in data mining, the NETWORK procedure makes no assumptions about the context or application from which the network arises. PROC NETWORK provides a number of network analysis algorithms (listed in Table 2.1) that take an abstract graph or network as input, help explain the 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
Node similarity	NODESIMILARITY
Path enumeration	PATH
Pattern matching	PATTERNMATCH
Network projection	PROJECTION
Reach (ego) networks	REACH
Shortest path	SHORTESTPATH
Graph summary	SUMMARY
Topological sorting	TOPOLOGICALSORT
Transitive closure	TRANSITIVECLOSURE

As input, the NETWORK procedure expects *graph* $G = (N, E)$, which is defined over a set N of nodes and a set E of links. A *node* is an abstract representation of some entity (or object), and a *link* defines a relationship (or connection) between two nodes. The terms *node* and *vertex* are interchangeable in describing an entity. The term *link* is interchangeable with the term *edge* or *arc* in describing a connection. Similarly, the terms *graph* and *network* are interchangeable.

Getting Started: NETWORK Procedure

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

This section presents two introductory examples for getting started with the NETWORK procedure. For more information about the expected input formats and the available algorithms, see the sections “[Details: NETWORK Procedure](#)” on page 61 and “[Examples: NETWORK Procedure](#)” on page 268.

Road Network Shortest Path

Consider the following road network between a SAS employee's home in Raleigh, North Carolina, and SAS headquarters nearby in Cary. In this road network (graph), the links are the roads and the nodes are intersections of the roads. For each road, you assign a *link attribute* in the variable `time_to_travel` to describe the number of minutes that it takes to drive from one node to another. The following data were collected using Google Maps (Google 2011), which gives an approximate number of minutes to travel between two nodes based on the length of the road and the typical speed during normal traffic patterns. These statements assume that the CAS engine libref is named `mylib`, but you can substitute any appropriately defined CAS engine libref.

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

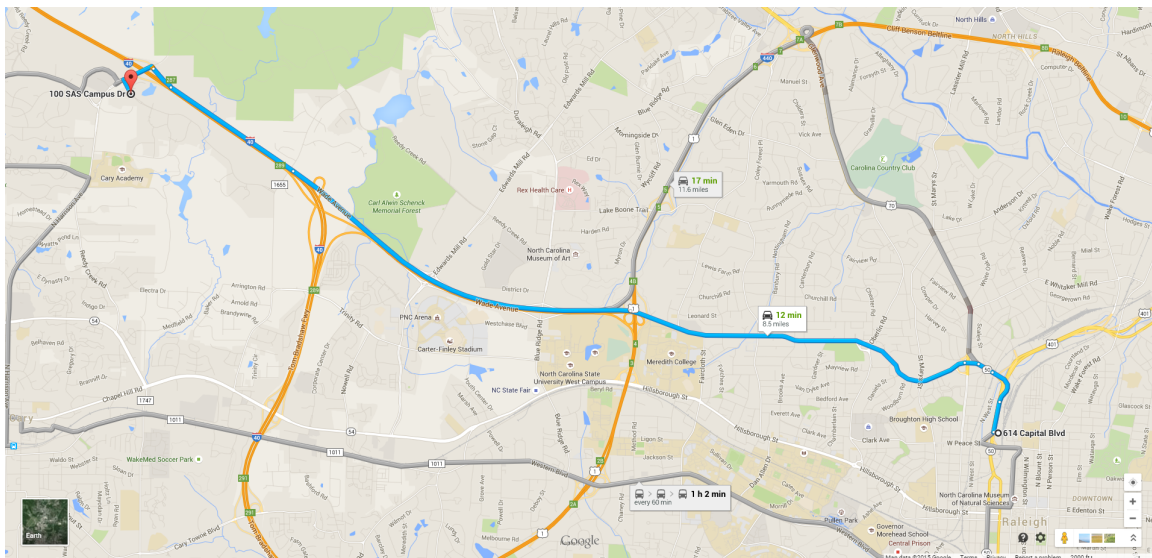
Using PROC NETWORK, you want to find the route that yields the shortest path between home (614 Capital Boulevard) and SAS headquarters (SAS Campus Drive). This can be done using the `SHORTESTPATH` statement as follows:

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

For more information about shortest path algorithms in PROC NETWORK, see the section “[Shortest Path](#)” on page 226. [Figure 2.1](#) displays the output data table `mylib.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 mylib.LinkSetInRoadNC5pm;
  input start_inter $1-20 end_inter $21-40 miles miles_per_hour;
  time_to_travel = miles * 1/miles_per_hour * 60;
  datalines;
614CapitalBlvd      Capital/WadeAve      0.6  25
614CapitalBlvd      Capital/US70W      0.6  25
614CapitalBlvd      Capital/US440W     3.0  45
Capital/WadeAve     WadeAve/RaleighExpy 3.0  25 /*high traffic*/
Capital/US70W       US70W/US440W      3.2  60
US70W/US440W       US440W/RaleighExpy 2.7  60
Capital/US440W      US440W/RaleighExpy 6.7  60
US440W/RaleighExpy RaleighExpy/US40W  3.0  60
WadeAve/RaleighExpy RaleighExpy/US40W  3.0  60
RaleighExpy/US40W  US40W/HarrisonAve  1.3  55
US40W/HarrisonAve  SASCampusDrive    0.5  25
;
```

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

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

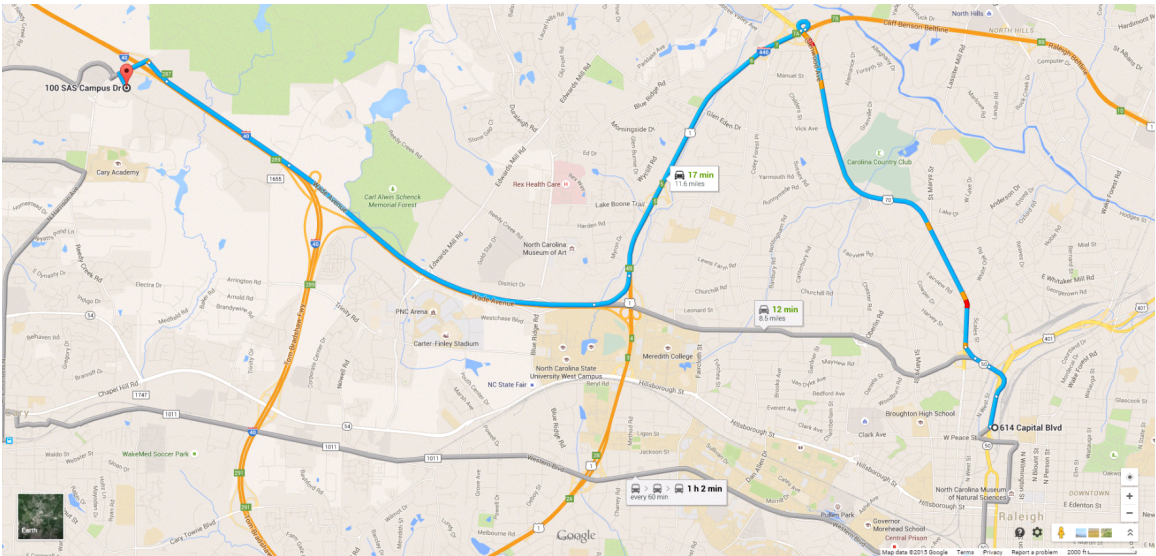
Now, the output data table mylib.ShortPath, shown in Figure 2.3, shows the best route for going home. Because the traffic on Wade Avenue is usually heavy at this time of day, the best route home is different from the best route to work.

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

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

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

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

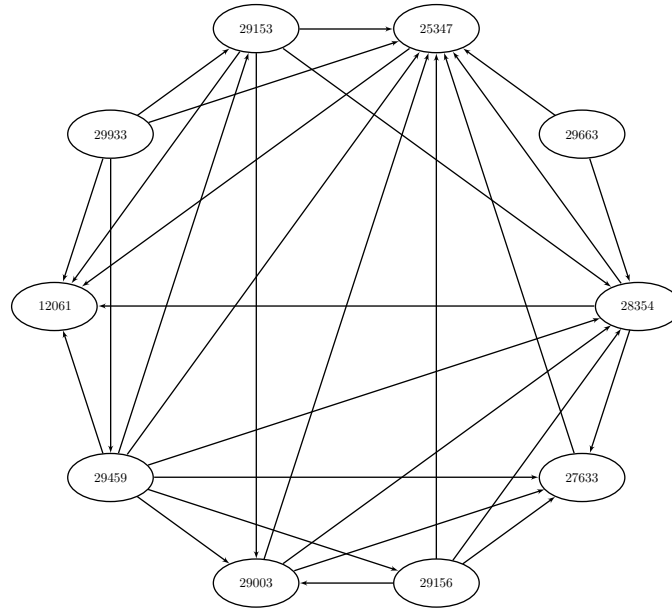


Authority in US Supreme Court Precedent

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 117. [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 `mylib.Cases` stores a mapping between the case name and the case identifier:

```
data mylib.Cases;
  length case_name $80;
  input case_id case_name $ 7-80;
  datalines;
12061 Jacobson v. Massachusetts, 197 U.S. 11 (1905)
25347 Roe vs. Wade, 410 U.S. 113 (1973)
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 `mylib.LinkSetInCourt` provides the citation network between case identifiers:

```

data mylib.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
  nodes      = mylib.Cases
  links      = mylib.LinkSetInCourt
  outNodes   = mylib.NodeSetOut;
  nodesVar
    node      = case_id
    vars      = (case_name);
  linksVar
    from      = from_case
    to        = to_case;
  centrality
    auth      = unweight;
run;

```

The output data table `mylib.NodeSetOut`, shown in [Figure 2.6](#), contains the authority score for each case sorted in descending order. 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	case_name	centr_auth_unwt
25347	Roe vs. Wade, 410 U.S. 113 (1973)	1.00000
28354	Thornburgh vs. American College, 476 U.S. 747 (1986)	0.72262
12061	Jacobson v. Massachusetts, 197 U.S. 11 (1905)	0.61717
27633	Akron vs. Akron Cntr for Repro-Health, 462 U.S. 416 (1983)	0.59831
29003	Webster vs. Repro-Health Services, 492 U.S. 490 (1989)	0.50930
29153	Cruzan v. Director, MO Dept of Health, 497 U.S. 261 (1990)	0.31742
29156	Hodgson v. Minnesota, 497 U.S. 417 (1990)	0.20968
29459	Planned Parenthood of SE PA vs. Casey, 505 U.S. 833 (1992)	0.10775
29155	Georgia v. South Carolina, 497 U.S. 376 (1990)	0.00000
29663	Madsen v. Women's Health Ctr., 512 U.S. 753 (1994)	0.00000
29933	Wash. v. Glucksberg, 521 U.S. 702 (1997)	0.00000

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

Syntax: NETWORK Procedure

PROC NETWORK statements are divided into four main categories:

PROC Statement

PROC NETWORK <options> ;

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

Data Statements

LOADGRAPH ;
UNLOADGRAPH ;

Data statements determine how in-memory graphs are used by PROC NETWORK.

Variables Statements

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

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

Algorithm Statements

```

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

```

Algorithm 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 > ;

```

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

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

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	Option
PROC NETWORK Statement	
Input	
Specifies the in-memory graph to use	GRAPH=
Specifies the links data table	LINKS=
Specifies the links data table to query	LINKSQUERY=
Specifies the nodes data table	NODES=
Specifies the nodes data table to query	NODESQUERY=
Specifies the nodes subset data table	NODESSUBSET=

Table 2.2 continued

Description	Option
Output	
Specifies the output data table to contain summary information about in-memory graphs	OUTGRAPHLIST=
Specifies the links output data table	OUTLINKS=
Specifies the nodes output data table	OUTNODES=
Options	
Specifies whether to enforce determinism	DETERMINISTIC=
Specifies the graph direction	DIRECTION=
Specifies whether to use a distributed graph	DISTRIBUTED=
Specifies the index offset for identifiers	INDEXOFFSET=
Specifies the desired frequency (in number of seconds) between log entries	LOGFREQTIME=
Specifies the overall log level	LOGLEVEL=
Specifies whether to include multilinks	MULTILINKS=
Specifies the maximum number of threads to use for multithreaded processing	NTHREADS=
Specifies whether to include self-links	SELFLINKS=
Specifies that the input graph data are in a standardized format	STANDARDIZEDLABELS
Requests that the output graph data include standardized format	STANDARDIZEDLABELSOUT
Specifies whether time units are in CPU time or real time	TIMETYPE=
Data Statements	
LOADGRAPH Statement	
UNLOADGRAPH Statement	
Variables Statements	
LINKSQUERYVAR Statement	
Specifies the data variable name for the query path expansion lower bounds	EXPANDLOWER=
Specifies the data variable name for the query path expansion upper bounds	EXPANDUPPER=
Specifies the data variable name for the <i>from</i> nodes in the query graph	FROM=
Specifies the data variable name for the <i>to</i> nodes in the query graph	TO=
Specifies the data variables for the link attributes to read	VARS=
Specifies the data variables for the link attributes to match in the query graph	VARSMATCH=
LINKSVAR Statement	
Specifies the data variable name for the auxiliary link weights	AUXWEIGHT=
Specifies the data variable name for the <i>from</i> nodes	FROM=
Specifies the data variable name for the <i>to</i> nodes	TO=
Specifies the data variables for the additional link attributes to read	VARS=
Specifies the data variables for the additional link attributes to write to the output results	VARSOUT=
Specifies the data variable name for the link weights	WEIGHT=

Table 2.2 *continued*

Description	Option
NODESQUERYVAR Statement	
Specifies the data variable name for the nodes in the query graph	NODE=
Specifies the data variables for the node attributes to read	VAR=
Specifies the data variables for the node attributes to match in the query graph	VARSMATCH=
NODESSUBSETVAR Statement	
Specifies the data variable name for the nodes	NODE=
Specifies the data variable name for the reach identifier	REACH=
Specifies the data variable name for the sink indicator	SINK=
Specifies the data variable name for the source indicator	SOURCE=
NODESVAR Statement	
Specifies the data variable name for the nodes	NODE=
Specifies the data variables for the additional node attributes to read	VAR=
Specifies the data variables for the additional node attributes to write to the output results	VAROUT=
Specifies the data variable name for the node weights	WEIGHT=
Algorithm Statements	
BICONNECTEDCOMPONENTS Statement	
Specifies the output data table for biconnected components	OUT=
Specifies the output data table for the links of the block-cut tree	OUTBCTREELINKS=
Specifies the output data table for the nodes of the block-cut tree	OUTBCTREENODES=
CENTRALITY Statement	
Specifies how to calculate authority centrality	AUTH=
Specifies how to calculate betweenness centrality	BETWEEN=
Specifies whether to normalize the betweenness calculation	BETWEENNORM=
Specifies how to calculate closeness centrality	CLOSE=
Specifies a method for accounting for the shortest path distance between two nodes when a path does not exist (disconnected nodes)	CLOSENOPATH=
Calculates the node clustering coefficient	CLUSTERINGCOEFFICIENT
Specifies how to calculate degree centrality	DEGREE=
Specifies how to calculate eigenvector centrality	EIGEN=
Specifies the algorithm to use for eigenvector calculation	EIGENALGORITHM=
Specifies the maximum number of iterations for eigenvector calculation	EIGENMAXITERS=
Specifies how to calculate hub centrality	HUB=
Specifies how to calculate influence centrality	INFLUENCE=
Specifies how to calculate PageRank centrality metric	PAGERANK=
Specifies the damping factor for the PageRank algorithm	PAGERANKALPHA=
Specifies the convergence tolerance for the PageRank algorithm	PAGERANKTOLERANCE=
Specifies the percentage of source nodes to sample for the approximate betweenness calculation	SAMPLEPERCENT=

Table 2.2 continued

Description	Option
CLIQUE Statement	
Specifies whether to calculate the clique number	CLIQUENUMBER=
Specifies the maximum number of cliques to return during clique enumeration	MAXCLIQUES=
Specifies the maximum link weight for the cliques found	MAXLINKWEIGHT=
Specifies the maximum node weight for the cliques found	MAXNODEWEIGHT=
Specifies the maximum size for the cliques found	MAXSIZE=
Specifies the maximum amount of time to spend finding cliques	MAXTIME=
Specifies the minimum link weight for the cliques found	MINLINKWEIGHT=
Specifies the minimum node weight for the cliques found	MINNODEWEIGHT=
Specifies the minimum size for the cliques found	MINSIZE=
Specifies the output data table for cliques	OUT=
COMMUNITY Statement	
Specifies the community detection algorithm	ALGORITHM=
Specifies which data variable defines groups of nodes to fix together in a community	FIX=
Specifies the percentage of small-weight links to be removed	LINKREMOVALRATIO=
Specifies the maximum number of iterations for community detection	MAXITERS=
Specifies the output data table for intercommunity links	OUTCOMMLINKS=
Specifies the output data table for the community summary	OUTCOMMUNITY=
Specifies the output data table for the community level summary	OUTLEVEL=
Specifies the output data table for the community overlap	OUTOVERLAP=
Specifies the random factor in the parallel label propagation algorithm	RANDOMFACTOR=
Specifies the random seed for the parallel label propagation algorithm	RANDOMSEED=
Applies the recursive option to break large communities	RECURSIVE
Specifies the resolution list for community detection	RESOLUTIONLIST=
Specifies the modularity tolerance value for community detection	TOLERANCE=
Specifies which data variable defines the initial node partition for warm starting community detection	WARMSTART=
CONNECTEDCOMPONENTS Statement	
Specifies the algorithm to use for connected components	ALGORITHM=
Specifies the output data table for connected components	OUT=
CORE Statement	
Specifies the maximum amount of time to spend calculating the core decomposition	MAXTIME=
CYCLE Statement	
Specifies the algorithm to use for cycle enumeration	ALGORITHM=
Specifies the maximum number of cycles to return during cycle enumeration	MAXCYCLES=

Table 2.2 *continued*

Description	Option
Specifies the maximum length for the cycles found	MAXLENGTH=
Specifies the maximum link weight for the cycles found	MAXLINKWEIGHT=
Specifies the maximum node weight for the cycles found	MAXNODEWEIGHT=
Specifies the maximum amount of time to spend finding cycles	MAXTIME=
Specifies the minimum length for the cycles found	MINLENGTH=
Specifies the minimum link weight for the cycles found	MINLINKWEIGHT=
Specifies the minimum node weight for the cycles found	MINNODEWEIGHT=
Specifies the output data table for the links of the cycles	OUTCYCLESLINKS=
Specifies the output data table for the nodes of the cycles	OUTCYCLESNODES=
NODESIMILARITY Statement	
Specifies whether to calculate Adamic-Adar similarity	ADAMICADAR=
Specifies the number of lowest-ranked node pairs to output	BOTTOMK=
Specifies whether to calculate common neighbors similarity	COMMONNEIGHBORS=
Specifies the convergence threshold for vector similarity	CONVERGENCETHRESHOLD=
Specifies whether to calculate cosine similarity	COSINE=
Specifies whether to calculate Dice similarity	DICE=
Specifies whether to calculate vector embeddings	EMBED=
Specifies the variable names of precomputed vector embeddings	EMBEDDINGS=
Specifies whether to calculate Jaccard similarity	JACCARD=
Specifies the maximum similarity score to output	MAXSCORE=
Specifies the minimum similarity score to output	MINSORE=
Specifies the number of dimensions for vector embeddings	NDIMENSIONS=
Specifies the number of negative samples per positive sample used in training the model	NEGATIVESAMPLEFACTOR=
Specifies the number of training samples for the vector algorithm	NSAMPLES=
Specifies the similarity measure to use for node pair ranking	ORDERBY=
Specifies the output data table to contain the convergence curves for vector similarity	OUTCONVERGENCE=
Specifies the output data table to contain the similarity scores between pairs of nodes	OUTSIMILARITY=
Specifies the proximity order for the vector similarity algorithm	PROXIMITYORDER=
Specifies the sink node for similarity calculations	SINK=
Specifies the source node for similarity calculations	SOURCE=
Specifies whether to output sparse similarity data	SPARSE=
Specifies the number of highest-ranked node pairs to output	TOPK=
Specifies whether to calculate vector similarity	VECTOR=
PATH Statement	
Specifies the maximum length for the paths found	MAXLENGTH=
Specifies the maximum link weight for the paths found	MAXLINKWEIGHT=
Specifies the maximum node weight for the paths found	MAXNODEWEIGHT=
Specifies the maximum amount of time to spend finding paths	MAXTIME=
Specifies the minimum length for the paths found	MINLENGTH=

Table 2.2 *continued*

Description	Option
Specifies the minimum link weight for the paths found	MINLINKWEIGHT=
Specifies the minimum node weight for the paths found	MINNODEWEIGHT=
Specifies the output data table for path links	OUTPATHSLINKS=
Specifies the output data table for path nodes	OUTPATHSNODES=
Specifies the sink node for path calculations	SINK=
Specifies the source node for path calculations	SOURCE=
PATTERNMATCH Statement	
Specifies the code that defines FCMP functions	CODE=
Specifies whether to filter matches by using the induced subgraph of the match	INDUCED=
Specifies the FCMP function for link filters	LINKFILTER=
Specifies the FCMP function for link-pair filters	LINKPAIRFILTER=
Specifies the FCMP function for a filter that is based on a potential match (that is, any subset of nodes or links or both)	MATCHFILTER=
Specifies the maximum number of matches to return per query	MAXMATCHESPERQUERY=
Specifies the maximum amount of time to spend in pattern matching	MAXTIME=
Specifies the FCMP function for node filters	NODEFILTER=
Specifies the FCMP function for node-pair filters	NODEPAIRFILTER=
Specifies the links output data table for the induced subgraph of matches	OUTMATCHGRAPHLINKS=
Specifies the nodes output data table for the induced subgraph of matches	OUTMATCHGRAPHNODES=
Specifies the links output data table for matching subgraphs	OUTMATCHLINKS=
Specifies the nodes output data table for matching subgraph mappings	OUTMATCHNODES=
Specifies the links query output data table to contain the generated queries	OUTQUERYLINKS=
Specifies the nodes query output data table to contain the generated queries	OUTQUERYNODES=
Specifies the summary output data table for pattern match queries	OUTSUMMARY=
Specifies the data variable name for the query key	QUERYKEY=
Specifies whether to aggregate nodes across query keys for creating the induced subgraph(s)	QUERYKEYAGGREGATE=
PROJECTION Statement	
Specifies whether to calculate Adamic-Adar similarity	ADAMICADAR=
Specifies whether to calculate common neighbors similarity	COMMONNEIGHBORS=
Specifies whether to calculate cosine similarity	COSINE=
Specifies whether to calculate Dice similarity	DICE=
Specifies the projection method for directed graphs	DIRECTEDMETHOD=
Specifies whether to calculate Jaccard similarity	JACCARD=
Specifies the output data table to contain the lists of common neighbors between pairs of nodes	OUTNEIGHBORSLIST=
Specifies the output data table to contain the links of the projected graph	OUTPROJECTIONLINKS=
Specifies the output data table to contain the nodes of the projected graph	OUTPROJECTIONNODES=
Specifies the data variable name for the partition flag	PARTITION=

Table 2.2 *continued*

Description	Option
REACH Statement	
Calculates the directed reach counts	DIGRAPH
Treats each node as a source in reach calculations	EACHSOURCE
Specifies the maximum number of hops in the reach calculations	MAXREACH=
Specifies the output data table for reach counts	OUTCOUNTS=
Specifies the output data table for reach links	OUTREACHLINKS=
Specifies the output data table for reach nodes	OUTREACHNODES=
SHORTESTPATH Statement	
Specifies the maximum absolute objective gap for the paths found	MAXABSOBJGAP=
Specifies the maximum link weight for the paths found	MAXLINKWEIGHT=
Specifies the maximum number of ranked paths to find for each pair	MAXPATHSPERPAIR=
Specifies the maximum relative objective gap for the paths found	MAXRELOBJGAP=
Specifies the minimum link weight for the paths found	MINLINKWEIGHT=
Specifies the output data table for shortest paths links	OUTPATHSLINKS=
Specifies the output data table for shortest paths nodes	OUTPATHSNODES=
Specifies the output data table for shortest path descriptive statistics	OUTSUMMARY=
Specifies the output data table for shortest path weights	OUTWEIGHTS=
Specifies which data variable in the nodes subset data table defines the sequence of nodes to visit	SEQUENCE=
Specifies the sink node for shortest path calculations	SINK=
Specifies the source node for shortest path calculations	SOURCE=
SUMMARY Statement	
Calculates the degree assortativity and specifies which type of calculation to perform	ASSORTATIVITYDEGREE=
Specifies the name of the data variable for computing the nominal assortativity	ASSORTATIVITYNOMINAL=
Specifies the name of the data variable for computing the numeric assortativity	ASSORTATIVITYNUMERIC=
Calculates information about biconnected components	BICONNECTEDCOMPONENTS
Calculates information about clustering coefficients	CLUSTERINGCOEFFICIENT
Calculates information about connected components	CONNECTEDCOMPONENTS
Calculates the approximate diameter and chooses the weight type	DIAMETERAPPROX=
Includes only finite values when calculating descriptive statistics that are related to shortest paths	FINITEPATH
Specifies the output data table for summary results	OUT=
Calculates information about shortest paths and chooses the weight type	SHORTESTPATH=
TRANSITIVECLOSURE Statement	
Specifies the output data table for transitive closure results	OUT=

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

Table 2.3 Supported Input Formats by Statement

Statement and Options	UNDIRECTED	DIRECTED
BICONNECTEDCOMPONENTS	X	
CENTRALITY		
AUTH=		X
HUB=		X
Otherwise	X	X
CLIQUE	X	
COMMUNITY	X	X
CONNECTEDCOMPONENTS		
ALGORITHM=DFS	X	X
Otherwise	X	
CORE	X	X
CYCLE	X	X
NODESIMILARITY	X	X
PATH	X	X
PATTERNMATCH	X	X
PROJECTION	X	X
REACH	X	X
SHORTESTPATH	X	X
SUMMARY		
BICONNECTEDCOMPONENTS	X	
CLUSTERINGCOEFFICIENT	X	
DIAMETERAPPROX=	X	
Otherwise	X	X
TOPOLOGICALSORT		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 and Options	OUTNODES=	OUTLINKS=	Algorithm Statement Options
BICONNECTEDCOMPONENTS	X	X	OUT=, OUTBCTREELINKS=, OUTBCTREENODES=
CENTRALITY			
BETWEEN=	X	X	
Otherwise	X		
CLIQUE			OUT=
CLIQUENUMBER=TRUE	X		
COMMUNITY	X	X	OUTCOMMLINKS=, OUTCOMMUNITY=, OUTLEVEL=,

Table 2.4 *continued*

Statement and Options	OUTNODES=	OUTLINKS=	Algorithm Statement Options
			OUTOVERLAP=
CONNECTEDCOMPONENTS	X	X	OUT=
CORE	X		
CYCLE			OUTCYCLESLINKS=, OUTCYCLESNODES=
NODESIMILARITY VECTOR=TRUE	X		OUTCONVERGENCE=
PATH			OUTSIMILARITY=
PATTERNMATCH			OUTPATHSLINKS=, OUTPATHSNODES=
			OUTMATCHGRAPHLINKS=, OUTMATCHGRAPHNODES=, OUTMATCHLINKS=, OUTMATCHNODES=, OUTQUERYLINKS=, OUTQUERYNODES=, OUTSUMMARY=
PROJECTION			OUTNEIGHBORSLIST=, OUTPROJECTIONLINKS=, OUTPROJECTIONNODES=
REACH	X		OUTCOUNTS=, OUTREACHLINKS=, OUTREACHNODES=
SHORTESTPATH			OUTPATHSLINKS=, OUTPATHSNODES=, OUTWEIGHTS=, OUTSUMMARY=
SUMMARY	X	X	OUT=
TOPOLOGICALSORT	X		
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:

DETERMINISTIC=TRUE | FALSE

specifies whether to enforce determinism. By default, DETERMINISTIC=TRUE, which ensures that each invocation (with the same machine configuration and parameter settings) produces the same final result. For more information about determinism, see the section “[Determinism](#)” on page 78.

DIRECTION=DIRECTED | UNDIRECTED

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

- DIRECTED** considers the input graph to be directed. In a directed graph, each link (i, j) has a direction that defines how something (such as information) can flow over that link. In link (i, j) , the flow is from node i to node j ($i \rightarrow j$). Node i is called the *source* (*tail*) node, and node j is called the *sink* (*head*) node.
- UNDIRECTED** considers the input graph to be 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 and Output Data](#)” on page 61.

DISTRIBUTED=TRUE | FALSE

specifies whether to use a distributed graph. By default, DISTRIBUTED=FALSE, which means that a distributed graph is not used. For more information about the algorithms that support distributed graph computation, see the section “[Execution Modes and Data Movement](#)” on page 78.

GRAPH=number

specifies the in-memory graph to use. This option can be used with any algorithm that supports in-memory execution.

For more information about using the GRAPH= option, see the section “[Persistent Data Structures \(In-Memory Graphs\)](#)” on page 81.

INDEXOFFSET=number

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

LINKS=libref.data-table

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

For more information about this input table, see the section “[Links Input Data](#)” on page 62.

LINKSQUERY=libref.data-table

specifies the input data table that contains the graph link information for the query graph. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the section “[Using CAS Sessions and CAS Engine Librefs](#)” on page 2 in Chapter 1, “[Introduction](#).”

This option is supported only when used in conjunction with the **PATTERNMATCH** statement. For more information about this input table, see the section “[Pattern Matching](#)” on page 168.

LOGFREQTIME=*number*

LOGFREQUENCYTIME=*number*

controls the frequency (in *number* of seconds) for displaying iteration logs for some algorithms, where *number* can be any integer greater than or equal to 1. This option is useful for computationally intensive algorithms. Setting *number* too low can hurt algorithm performance. This option is ignored for a single-server setup. 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 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.

MULTILINKS=TRUE | FALSE

specifies whether to include or aggregate multilinks when an input graph (specified by the LINKS= or LINKSQUERY= option) is read. You can specify the following values:

FALSE	aggregates multilinks.
TRUE	includes multilinks.

By default, MULTILINKS=TRUE for algorithms that support multilinks.

For more information about this option, see the section “[Multigraphs](#)” on page 68.

NODES=*libref.data-table*

specifies the input data table that contains the graph node information. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the section “[Using CAS Sessions and CAS Engine Librefs](#)” on page 2 in Chapter 1, “[Introduction](#).”

For more information about this input table, see the section “[Nodes Input Data](#)” on page 66.

NODESQUERY=*libref.data-table*

specifies the input data table that contains the graph node information for the query graph. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the section “[Using CAS Sessions and CAS Engine Librefs](#)” on page 2 in Chapter 1, “[Introduction](#).”

This option is supported only when used in conjunction with the **PATTERNMATCH** statement. For more information about this input table, see the section “[Pattern Matching](#)” on page 168.

NODESSUBSET=libref.data-table

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

For more information about this input table, see the section “Nodes Subset Input Data” on page 67.

NTHREADS=number

specifies the maximum number of threads to use for multithreaded processing. Some of the algorithms can take advantage of multicore machines and can run faster when *number* is greater than 1 (see Table 2.6 for a list). Algorithms that cannot take advantage of this option use only one thread even if *number* is greater than 1. For distributed execution, *number* specifies the maximum number of threads to use on each machine. The value of *number* can be any integer between 1 and 1024, 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.

OUTGRAPHLIST=libref.data-table

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

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

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

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

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

SELFLINKS=TRUE | FALSE

specifies whether to include or remove self-links when an input graph (specified by the LINKS= or LINKSQQUERY= option) is read. You can specify the following values:

FALSE	removes self-links.
TRUE	includes self-links.

By default, SELFLINKS=TRUE for algorithms that support self-links.

For more information about this option, see the section “Self-Links” on page 71.

STANDARDIZEDLABELS

specifies that the input graph data are in a standardized format, as described in the section “[Standardized Labels Input](#)” on page 74.

STANDARDIZEDLABELSOUT

specifies that the output graph data include standardized format, as described in the section “[Standardized Labels Output](#)” on page 76.

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 < option > ;

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 94.

You can specify the following *option*:

OUT=libref.data-table

specifies the output data table to contain the biconnected components summary results. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “[Using CAS Sessions and CAS Engine Librefs](#)” on page 2 in Chapter 1, “[Introduction](#).”

OUTBCTREELINKS=libref.data-table

specifies the output data table to contain the links of the block-cut tree. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “[Using CAS Sessions and CAS Engine Librefs](#)” on page 2 in Chapter 1, “[Introduction](#).”

OUTBCTREENODES=libref.data-table

specifies the output data table to contain the nodes of the block-cut tree. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “[Using CAS Sessions and CAS Engine Librefs](#)” on page 2 in Chapter 1, “[Introduction](#).”

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 the “Grouping Data” section of *SAS Programmers Guide: Essentials*.

The specified BY variables must come from the LINKS= data table. The BY statement in PROC NETWORK is not supported when multiple input data tables are used. Two examples of using BY-group processing are shown in “[Example 2.8: Centrality Metrics for an Undirected Graph by Community](#)” on page 290 and “[Example 2.14: Shortest Path in a Road Network by Date and Time](#)” on page 309.

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 100.

You can specify the following *options*:

AUTH=WEIGHT | UNWEIGHT | BOTH

specifies how to calculate authority centrality. You can specify the following values:

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

If the input graph does not contain link weights, then AUTH=WEIGHT and AUTH=UNWEIGHT produce the same results. 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 117.

BETWEEN=WEIGHT | UNWEIGHT | BOTH

specifies how to calculate betweenness centrality for node betweenness or link betweenness. You can specify the following values:

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

If the input graph does not contain link weights, then `BETWEEN=WEIGHT` and `BETWEEN=UNWEIGHT` produce the same results. 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 111.

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 111. By default, `BETWEENNORM=TRUE`.

CLOSE=WEIGHT | UNWEIGHT | BOTH

specifies how to calculate closeness centrality. You can specify the following values:

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

If the input graph does not contain link weights, then `CLOSE=WEIGHT` and `CLOSE=UNWEIGHT` produce the same results. For more information about the closeness centrality metric, see the section “[Closeness Centrality](#)” on page 108.

CLOSENOPATH=DIAMETER | HARMONIC | NNODES | ZERO

specifies a method for accounting for the shortest path distance between two nodes when a path does not exist (disconnected nodes). You can specify the following values:

DIAMETER	uses the graph diameter (plus one) as the shortest path distance between disconnected nodes.
HARMONIC	uses the harmonic formula for calculating closeness centrality.
NNODES	uses the number of nodes as the shortest path distance between disconnected nodes. You cannot specify this option when <code>CLOSE=WEIGHT</code> or <code>CLOSE=BOTH</code> .
ZERO	uses zero as the shortest path distance between disconnected nodes.

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 108. By default, `CLOSENOPATH=DIAMETER`.

CLUSTERINGCOEFFICIENT**CLUSTERINGCOEF**

calculates the node clustering coefficient. For more information about clustering coefficients, see the section “[Clustering Coefficient](#)” on page 105.

DEGREE=WEIGHT | UNWEIGHT | BOTH

specifies how to calculate degree centrality. You can specify the following values:

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

If the input graph does not contain link weights, then **DEGREE=WEIGHT** and **DEGREE=UNWEIGHT** produce the same results. For more information about the degree centrality metric, see the section “[Degree Centrality](#)” on page 100.

EIGEN=WEIGHT | UNWEIGHT | BOTH

specifies how to calculate eigenvector centrality. You can specify the following values:

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

If the input graph does not contain link weights, then **EIGEN=WEIGHT** and **EIGEN=UNWEIGHT** produce the same results. For more information about the eigenvector centrality metric, see the section “[Eigenvector Centrality](#)” on page 115.

EIGENALGORITHM=AUTOMATIC | JACOBIDAVIDSON | POWER

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

AUTOMATIC	automatically determines the eigensolver to use.
JACOBIDAVIDSON JD	uses a variant of the Jacobi-Davidson algorithm for solving eigensystems (Sleijpen and van der Vorst 2000). This algorithm is not supported for PageRank calculations.
POWER	uses the power method to calculate eigenvectors.

By default, **EIGENALGORITHM=AUTOMATIC**.

EIGENMAXITERS=number

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

HUB=WEIGHT | UNWEIGHT | BOTH

specifies how to calculate hub centrality. You can specify the following values:

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

If the input graph does not contain link weights, then HUB=WEIGHT and HUB=UNWEIGHT produce the same results. 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 117.

INFLUENCE=WEIGHT | UNWEIGHT | BOTH

specifies how to calculate influence centrality. You can specify the following values:

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

If the input graph does not contain link or node weights, then INFLUENCE=WEIGHT and INFLUENCE=UNWEIGHT produce the same results. For more information about the influence centrality metric, see the section “[Influence Centrality](#)” on page 103.

PAGERANK=WEIGHT | UNWEIGHT | BOTH

specifies how to calculate PageRank centrality. You can specify the following values:

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

If the input graph does not contain link or node weights, then PAGERANK=WEIGHT and PAGERANK=UNWEIGHT produce the same results. For more information about the PageRank centrality metric, see the section “[PageRank Centrality](#)” on page 120.

PAGERANKALPHA=*number*

specifies the damping factor to use in the PageRank algorithm. The value of *number* must be between 0 and 1 (inclusive). The default is 0.85, which means that a random web surfer has a 15% chance of jumping to any other node in the network at any time. The algorithm usually takes more iterations to converge (or does not converge at all) as the damping factor is set closer to 1. See the section “[PageRank Centrality](#)” on page 120 for an example that uses this option.

PAGERANKTOLERANCE=*number*

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

SAMPLEPERCENT=number

specifies the percentage of source nodes to sample for the approximate betweenness calculation. The value of *number* must be a positive number less than or equal to 100. The default is 100.

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 123.

You can specify the following *options*:

CLIQUENUMBER=TRUE | FALSE

specifies whether to calculate the clique number of the graph. The *clique number* of a graph is the maximum size (number of nodes) of any clique that the algorithm finds in the graph. For each node *i*, the *node clique number* is the maximum size of any clique that the algorithm finds in the graph that contains node *i*. The calculated clique number is subject to any filter options (for example, MAXLINKWEIGHT=). If CLIQUENUMBER=TRUE, the MAXCLIQUES= option is set to ALL. If CLIQUENUMBER=TRUE and you specify the OUTNODES= option in the PROC NETWORK statement, the *node clique number* is calculated for each node. By default, CLIQUENUMBER=FALSE.

MAXCLIQUES=number | ALL

specifies the maximum number of cliques for clique enumeration to return. You can specify either a *number* (any 64-bit integer greater than or equal to 1) or ALL (the maximum value that can be represented by a 64-bit integer). By default, MAXCLIQUES=1 when CLIQUENUMBER=FALSE; otherwise, MAXCLIQUES=ALL.

MAXLINKWEIGHT=number

specifies the maximum sum of link weights in a clique. Any clique whose sum of link weights is greater than *number* is removed from the results. In the case of a multigraph, all of the links in the induced subgraph are included in the sum. The default is the largest number that can be represented by a double. When the default is used, no cliques are removed from the results.

MAXNODEWEIGHT=number

specifies the maximum sum of node weights in a clique. Any clique whose sum of node weights is greater than *number* is removed from the results. The default is the largest number that can be represented by a double. When the default is used, no cliques are removed from the results.

MAXSIZE=number

specifies the maximum number of nodes in a clique. Any clique whose size is greater than *number* is removed from the results. The default is the largest number that can be represented by a 32-bit integer. When the default is used, no cliques are removed from the results.

MAXTIME=number

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

MINLINKWEIGHT=number

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

MINNODEWEIGHT=number

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

MINSIZE=number

specifies the minimum number of nodes in a clique. Any clique that has fewer nodes than *number* is removed from the results. By default, MINSIZE=1 and no cliques are removed from the results.

OUT=libref.data-table

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

COMMUNITY Statement

COMMUNITY < options > ;

The COMMUNITY statement invokes an algorithm that detects communities in the input graph. For more information about community detection, see the section “Community Detection” on page 128.

You can specify the following *options*:

ALGORITHM=LABELPROPAGATION | LOUVAIN

specifies the algorithm to use for community detection. You can specify the following values:

LABELPROPAGATION uses the label propagation algorithm proposed by Raghavan, Albert, and Kumara (2007).

LOUVAIN uses the Louvain algorithm proposed by Blondel et al. (2008).

By default, ALGORITHM=LOUVAIN.

FIX=column

specifies which data variable defines groups of nodes to fix together in a community. Each value of the variable must be a 32-bit integer greater than or equal to 0 or missing. You cannot use this option when **ALGORITHM=LABELPROPAGATION** unless **LABELUPDATEMODE=ASYNCHRONOUS**.

LABELUPDATEMODE=ASYNCHRONOUS | SYNCHRONOUS

specifies whether nodes update their labels according to the labels of their neighbors at the current iteration (asynchronous) or the previous iteration (synchronous). This option is ignored unless you specify `ALGORITHM=LABELPROPAGATION`. By default, `LABELUPDATEMODE=SYNCHRONOUS`.

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 a weight of 100, and link $A \rightarrow C$ has a 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. If you specify this option, then the algorithm ignores any link whose weight is less than $(\text{number}/100) * \text{max_link_weight}$, where *max_link_weight* is the maximum link weight among all links incident to the same node. 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=LABELPROPAGATION`.

OUTCOMMLINKS=libref.data-table

specifies the output data table to describe the links between communities. If you also specify the `RECURSIVE` option, PROC NETWORK ignores the `OUTCOMMLINKS=` option. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

OUTCOMMUNITY=libref.data-table

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

OUTLEVEL=libref.data-table

specifies the output data table to contain community information at different resolution levels. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

OUTOVERLAP=libref.data-table

specifies the output data table to describe the intensity of each node. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

RANDOMFACTOR=number

specifies the random factor for the synchronous 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 synchronous 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 largest number that can be represented by a 32-bit integer.
- MAXDIAMETER=** specifies the maximum number of links on the shortest paths between any pair of nodes in any community. This option is ignored when you specify **ALGORITHM=LABELPROPAGATION** and **LABELUPDATEMODE=SYNCHRONOUS**. The default is the largest number that can be represented by a 32-bit integer.
- RELATION=** the relationship between the values of MAXCOMMSIZE= and MAXDIAMETER= options. If RELATION=AND, then recursive splitting continues until both of the MAXCOMMSIZE and MAXDIAMETER conditions are satisfied. If RELATION=OR, then recursive splitting continues until either the MAXCOMMSIZE or the MAXDIAMETER condition is satisfied.

RESOLUTIONLIST=num_list

specifies a list of resolution values (nonnegative numbers) that are separated by spaces (for example, 4.3 2.1 1.0 0.6 0.2). 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, a resolution of 4.3 produces more communities than a resolution of 0.2. The NETWORK procedure interprets the RESOLUTIONLIST= option differently, depending on the value of the **ALGORITHM=** option:

- When **ALGORITHM=LABELPROPAGATION**, the resolution value represents a density threshold. If **LABELUPDATEMODE=ASYNCHRONOUS**, then PROC NETWORK ignores the RESOLUTIONLIST= option and uses the default value of 0. If **LABELUPDATEMODE=SYNCHRONOUS**, then RESOLUTIONLIST=0.001 by default. In this case, specifying multiple resolution values causes PROC NETWORK to perform community detection multiple times, each time with a different resolution value, and the RESOLUTIONLIST= option is fully compatible with the RECURSIVE option.

- When **ALGORITHM=LOUVAIN**, the resolution value represents a factor in the modularity equation that affects the expected number of links between two nodes. If you specify more than one resolution value and the **RECURSIVE** option, PROC NETWORK ignores the **RESOLUTIONLIST=** option and uses the default value of 1.0.

For more information about the use of the **RESOLUTIONLIST=** option, see the section “[Large Communities](#)” on page 130.

TOLERANCE=number

MODULARITY=number

specifies the tolerance value for when to stop iterations. When **ALGORITHM=LOUVAIN**, the algorithm stops iterations when the fraction of modularity gain between two consecutive iterations is less than *number*. When **ALGORITHM=LABELPROPAGATION**, the algorithm stops iterations when the fraction 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=0.05** if **ALGORITHM=LABELPROPAGATION** and **LABELUPDATEMODE=SYNCHRONOUS**; otherwise **TOLERANCE=0.001**.

WARMSTART=column

specifies which data variable defines the initial node partition for warm starting community detection. Each value of the variable must be a 32-bit integer greater than or equal to 0 or missing.

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 137. You can specify the following *options*:

ALGORITHM=AUTOMATIC | AFFOREST | DFS | UNIONFIND

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

AUTOMATIC	uses the union-find or afforest algorithm for undirected graphs and the depth-first search algorithm for directed graphs.
AFFOREST	uses the afforest algorithm for connected components. You can use this algorithm only with undirected graphs.
DFS	uses the depth-first search algorithm for connected components.
UNIONFIND	uses the union-find algorithm for connected components. You can use this algorithm only with undirected graphs. You can also execute this algorithm in a distributed manner across multiple machines by specifying DISTRIBUTED=TRUE in the PROC NETWORK statement.

By default, **ALGORITHM=AUTOMATIC**.

OUT=libref.data-table

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

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 142.

You can specify the following *option*:

MAXTIME=number

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

CYCLE Statement

CYCLE < options > ;

The CYCLE statement invokes an algorithm that finds the cycles (or the existence of a cycle) in the input graph. For more information about cycles, see the section “Cycle Enumeration” on page 146.

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 a **MAXLENGTH=** option value greater than 20; otherwise, **ALGORITHM=BUILD**.

MAXCYCLES=number | ALL

specifies the maximum number of cycles for cycle enumeration to return. You can specify either a *number* (any 64-bit integer greater than or equal to 1) or **ALL** (the maximum value that can be represented by a 64-bit integer). By default, **MAXCYCLES=1**.

MAXLENGTH=number

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

MAXLINKWEIGHT=number

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

MAXNODEWEIGHT=number

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

MAXTIME=number

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

MINLENGTH=number

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

MINLINKWEIGHT=number

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

MINNODEWEIGHT=number

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

OUTCYCLESLINKS=libref.data-table

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

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

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

DISPLAY Statement

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

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

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

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

The table names that you can specify are listed in the section “[ODS Table Names](#)” on page 261. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

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

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

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

CASESENSITIVE

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

EXCLUDE

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

EXCLUDEALL

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

TRACE

displays the display table names, labels, and paths.

DISPLAYOUT Statement

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

The DISPLAYOUT statement enables you to create 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 output tables to create. Each entry in the list has either a *key=value* format or a *key* format:

<i>key=value</i>	specifies <i>key</i> as the ODS table name, path, or partial pathname, and specifies <i>value</i> as the output table name.
<i>key</i>	specifies <i>key</i> as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section “[ODS Table Names](#)” on page 261. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the [DISPLAY](#) statement. The DISPLAYOUT statement does not support regular expressions.

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

INCLUDEALL

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

NOREPLACE

does not replace any existing output table of the same name.

REPEATED

replicates all output tables on all nodes.

The output tables that the NETWORK procedure produces when you use the DISPLAYOUT statement are a transposed version of the displayed tables. This allows for easier use in subsequent analyses, especially when it is used together with BY-group processing. Two examples of using the DISPLAYOUT statement are shown in “[Example 2.8: Centrality Metrics for an Undirected Graph by Community](#)” on page 290 and “[Example 2.14: Shortest Path in a Road Network by Date and Time](#)” on page 309.

LINKSQUERYVAR Statement

LINKSQUERYVAR < *options* > ;

The LINKSQUERYVAR statement enables you to explicitly specify the data variable names for PROC NETWORK to use when it reads the data table that you specify in the **LINKSQUERY=** option in the PROC NETWORK statement. This option is supported only when used in conjunction with the **PATTERNMATCH** statement. For more information about this input table, see the section “[Pattern Matching](#)” on page 168.

You can specify the following *options*:

EXPANDLOWER=*column*

specifies the name of the data variable for the query path expansion lower bounds. The value of the variable must be a 32-bit integer greater than 0. By default, the value is 1. A missing value indicates no expansion.

EXPANDUPPER=*column*

specifies the name of the data variable for the query path expansion upper bounds. The value of the variable must be a 32-bit integer greater than or equal to the value that you specify for the expansion lower bound. By default, the value is 1. A missing value indicates no expansion.

FROM=*column*

FROMVAR=*column*

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

TO=*column*

TOVAR=*column*

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

VAR=(*columns*)

specifies the names of the data variables to read. The value of the variables can be numeric or character.

VARSMATCH=(*columns*)

specifies the names of the data variables to match in the query graph. The value of the variables can be numeric or character. By default, the variables to match are the same as the variables read (as specified in the **VAR=** option). A common use case of the **VARSMATCH=** option is when you want to use query graph link attributes in a filter function without forcing an exact attribute match. To do this, specify these variables in the **VAR=** option but not in the **VARSMATCH=** option.

LINKSVAR Statement

LINKSVAR < options > ;

The LINKSVAR statement enables you to explicitly specify the data variable names for PROC NETWORK to use when it reads the data table that you specify in the **LINKS=** option in the PROC NETWORK statement. For more information about the format of the links input data table, see the section “[Links Input Data](#)” on page 62.

You can specify the following *options*:

AUXWEIGHT=*column*

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

FROM=*column*

FROMVAR=*column*

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

TO=*column*

TOVAR=*column*

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

VAR=(*columns*)

specifies the names of the additional data variables to read. The value of the variables can be numeric or character.

VAR=(*columns*)

specifies the names of the additional data variables to write to the output results. The value of the variables can be numeric or character. By default, the variables to write are the same as the variables read (as specified in the VAR= option).

WEIGHT=*column*

specifies the name of the data variable for the link weights. The value of the variable must be numeric. The default name is weight.

LOADGRAPH Statement

LOADGRAPH ;

The LOADGRAPH statement reads the input graph from tables that are specified by the LINKS= option or the NODES= option (or both) in the PROC NETWORK statement, and it retains the graph in memory (within the current CAS session). The LOADGRAPH statement is described further in the section “[Persistent Data Structures \(In-Memory Graphs\)](#)” on page 81.

NODESIMILARITY Statement

NODESIMILARITY < options > ;

The NODESIMILARITY statement invokes an algorithm that calculates how similar the neighborhoods of pairs of nodes are in an input graph. The node similarity problem is described in the section “[Node Similarity](#)” on page 152.

You can specify the following *options*:

ADAMICADAR=TRUE | FALSE

specifies whether to calculate Adamic-Adar node similarity. You can specify the following values:

TRUE	calculates Adamic-Adar node similarity and saves the results in the output table that is specified in the OUTSIMILARITY= option.
FALSE	does not calculate Adamic-Adar node similarity.

By default, ADAMICADAR=FALSE.

BOTTOMK=*number*

specifies the maximum *number* of lowest-ranked similarity pairs to output in the OUTSIMILARITY= table, where *number* can be any integer greater than or equal to 0. If more than one similarity measure is requested, you must use the ORDERBY= option to specify which similarity measure to use for ranking.

COMMONNEIGHBORS=TRUE | FALSE

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

TRUE	calculates common neighbors node similarity and saves the results in the output table that is specified in the OUTSIMILARITY= option.
FALSE	does not calculate common neighbors node similarity.

By default, COMMONNEIGHBORS=FALSE.

CONVERGENCE*THRESHOLD=number*

specifies the convergence threshold for vector similarity. If the convergence value drops below this threshold before the procedure completes the number of samples that are requested in the **NSAMPLES=** option, the vector embeddings training phase terminates early. By default, **CONVERGENCE***THRESHOLD=0.01*.

COSINE=TRUE | FALSE

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

TRUE	calculates cosine node similarity and saves the results in the output table that is specified in the OUTSIMILARITY= option.
FALSE	does not calculate cosine node similarity.

By default, **COSINE=FALSE**.

DICE=TRUE | FALSE

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

TRUE	calculates Dice node similarity and saves the results in the output table that is specified in the OUTSIMILARITY= option.
FALSE	does not calculate Dice node similarity.

By default, **DICE=FALSE**.

EMBED=TRUE | FALSE

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

TRUE	calculates vector embeddings and saves the results in the output table that is specified in the OUTNODES= option.
FALSE	does not calculate vector embeddings.

By default, **EMBED=FALSE** when **VECTOR=FALSE**, and **EMBED=TRUE** when **VECTOR=TRUE**.

EMBEDDINGS=(columns)

specifies the names of one or more data variables to be used as precalculated vector embeddings. The value of the variables can be numeric only. Each variable must exist in the input table that is specified in the **NODES=** option, and a nonmissing value must exist for each variable for each node that appears in the graph. This option is ignored unless **VECTOR=TRUE** or **EMBED=TRUE** is specified.

JACCARD=TRUE | FALSE

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

TRUE	calculates Jaccard node similarity and saves the results in the output table that is specified in the OUTSIMILARITY= option.
FALSE	does not calculate Jaccard node similarity.

By default, **JACCARD=TRUE**.

MAXSCORE=number

specifies the maximum similarity score to output in the OUTSIMILARITY= table, where *number* must be in the range [0, 1]. The MAXSCORE= option is supported only for the Jaccard and vector similarity measures; it does not apply to the Adamic-Adar, common neighbors, cosine, or Dice similarity measure. If you specify both JACCARD=TRUE and VECTOR=TRUE, node pairs are output if either the Jaccard or vector similarity score satisfies the maximum similarity condition. By default, MAXSCORE=1.

MINSORE=number

specifies the minimum similarity score to output in the OUTSIMILARITY= table, where *number* must be in the range [0, 1]. The MINSORE= option is supported only for the Jaccard and vector similarity measures; it does not apply to the Adamic-Adar, common neighbors, cosine, or Dice similarity measure. If you specify both JACCARD=TRUE and VECTOR=TRUE, node pairs are output if either the Jaccard or vector similarity score satisfies the minimum similarity condition. By default, MINSORE=0.

NDIMENSIONS=number

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

NEGATIVESAMPLEFACTOR=number

specifies a multiplier for the number of negative training samples per positive training sample for the vector algorithm. If NSAMPLES=*s*, the vector algorithm uses *s* positive samples and *number* times *s* negative samples. By default, NEGATIVESAMPLEFACTOR=5.

NSAMPLES=number

specifies the number of training samples for the vector algorithm. By default, NSAMPLES= $1000 \times |E|$, where $|E|$ is the number of links in the input graph. For distributed execution, this number of training samples is processed per machine.

ORDERBY=ADAMICADAR | COMMONNEIGHBORS | COSINE | DICE | JACCARD | VECTOR

specifies the similarity measure to use for ranking the similarity pairs to output in the OUTSIMILARITY= table. This option is ignored if neither the TOPK= option nor the BOTTOMK= option is specified.

OUTCONVERGENCE=libref.data-table

specifies the output data table to contain the convergence curves for vector similarity. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

OUTSIMILARITY=libref.data-table

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

PROXIMITYORDER=FIRST | SECOND

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

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

By default, PROXIMITYORDER=SECOND.

SINK=sink-node

specifies the sink node for node similarity calculations.

SOURCE=source-node

specifies the source node for node similarity calculations.

SPARSE=TRUE | FALSE

specifies whether to output sparse similarity data. You can specify the following values:

TRUE

outputs a sparse set of node pairs in the OUTSIMILARITY= table.

FALSE

outputs a dense set of node pairs in the OUTSIMILARITY= table.

You cannot specify SPARSE=TRUE when you specify VECTOR=TRUE. By default, SPARSE=FALSE.

TOPK=number

specifies the maximum *number* of highest-ranked similarity pairs to output in the OUTSIMILARITY= table, where *number* can be any integer greater than or equal to 0. If more than one similarity measure is requested, you must use the ORDERBY= option to specify which similarity measure to use for ranking.

VECTOR=TRUE | FALSE

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

TRUE

calculates vector node similarity and saves the results in the output table that is specified in the OUTSIMILARITY= option.

FALSE

does not calculate vector node similarity.

When you specify VECTOR=TRUE, you cannot specify SPARSE=TRUE. By default, VECTOR=FALSE.

NODESQUERYVAR Statement

NODESQUERYVAR < options > ;

The NODESQUERYVAR statement enables you to explicitly specify the data variable names for PROC NETWORK to use when it reads the data table that you specify in the **NODESQUERY=** option in the PROC NETWORK statement. This option is supported only when used in conjunction with the **PATTERNMATCH** statement. For more information about this input table, see the section “[Pattern Matching](#)” on page 168.

You can specify the following *options*:

NODE=column

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

VAR=(columns)

specifies the names of the data variables to read. The value of the variables can be numeric or character.

VARSMATCH=(columns)

specifies the names of the data variables to match in the query graph. The value of the variables can be numeric or character. By default, the variables to match are the same as the variables read (as specified in the VAR= option). A common use case of the VARSMATCH= option is when you want to use query graph node attributes in a filter function without forcing an exact attribute match. To do this, specify these variables in the VAR= option but not in the VARSMATCH= option.

NODESSUBSETVAR Statement

NODESSUBSETVAR < options > ;

The NODESSUBSETVAR statement enables you to explicitly specify the data variable names for PROC NETWORK to use when it reads the data table that you specify in the **NODESSUBSET=** option in the PROC NETWORK statement. For more information about the format of the node subset input data table, see the section “[Nodes Input Data](#)” on page 66.

You can specify the following *options*:

NODE=column

specifies the name of the data variable for the nodes. The value of the variable can be numeric or character. The default name is `node`.

REACH=column

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

SINK=column

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

SOURCE=column

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

NODESVAR Statement

NODESVAR < options > ;

The NODESVAR statement enables you to explicitly specify the data variable names for PROC NETWORK to use when it reads the data table that you specify in the **NODES=** option in the PROC NETWORK statement. For more information about the format of the node input data table, see the section “[Nodes Input Data](#)” on page 66.

You can specify the following *options*:

NODE=*column*

specifies the name of the data variable for the nodes. The value of the variable can be numeric or character. The default name is `node`.

VAR=*(column(s))*

specifies the names of the additional data variables to read. The value of the variables can be numeric or character.

VAROUT=*(column(s))*

specifies the names of the additional data variables to write to the output results. The value of the variables can be numeric or character. By default, the variables to write are the same as the variables read (as specified in the `VAR=` option).

WEIGHT=*column*

specifies the name of the data variable for the node weights. The value of the variable must be numeric. The default name is `weight`.

PATH Statement

PATH < *options* > ;

The **PATH** statement invokes an algorithm that finds the paths in the input graph. For more information about paths, see the section “[Path Enumeration](#)” on page 164.

You can specify the following *options*:

MAXLENGTH=*number*

specifies the maximum number of links in a path. Any path whose length is greater than *number* is removed from the results. The default is the largest number that can be represented by a 32-bit integer. When the default is used, no paths are removed from the results.

MAXLINKWEIGHT=*number*

specifies the maximum sum of link weights in a path. Any path whose sum of link weights is greater than *number* is removed from the results. The default is the largest number that can be represented by a double. When the default is used, no paths are removed from the results.

MAXNODEWEIGHT=*number*

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

MAXTIME=*number*

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

MINLENGTH=number

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

MINLINKWEIGHT=number

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

MINNODEWEIGHT=number

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

OUTPATHSLINKS=libref.data-table**OUTPATHS=libref.data-table**

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

OUTPATHSNODES=libref.data-table

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

SINK=sink-node

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

SOURCE=source-node

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

PATTERNMATCH Statement

PATTERNMATCH < options > ;

The PATTERNMATCH statement invokes an algorithm that searches for subgraph patterns in an input graph. The pattern matching problem is described in the section “Pattern Matching” on page 168.

You can specify the following *options*:

CODE=string

specifies the code that defines FCMP functions.

INDUCED=TRUE | FALSE

specifies whether to filter matches by using the [induced subgraph](#) of the match. You can specify the following values:

TRUE	keeps only matches whose induced subgraph is topologically equivalent to the query graph.
FALSE	does not filter by using the induced subgraph of the match.

By default, INDUCED=FALSE.

LINKFILTER=function-name

specifies the FCMP function for link filters.

LINKPAIRFILTER=function-name

specifies the FCMP function for link-pair filters.

MATCHFILTER=function-name

specifies the FCMP function for a filter that is based on a potential match (that is, any subset of nodes or links or both).

MAXMATCHESPERQUERY=number | ALL**MAXMATCHES=number | ALL**

specifies the maximum number of matches to return for each pattern-matching query. You can specify either a *number* (any 64-bit integer greater than or equal to 1) or ALL (the maximum value that can be represented by a 64-bit integer). By default, MAXMATCHESPERQUERY=ALL.

MAXTIME=number

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

NODEFILTER=function-name

specifies the FCMP function for node filters.

NODEPAIRFILTER=function-name

specifies the FCMP function for node-pair filters.

OUTMATCHGRAPHLINKS=libref.data-table

specifies the links output data table for the induced subgraph of matches. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “[Using CAS Sessions and CAS Engine Librefs](#)” on page 2 in Chapter 1, “[Introduction](#).”

OUTMATCHGRAPHNODES=libref.data-table

specifies the nodes output data table for the induced subgraph of matches. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “[Using CAS Sessions and CAS Engine Librefs](#)” on page 2 in Chapter 1, “[Introduction](#).”

OUTMATCHLINKS=libref.data-table

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

OUTMATCHNODES=libref.data-table

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

OUTQUERYLINKS=libref.data-table

specifies the links query output data table to contain the generated queries. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

OUTQUERYNODES=libref.data-table

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

OUTSUMMARY=libref.data-table

specifies the summary output data table for pattern match queries. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

QUERYKEY=column

specifies the name of the data variable for the query key. The value of the variable can be numeric or character.

QUERYKEYAGGREGATE=TRUE | FALSE

specifies whether to aggregate nodes across query keys for creating the induced subgraph(s). You can specify the following values:

TRUE aggregates nodes across query keys for creating the induced subgraph(s).

FALSE does not aggregate nodes across query keys for creating the induced subgraphs(s).

By default, QUERYKEYAGGREGATE=TRUE.

PROJECTION Statement

PROJECTION < options > ;

The PROJECTION statement invokes an algorithm that calculates the network projection of the multiple-partition input graph and produces a single-partition output graph. The network projection problem is described in the section “[Network Projection](#)” on page 212.

You can specify the following *options*:

ADAMICADAR=TRUE | FALSE

specifies whether to calculate Adamic-Adar node similarity. You can specify the following values:

TRUE	calculates Adamic-Adar node similarity and saves the results in the output table that is specified in the OUTPROJECTIONLINKS= option.
FALSE	does not calculate Adamic-Adar node similarity.

By default, ADAMICADAR=FALSE.

COMMONNEIGHBORS=TRUE | FALSE

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

TRUE	calculates common neighbors node similarity and saves the results in the output table that is specified in the OUTPROJECTIONLINKS= option.
FALSE	does not calculate common neighbors node similarity.

By default, COMMONNEIGHBORS=FALSE.

COSINE=TRUE | FALSE

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

TRUE	calculates cosine node similarity and saves the results in the output table that is specified in the OUTPROJECTIONLINKS= option.
FALSE	does not calculate cosine node similarity.

By default, COSINE=FALSE.

DICE=TRUE | FALSE

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

TRUE	calculates Dice node similarity and saves the results in the output table that is specified in the OUTPROJECTIONLINKS= option.
FALSE	does not calculate Dice node similarity.

By default, DICE=FALSE.

DIRECTEDMETHOD=CONVERGING | DIVERGING | TRANSITIVE

specifies the method to use for projection of directed graphs. This option can be used only for directed graphs. You can specify the following values:

- CONVERGING** uses shared out-neighbors to define projected links.
- DIVERGING** uses shared in-neighbors to define projected links.
- TRANSITIVE** uses directed paths of length two to define projected links.

By default, DIRECTEDMETHOD=CONVERGING.

JACCARD=TRUE | FALSE

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

- TRUE** calculates Jaccard node similarity and saves the results in the output table that is specified in the OUTPROJECTIONLINKS= option.
- FALSE** does not calculate Jaccard node similarity.

By default, JACCARD=FALSE.

OUTNEIGHBORSLIST=libref.data-table

specifies the output data table to contain the lists of common neighbors between pairs of nodes. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “[Using CAS Sessions and CAS Engine Librefs](#)” on page 2 in Chapter 1, “[Introduction](#).”

OUTPROJECTIONLINKS=libref.data-table

specifies the output data table to contain the links of the projected graph. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “[Using CAS Sessions and CAS Engine Librefs](#)” on page 2 in Chapter 1, “[Introduction](#).”

OUTPROJECTIONNODES=libref.data-table

specifies the output data table to contain the nodes of the projected graph. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “[Using CAS Sessions and CAS Engine Librefs](#)” on page 2 in Chapter 1, “[Introduction](#).”

PARTITION=column

specifies the name of the data variable for the partition flag. The value of the variable must be 0, 1, or missing. The input graph is projected onto the set of nodes that have a partition flag of 1. The projected graph has links between node pairs that are connected via a common neighbor among the set of nodes that have a partition flag of 0. Nodes that have a missing value for the partition flag are excluded from the calculation.

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 219.

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 requires MAXREACH=1.

EACHSOURCE

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

MAXREACH=*number*

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

OUTCOUNTS=*libref.data-table*

specifies the output data table to contain the node counts in each reach network. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “[Using CAS Sessions and CAS Engine Librefs](#)” on page 2 in Chapter 1, “[Introduction](#).”

OUTREACHLINKS=*libref.data-table*

specifies the output data table to contain the links in each reach network. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “[Using CAS Sessions and CAS Engine Librefs](#)” on page 2 in Chapter 1, “[Introduction](#).”

OUTREACHNODES=*libref.data-table*

specifies the output data table to contain the nodes in each reach network. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “[Using CAS Sessions and CAS Engine Librefs](#)” on page 2 in Chapter 1, “[Introduction](#).”

SHORTESTPATH 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 226.

You can specify the following *options*:

MAXABSOBJGAP=*number*

MAXABSOLUTEOBJECTIVEGAP=*number*

specifies an acceptance criterion for the sum of link weights (CandidatePath) in paths relative to the sum of link weights in the shortest path (ShortestPath). The absolute objective gap is equal to

$$|\text{CandidatePath} - \text{ShortestPath}|$$

Any path whose absolute objective gap is greater than *number* is removed from the results. The value of *number* can be any nonnegative number. The default is the largest number that can be represented by a double. When the default is used, no paths are removed from the results.

MAXLINKWEIGHT=*number*

MAXPATHWEIGHT=*number*

specifies the maximum sum of link weights in a path. Any shortest path whose sum of link weights is greater than *number* is removed from the results. The default is the largest number that can be represented by a double. When the default is used, no paths are removed from the results.

MAXPATHSPERPAIR=*number*

specifies the maximum number of ranked paths to find for each source-sink pair. The algorithm finds at most *number* best paths between each source-sink pair. The paths that are returned are arranged in ascending order in the sum of link weights for each path. By default, MAXPATHSPERPAIR=1 and the algorithm finds one shortest path between each source-sink pair.

MAXRELOBJGAP=*number*

MAXRELATIVEOBJECTIVEGAP=*number*

specifies an acceptance criterion for the sum of link weights (CandidatePath) in paths relative to the sum of link weights in the shortest path (ShortestPath). The relative objective gap is equal to

$$\frac{|\text{CandidatePath} - \text{ShortestPath}|}{|\text{ShortestPath}| + 1\text{E}-10}$$

Any path whose relative objective gap is greater than *number* is removed from the results. The value of *number* can be any nonnegative number. The default is the largest number that can be represented by a double. When the default is used, no paths are removed from the results.

MINLINKWEIGHT=*number*

MINPATHWEIGHT=*number*

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

OUTPATHSLINKS=*libref.data-table*

OUT=*libref.data-table*

OUTPATHS=*libref.data-table*

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

OUTPATHSNODES=*libref.data-table*

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

OUTSUMMARY=*libref.data-table*

specifies the output data table to contain descriptive statistics of the finite shortest paths for each source. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

OUTWEIGHTS=*libref.data-table*

specifies the output data table to contain the total weight of the shortest path for each source-sink pair. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section “Using CAS Sessions and CAS Engine Librefs” on page 2 in Chapter 1, “Introduction.”

SEQUENCE=*column*

specifies which data variable in the nodes subset table defines the sequence of nodes to visit. The variable data type must be numeric. The values must define an ordering from lowest (visit first) to highest (visit last). The resulting path contains a node subsequence that matches the specified node sequence. You cannot use this SEQUENCE= option in combination with the SOURCE= or SINK= option in the SHORTESTPATH statement. In addition, the nodes subset table cannot contain a source or sink variable. To specify the source and sink nodes for the path, include them as the first and last nodes, respectively, in the sequence of nodes that you specify with the SEQUENCE= option.

SINK=*sink-node*

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

SOURCE=*source-node*

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

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 246.

You can specify the following *options*:

ASSORTATIVITYDEGREE=WEIGHT | UNWEIGHT | BOTH

calculates degree assortativity and specifies which type of calculation to perform. You can specify the following values:

WEIGHT	calculates degree assortativity by using the weighted graph.
UNWEIGHT	calculates degree assortativity by using the unweighted graph.
BOTH	calculates degree assortativity by using both the weighted and unweighted graphs.

If the input graph does not contain link weights, then ASSORTATIVITYDEGREE=WEIGHT and ASSORTATIVITYDEGREE=UNWEIGHT produce the same results.

ASSORTATIVITYNOMINAL=*column*

specifies the name of the data variable for computing the nominal assortativity. The variable can be numeric or character.

ASSORTATIVITYNUMERIC=*column*

specifies the name of the data variable for computing the numeric assortativity. The variable must be numeric.

BICONNECTEDCOMPONENTS

calculates information about biconnected components. You can use this option only for an undirected graph.

CLUSTERINGCOEFFICIENT**CLUSTERINGCOEF**

calculates information about clustering coefficients. You can use this option only for an undirected graph.

CONNECTEDCOMPONENTS

calculates information about connected components.

DIAMETERAPPROX=WEIGHT | UNWEIGHT | BOTH

calculates information about the approximate diameter and specifies which type of calculation to perform. Use this option when calculating the exact diameter (by calculating all shortest paths) is too computationally expensive. You can specify the following values:

WEIGHT	calculates the approximate diameter by using the weighted graph.
UNWEIGHT	calculates the approximate diameter by using the unweighted graph.
BOTH	calculates the approximate diameter by using both the weighted and unweighted graphs.

If the input graph does not contain link weights, then **DIAMETERAPPROX=WEIGHT** and **DIAMETERAPPROX=UNWEIGHT** produce the same results. This option works only for undirected graphs.

FINITEPATH

includes only finite values when calculating descriptive statistics that are related to shortest paths (eccentricity, diameter, and so on).

OUT=libref.data-table

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

SHORTESTPATH=WEIGHT | UNWEIGHT | BOTH

calculates information about shortest paths and specifies which type of calculation to perform. You can specify the following values:

WEIGHT	calculates shortest paths by using the weighted graph.
UNWEIGHT	calculates shortest paths by using the unweighted graph.
BOTH	calculates shortest paths by using both the weighted and unweighted graphs.

If the input graph does not contain link weights, then **SHORTESTPATH=WEIGHT** and **SHORTESTPATH=UNWEIGHT** produce the same results.

TOPOLOGICALSORT Statement

TOPOLOGICALSORT ;

TOPSORT ;

The **TOPOLOGICALSORT** statement calculates a topological ordering of the nodes of a directed acyclic input graph. For more information, see the section “Topological Sort” on page 254.

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 256.

You can specify the following *option*:

OUT=*libref.data-table*

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

UNLOADGRAPH Statement

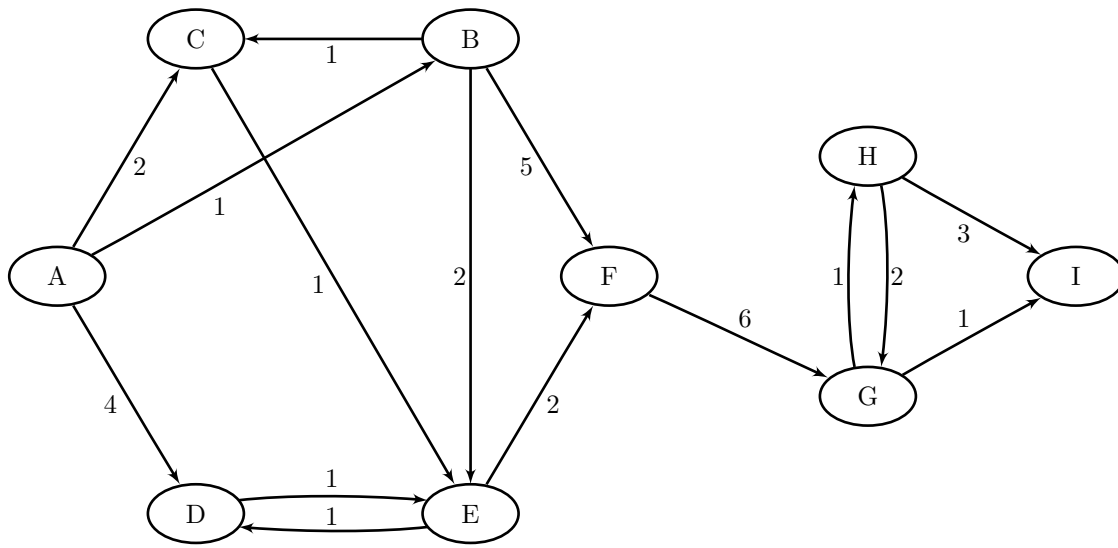
UNLOADGRAPH ;

The UNLOADGRAPH statement deletes the in-memory graph that is specified in the GRAPH= option in the PROC NETWORK statement. The UNLOADGRAPH statement is described further in the section “[Persistent Data Structures \(In-Memory Graphs\)](#)” on page 81.

Details: NETWORK Procedure

Graph Input and Output Data

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

Figure 2.7 Directed Graph

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

Links Input Data

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

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

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

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

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

The data variable names can have any values that you want. If you use nonstandard names for a built-in variable, you must identify the variables by using the LINKSVAR statement, as described in the section “LINKSVAR Statement” on page 44.

In addition, the links data table can contain any number of user-defined additional variables (attributes). The attributes that are defined in the VARS= option in the LINKSVAR statement are read into memory. The

attributes that are defined in the `VARSOOT=` option in the `LINKSVAR` statement are written to the resulting output data tables. If the `VARSOOT=` option is unspecified, all attribute variables that are specified in the `VAR=` option are written to the resulting output data tables. For example, the two data tables that are created by the following DATA steps identify the same graph:

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

data mylib.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 = mylib.LinkSetInA;
run;

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

The directed graph G that is shown in [Figure 2.7](#) can be represented by the links data table, `mylib.LinkSetIn`, that is created by the following DATA step:

```
data mylib.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      = mylib.LinkSetIn
    outNodes   = mylib.NodeSetOut
    outLinks   = mylib.LinkSetOut;
```

```
run;
```

The output data table mylib.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 Directed Graph

node
A
B
C
D
E
F
G
H
I

The output data table mylib.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 Directed Graph

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

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

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


```

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

```

The progress of the procedure is shown in [Figure 2.10](#). The log shows the number of links that were declared as multilinks and aggregated.

Figure 2.10 PROC NETWORK Log: Links Data Table of an Undirected Graph

```

NOTE: -----
NOTE: Running NETWORK.
NOTE: -----
NOTE: The number of nodes in the input graph is 9.
NOTE: The number of links in the input graph is 15.
NOTE: The Cloud Analytic Services server processed the request in 0.097757 seconds.
NOTE: The data set MYLIB.NODESETOUT has 9 observations and 1 variables.
NOTE: The data set MYLIB.LINKSETOUT has 15 observations and 3 variables.

```

The output data table `mylib.NodeSetOut` is equivalent to the one shown in [Figure 2.8](#). However, the new links data table `mylib.LinkSetOut`, shown in [Figure 2.11](#), contains two fewer links than before, because multilinks are aggregated.

Figure 2.11 Links Data Table of an 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	D	1
10	E	F	2
11	F	G	6
12	G	H	1
13	G	I	1
14	H	G	2
15	H	I	3

The **MULTILINKS=** option can be used to include (rather than aggregate) multilinks. For more information about this option, see the section “[Multigraphs](#)” on page 68.

Nodes Input Data

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

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

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

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

- `weight`: the node weight (must be numeric)

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

You can specify any value that you want for the data table variable name. If you use a nonstandard name for a built-in variable, you must identify the variable by using the `NODESVAR` statement, as described in the section “[NODESVAR Statement](#)” on page 49.

In addition, the nodes data table can contain any number of user-defined additional variables (attributes). The attributes that are defined in the `VAR=` option in the `NODESVAR` statement are read into memory. The attributes that are defined in the `VAROUT=` option in the `NODESVAR` statement are written to the resulting output data tables. If the `VAROUT=` option is unspecified, all attribute variables that are specified in the `VAR=` option are written to the resulting output data tables.

The data table that you specify in the `LINKS=` option defines the set of nodes that are incident to some link. If the graph contains a node that has no links (called a *singleton* node), then you must define this node in the `NODES=` data table. The following statements produce a graph that has three links but four nodes, including the singleton node D:

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

data mylib.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.

The graph is defined as the union of the set of nodes that are specified in the nodes data table and the links data table. The associated attributes for nodes in the links data table that are not specified in the nodes data table default to 0 (for numeric attributes) or to an empty character string (for character attributes).

Nodes Subset Input Data

For some algorithms, you might want to process only a subset of the nodes that appear in the input graph. You can accomplish this by using the NODESSUBSET= option in the PROC NETWORK statement. You can use the nodes subset data table in conjunction with the PATH, SHORTESTPATH, NODESIMILARITY, or REACH statement. (See the sections “[Path Enumeration](#)” on page 164, “[Shortest Path](#)” on page 226, “[Node Similarity](#)” on page 152, and “[Reach \(Ego\) Network](#)” on page 219, respectively.) The nodes subset data table is expected to contain some combination of the following variables:

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

[Table 2.5](#) shows how PROC NETWORK processes nodes for each algorithm type. The missing indicator (.) can also be used in place of 0 to designate that a node is not to be processed.

Table 2.5 Determining How to Process a Node

Algorithm Type	Variable Designations	Example Shown In:
(Shortest) path, Node similarity	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 226 or “ Example 2.17: Node Similarity for Link Prediction ” on page 335
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 219

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

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

The data table `mylib.NodeSubSetIn` indicates that you want to process the following:

- the reach network from the subgraph defined by node *A*
- the reach network from the subgraph defined by nodes *F* and *E*
- the (shortest) paths (or node similarity) for the source-sink pairs in $\{A, E\} \times \{F\}$ (the crossproduct of subsets $\{A, E\}$ and $\{F\}$)

Multigraphs

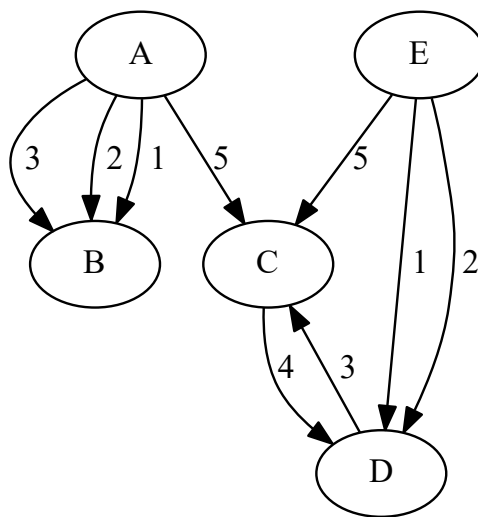
A *multigraph* is a graph that allows multiple (also called parallel) links between nodes (called *multilinks*). A graph that has no multilinks is called a *simple* graph (or just a graph). You can specify whether to include multilinks (`MULTILINKS=TRUE`) or to aggregate them (`MULTILINKS=FALSE`) when an input graph is read. For `MULTILINKS=FALSE`, each multilink is aggregated into one link that uses the minimum of each attribute value. That is, a multigraph is transformed into a simple graph. By default, `MULTILINKS=TRUE` (multilinks are included for algorithms that support them).

Aggregating links (`MULTILINKS=FALSE`) implies a performance cost. When multilinks are aggregated into a simple graph, the in-memory graph representation requires more memory and can take longer to build, relative to using `MULTILINKS=TRUE`. However, if your data contain many redundant parallel links, then using `MULTILINKS=FALSE` can improve overall performance by reducing the size of the working graph.

For the [clique enumeration algorithm](#), multigraphs are not supported. The clique enumeration algorithm uses `MULTILINKS=FALSE` by default and issues an error message if you specify `MULTILINKS=TRUE`.

Consider the directed multigraph shown in [Figure 2.12](#).

Figure 2.12 Directed Multigraph



The links data table is created by the following DATA step:

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

The following statements read in this graph, declare it as a directed (simple) graph, and output the resulting links:

```
proc network
    direction = directed
    multiLinks = false
    links      = mylib.LinkSetIn
    outLinks   = mylib.LinkSetOut;
run;
```

The progress of the procedure is shown in [Figure 2.13](#). The log now shows the number of links that were declared as multilinks and aggregated.

Figure 2.13 PROC NETWORK Log: Links Data Table of a Directed Graph

```
NOTE: -----
NOTE: Running NETWORK.
NOTE: -----
NOTE: Multilink aggregation removed 3 links from the input graph.
NOTE: The number of nodes in the input graph is 5.
NOTE: The number of links in the input graph is 6.
NOTE: The Cloud Analytic Services server processed the request in 0.076402 seconds.
NOTE: The data set MYLIB.LINKSETOUT has 6 observations and 3 variables.
```

The output data table mylib.LinkSetOut, shown in [Figure 2.14](#), contains the transformed (simple) graph from the input multigraph with attribute values aggregated (using the minimum attribute).

Figure 2.14 Links Data Table of a Directed Graph

Obs	from	to	weight
1	A	B	1
2	A	C	5
3	C	D	4
4	D	C	3
5	E	C	5
6	E	D	1

The following statements read in this graph, declare it as a directed multigraph, and output the resulting links:

```
proc network
  direction = directed
  multiLinks = true
  links      = mylib.LinkSetIn
  outLinks   = mylib.LinkSetOut;
run;
```

The progress of the procedure is shown in [Figure 2.15](#). In this case, no links are aggregated and the resulting output data table `mylib.LinkSetOut` is equivalent to the input data table `mylib.LinkSetIn`.

Figure 2.15 PROC NETWORK Log: Links Data Table of a Directed Multigraph

```
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 9.
NOTE: The Cloud Analytic Services server processed the request in 0.074422 seconds.
NOTE: The data set MYLIB.LINKSETOUT has 9 observations and 3 variables.
```

When you want to transform your multigraph into a simple graph with finer control over how the aggregation step is performed, one possible approach is to use the FEDSQL procedure. The following statements use PROC FEDSQL to aggregate multilinks by summing their weights:

```
proc fedsql sessref=mysess;
  create table LinkSetAgg as
  select "from", "to", sum(weight) as weight
  from LinkSetIn
  group by "from", "to";
quit;
```

The resulting output data table `mylib.LinkSetAgg`, shown in [Figure 2.16](#), can now be used as a simple graph in a subsequent PROC NETWORK call. Using PROC FEDSQL to do the transformation of a multigraph to a simple graph provides greater flexibility in defining how the aggregation step is performed. For example, in place of the `sum` operator, you can use any of the operators that PROC FEDSQL supports. For more information about PROC FEDSQL, see *SAS Viya: FedSQL Programming for SAS Cloud Analytic Services*.

Figure 2.16 Links Data Table after Aggregation

Obs	from	to	WEIGHT
1	E	D	3
2	D	C	3
3	C	D	4
4	A	C	5
5	E	C	5
6	A	B	6

Self-Links

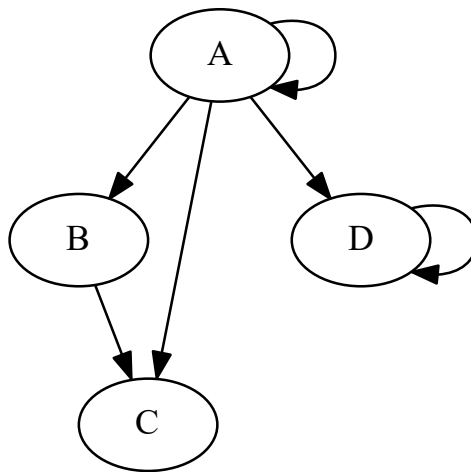
A *self-link* is a link for which the *from* node and *to* node are the same. You can specify whether to include self-links (SELFLINKS=TRUE) or to remove them (SELFLINKS=FALSE) when an input graph is read. By default, self-links are not removed.

For certain algorithms in PROC NETWORK, self-links are not supported. The following algorithm issues an error message if you specify SELFLINKS=TRUE:

- [CLIQUE](#)

Consider the directed graph shown in [Figure 2.17](#).

Figure 2.17 Directed Graph



The links data table is created by the following DATA step:

```

data mylib.LinkSetIn;
  input from $ to $;
  datalines;
A A
A B
A C
A D
B C
D D
;

```

The following statements read in this graph, declare it as a directed graph that does not allow self-links, and output the resulting links:

```
proc network
  direction = directed
  selfLinks = false
  links      = mylib.LinkSetIn
  outLinks   = mylib.LinkSetOut;
run;
```

The progress of the procedure is shown in [Figure 2.18](#). The log shows the number of self-links that were ignored and therefore removed.

Figure 2.18 PROC NETWORK Log: Links Data Table of a Directed Graph

```
NOTE: -----
NOTE: Running NETWORK.
NOTE: -----
NOTE: The graph contains 2 self-links that are ignored.
NOTE: The number of nodes in the input graph is 4.
NOTE: The number of links in the input graph is 4.
NOTE: The Cloud Analytic Services server processed the request in 0.07466 seconds.
NOTE: The data set MYLIB.LINKSETOUT has 4 observations and 2 variables.
```

The output data table mylib.LinkSetOut, shown in [Figure 2.19](#), contains the remaining links after self-links have been removed.

Figure 2.19 Links Data Table of a Directed Graph

	Obs	from	to
1	A	B	
2	A	C	
3	A	D	
4	B	C	

The following statements read in the graph that is shown in [Figure 2.17](#), declare it as a directed graph that allows self-links, and output the resulting links:

```
proc network
  direction = directed
  selfLinks = true
  links      = mylib.LinkSetIn
  outLinks   = mylib.LinkSetOut;
run;
```

The progress of the procedure is shown in [Figure 2.20](#). In this case, no links are removed and the resulting output data table mylib.LinkSetOut is equivalent to the input data table mylib.LinkSetIn.

Figure 2.20 PROC NETWORK Log: Links Data Table of a Directed 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 6.
NOTE: The Cloud Analytic Services server processed the request in 0.067046 seconds.
NOTE: The data set MYLIB.LINKSETOUT has 6 observations and 2 variables.
```

Output Variables

You can write any link or node attribute to the output results table by using the `VARSOOT=` option in the `LINKSVAR` or `NODESVAR` statement. By default, the variables that are carried over to the output results are defined by the corresponding `VAR=` option. Defining the additional variables to output is supported for all output tables except the table that is specified in the `OUTNODES=` option when this table is used in conjunction with a distributed graph algorithm. The list of algorithms that operate on a distributed graph is described in the section “[Execution Modes and Data Movement](#)” on page 78.

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

```
data mylib.NodeSetIn;
    input node $ attrStr $ attrNum;
    datalines;
A ThisIsA    13
B B          1
C LabelC     55.5
D NodeD      7
;
data mylib.LinkSetIn;
    input from $ to $ weight attrStr1 $ attrStr2 $12.;
    datalines;
A B 1 Link1 555-789-1234
A C 2 Link2 556-453-7456
B C 1 Link3 800-123-7787
;
```

The following statements read in this graph and all of its node and link attributes (defined in the `VAR=` options) and output a subset of those attributes (defined in the `VARSOOT=` options):

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

```
run;
```

The nodes data table `mylib.NodeSetOut`, shown in Figure 2.21, contains only the additional attributes that are defined in the `VARSOOT=` option in the `NODESVAR` statement.

Figure 2.21 Nodes Data Table with Output Variables

Obs	node	attrNum
1	A	13.0
2	B	1.0
3	C	55.5
4	D	7.0

The links data table `mylib.LinkSetOut`, shown in Figure 2.22, contains only the additional attributes that are defined in the `VARSOOT=` option in the `LINKSVAR` statement.

Figure 2.22 Links Data Table with Output Variables

Obs	from	to	weight	attrStr2
1	A	B	1	555-789-1234
2	A	C	2	556-453-7456
3	B	C	1	800-123-7787

Standardized Labels Input

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 mylib.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      = mylib.LinkSetIn
  outNodes   = mylib.NodeSetOut
  outLinks   = mylib.LinkSetOut;
run;
```

The log is shown in [Figure 2.23](#).

Figure 2.23 PROC NETWORK Log: 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 0.074414 seconds.
NOTE: The data set MYLIB.NODESETOUT has 4 observations and 1 variables.
NOTE: The data set MYLIB.LINKSETOUT has 3 observations and 3 variables.
```

The output data table mylib.NodeSetOut, shown in [Figure 2.24](#), contains the unique numeric node labels, {0, 1, 3, 5}.

Figure 2.24 Nodes Data Table of a 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      = mylib.LinkSetIn
  outNodes   = mylib.NodeSetOut
  outLinks   = mylib.LinkSetOut;
run;
```

The log that results from using standardized labels is shown in [Figure 2.25](#).

Figure 2.25 PROC NETWORK Log: Undirected Graph Using Standardized Labels

```
NOTE: -----
NOTE: Running NETWORK.
NOTE: -----
NOTE: The number of nodes in the input graph is 6.
NOTE: The number of links in the input graph is 3.
NOTE: The Cloud Analytic Services server processed the request in 0.078071 seconds.
NOTE: The data set MYLIB.NODESETOUT has 6 observations and 1 variables.
NOTE: The data set MYLIB.LINKSETOUT has 3 observations and 3 variables.
```

The output data table mylib.NodeSetOut, shown in [Figure 2.26](#), now contains all node labels from 0 to 5, based on the assumptions when you use the STANDARDIZEDLABELS option.

Figure 2.26 Nodes Data Table of a Directed Graph

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

Standardized Labels Output

You can specify the `STANDARDIZEDLABELSOUT` option in the `PROC NETWORK` statement to generate a mapping between node labels and node identifiers that satisfies the requirements needed to use the `STANDARDIZEDLABELS` option in subsequent calls. Specifying the `STANDARDIZEDLABELSOUT` option can be useful to avoid repeated calls to the processing stage where the mapping is created. The standardized labels are written to the output links data table (or to the output nodes data table or to both) that you specify in the `OUTLINKS=` (`OUTNODES=`) option in the `PROC NETWORK` statement.

The `OUTLINKS=` data table contains the following columns in addition to the variables that are described in the section “[Links Input Data](#)” on page 62:

- `fromId`: the *from* node identifier (a mapping to the label in the *from* column)
- `told`: the *to* node identifier (a mapping to the label in the *to* column)

The `OUTNODES=` data table contains the following column in addition to the variables that are described in the section “[Nodes Input Data](#)” on page 66:

- `nodeId`: the node identifier (a mapping to the label in the *node* column)

Consider the following nodes and links data tables:

```
data mylib.NodeSetIn;
    input node $;
    datalines;
Z
;
data mylib.LinkSetIn;
    input from $ to $;
    datalines;
B C
B D
X Y
;
```

The following statements create the standardized mapping and output the results in the data tables `mylib.NodeSetSLabels` and `mylib.LinkSetSLabels`:

```

proc network
  nodes      = mylib.NodeSetIn
  links      = mylib.LinkSetIn
  outNodes   = mylib.NodeSetSLabels
  outLinks   = mylib.LinkSetSLabels
  standardizedLabelsOut;
run;

```

The output data table `mylib.NodeSetSLabels`, shown in [Figure 2.27](#), contains the node mapping.

Figure 2.27 Nodes Data Table with Standardized Label Mapping

Obs	node	nodeId
1	B	0
2	C	1
3	D	2
4	X	3
5	Y	4
6	Z	5

The output data table `mylib.LinkSetSLabels`, shown in [Figure 2.28](#), contains the node mapping that is applied to each link.

Figure 2.28 Links Data Table with Standardized Label Mapping

Obs	from	to	fromId	toId
1	B	C	0	1
2	B	D	0	2
3	X	Y	3	4

In subsequent calls, the mapping can be used with the `STANDARDIZEDLABELS` option, and the `LINKSVAR` (and `NODESVAR`) statements, to avoid the expense of the input processing stage. It also might be useful to use the `VARS=` option in the `LINKSVAR` (and `NODESVAR`) statements, as described in the section “[Output Variables](#)” on page 73, to write the mapping to the output results, as follows:

```

proc network
  standardizedLabels
  nodes      = mylib.NodeSetSLabels
  links      = mylib.LinkSetSLabels
  outNodes   = mylib.NodeSetOut
  outLinks   = mylib.LinkSetOut;
  nodesVar
    node     = nodeId
    vars     = (node);
  linksVar
    from     = fromId
    to       = toId
    vars     = (from to);
  connectedComponents;
run;

```

The output data table `mylib.NodeSetOut`, shown in [Figure 2.29](#), contains the assignments of nodes to connected components, in addition to the label (identifier) mappings.

Figure 2.29 Nodes Data Table Including Connected Components and Mappings

Obs	nodeid	node	concomp
1	0	B	1
2	1	C	1
3	2	D	1
4	3	X	2
5	4	Y	2
6	5	Z	3

The output data table `mylib.LinkSetOut`, shown in [Figure 2.30](#), contains the assignments of links to connected components, in addition to the label (identifier) mappings.

Figure 2.30 Links Data Table Including Connected Components and Mappings

Obs	fromid	told	from	to	concomp
1	0	1	B	C	1
2	0	2	B	D	1
3	3	4	X	Y	2

Determinism

Many algorithms are sensitive to the order in which PROC NETWORK loads the data. Reading data tables in the same order at each invocation is not guaranteed. If the order of the nodes or links is different, the final result might change. By default, PROC NETWORK ensures that each invocation (with the same machine configuration and parameter settings) produces the same final result. However, this comes at a performance cost, which can be avoided by specifying `DETERMINISTIC=FALSE` in the PROC NETWORK statement. Specifying `DETERMINISTIC=FALSE` might improve performance, but the final results might differ. In some cases, this difference is simply a permutation of identifiers (for example, connected components). In other cases, when the algorithm uses local optimization (for example, community detection), the final result might be a local (or alternative) solution.

Execution Modes and Data Movement

When you run PROC NETWORK, the execution mode and the underlying data movement that must be implemented to support that execution mode depend on which algorithm you select.

The data movement and execution modes for each algorithm are listed in [Table 2.6](#). The following abbreviations are used in [Table 2.6](#) for processing mode:

- SM: single machine
- MM: multiple machines

- MT: multithreaded execution

For a single-machine CAS server, there is no data movement. The algorithm runs on the same machine where the data reside. For a multiple-machine CAS server, the assumption is that the data reside in parts on one or more of the machines in the server. The following phrases are used in Table 2.6 to describe data movements for a multiple-machine CAS server:

- Moved to SM: Some algorithms run only in single-machine mode. In such cases, one particular machine (chosen randomly) is assigned the role of the processing machine, and the data from all the other machines are moved over to this processing machine.
- Repeated on MM: Some algorithms use multiple machines, but each machine requires a global view of the input data. In such cases, each data part is repeated on all machines. Each machine processes a portion of the work across the entire graph. The resulting output tables are distributed tables.
- Shuffled across MM: Some algorithms use multiple machines and require only a portion of the data (distributed graphs). However, because the original data are usually randomly distributed, the first step is to shuffle data between machines such that the data are appropriately aligned for the particular algorithm's needs. When the data are aligned correctly, each machine processes a part of the data and then iteratively merges results across the grid to obtain the final result. Again, the resulting output tables are distributed tables.
- None: Some algorithms require no data movement and process the original data (randomly distributed) directly.

Table 2.6 Execution Modes and Data Movement

Statement and Options	Data Movement	Processing Mode
PROC NETWORK		
DISTRIBUTED=FALSE	Moved to SM	SM (MT)
DISTRIBUTED=TRUE	None	MM (MT)
BICONNECTEDCOMPONENTS	Moved to SM	SM
BY	Shuffled across MM	MM (MT)
CENTRALITY		
AUTH=	Moved to SM	SM (MT)
BETWEEN=	Repeated on MM	MM (MT)
CLOSE=	Repeated on MM	MM (MT)
DEGREE=	Moved to SM	SM
DEGREE= (with DISTRIBUTED=TRUE)	None	MM
INFLUENCE=	Moved to SM	SM
Otherwise	Moved to SM	SM (MT)
CLIQUE	Moved to SM	SM (MT)
COMMUNITY		
ALGORITHM=LABELPROPAGATION [†]		
with LABELUPDATESMODE=ASYNCHRONOUS	Moved to SM	SM
with LABELUPDATESMODE=SYNCHRONOUS	Shuffled across MM	MM (MT)
ALGORITHM=LOUVAIN	Moved to SM	SM

Table 2.6 continued

Statement and Options	Data Movement	Processing Mode
CONNECTEDCOMPONENTS		
ALGORITHM=DFS	Moved to SM	SM
ALGORITHM=UNIONFIND (with DISTRIBUTED=TRUE)	None	MM
Otherwise	Moved to SM	SM (MT)
CORE	Moved to SM	SM (MT)
CYCLE		
ALGORITHM=BACKTRACK	Moved to SM	SM
ALGORITHM=BUILD	Moved to SM	SM (MT)
NODESIMILARITY	Repeated on MM	MM (MT)
PATH	Repeated on MM	MM (MT)
PATTERNMATCH		
With QUERYKEY=	Repeated on MM	MM (MT)
Without QUERYKEY=	Moved to SM	SM (MT)
PROJECTION	Repeated on MM	MM (MT)
REACH	Repeated on MM	MM (MT)
SHORTESTPATH	Repeated on MM	MM (MT)
SUMMARY	Moved to SM	SM (MT)
TOPOLOGICALSORT	Moved to SM	SM
TRANSITIVECLOSURE	Moved to SM	SM (MT)

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

In addition, on each machine, some algorithms (in addition to the input phase) take advantage of multicore chip technology by executing on multiple threads simultaneously. You can use the `NTHREADS=number` 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). Specifying a *number* greater than the number of available cores might hurt performance. Specifying a high *number* does not guarantee shorter solution time; the actual change in solution time depends on the computing hardware and the scalability of the underlying algorithms. In some circumstances, the NETWORK procedure might use fewer threads than the specified *number* because the procedure's internal algorithms have determined that a smaller number is preferable.

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

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, the CAS session is configured for 16 worker nodes, each with 80 cores, unless otherwise noted. For general information about CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*.

Persistent Data Structures (In-Memory Graphs)

For large-scale graphs, loading the graph input data tables and building in-memory data structures can be computationally expensive. When the input data are not changing frequently and your workflow consists of multiple analyses, it might be more efficient to retain the in-memory data structures. You can do this by using the **LOADGRAPH** statement along with any other options that define the graph input. The options that define the graph input include the **DIRECTION=**, **LINKS=**, **NODES=**, **MULTILINKS=**, **SELFLINKS=**, and **STANDARDIZEDLABELS** options in the **PROC NETWORK** statement, in addition to the **LINKSVAR** and **NODESVAR** statements.

Loading and Unloading the Graph

Loading the graph (via the **LOADGRAPH** statement) consists of reading the graph input data tables and storing a standard representation of the graph in memory as part of the current CAS session. The resulting macro variable, **_NETWORK_**, includes **GRAPH**, which is a reference identifier for the in-memory graph. You can then use this identifier in the **GRAPH=** option in the **PROC NETWORK** statement for subsequent calls to any supporting algorithm without needing to read from tables. Subsequent calls that include the **GRAPH=** option directly use the in-memory graph that was defined in the loading stage. This means that you cannot use options that are related to defining the graph in conjunction with the **GRAPH=** option.

When the **GRAPH=** option is used, additional data structures might need to be built, depending on the algorithm chosen, when a particular algorithm is first invoked. In some cases, this is an additional one-time expense. Subsequent calls can then use these data structures directly. For this reason, a first call that includes the **GRAPH=** option to some algorithms can sometimes take longer than subsequent calls.

You can load any number of graph input tables into memory. You can delete an in-memory graph and its persistent data structures by using the **UNLOADGRAPH** statement or by closing the CAS session.

OUTGRAPHLIST= Option

The **OUTGRAPHLIST=** option in the **PROC NETWORK** statement produces a data table that contains summary information about the in-memory graphs that are currently loaded (or unloaded) in the current CAS session. This data table contains the following columns:

- **graph**: in-memory graph identifier
- **createTime**: the creation time of the in-memory graph
- **loaded**: 1 if the graph has been loaded; 0 if the graph has been unloaded
- **direction**: the direction of the in-memory graph
- **nodes**: the number of nodes in the in-memory graph
- **links**: the number of links in the in-memory graph
- **multiLinks**: 1 if the in-memory graph load used **MULTILINKS=TRUE**; 0 otherwise
- **selfLinks**: 1 if the in-memory graph load used **SELFLINKS=TRUE**; 0 otherwise
- **standardizedLabels**: 1 if the in-memory graph load used **STANDARDIZEDLABELS**; 0 otherwise

Supported Algorithms

The section “[Execution Modes and Data Movement](#)” on page 78 describes the execution modes for each algorithm and feature combination. Using persistent data structures is supported for all algorithms where data are moved to a single machine (Moved to SM) or repeated on multiple machines (Repeated on MM). Persistence of data structures is not supported when data are shuffled across multiple machines (Shuffled across MM) nor for algorithms that move no data (None).

Using Persistent Data Structures with the Shortest Path Algorithm

As an example, consider the road network that is described in the section “[Road Network Shortest Path](#)” on page 11 and is defined by the data table `mylib.LinkSetInRoadNC10am`. To find the route that yields the shortest path between `source="614CapitalBlvd"` and `sink="SASCampusDrive"`, you can use the following statements:

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

These statements read the graph input from the table `mylib.LinkSetInRoadNC10am`, build the in-memory data structures that are required to run the shortest path algorithm, calculate the shortest path, and then delete the in-memory data structures that were needed to represent the graph data.

Next, you might want to calculate the shortest path between two other locations—for example, `source="US70W/US440W"` and `sink="SASCampusDrive"`. In this case, you would use the following statements:

```
proc network
  logLevel          = aggressive
  links             = mylib.LinkSetInRoadNC10am;
  linksVar
    from            = start_inter
    to              = end_inter
    weight          = time_to_travel;
  shortestPath
    outPathsLinks   = mylib.ShortPath
    source          = "US70W/US440W"
    sink            = "SASCampusDrive";
run;
```

These statements read the graph input from the table `mylib.LinkSetInRoadNC10am`, build the in-memory data structures, calculate the shortest path, and then delete the in-memory data structures. In this case, the graph input table and the in-memory data structures that are built are the same in both executions. The processing

time that is needed to execute the second call can be reduced by using the LOADGRAPH statement to store the in-memory data structures.

First, load the graph by using the following statements:

```
proc network
  logLevel      = aggressive
  links         = mylib.LinkSetInRoadNC10am
  outGraphList  = mylib.OutGraphList;
  linksVar
    from        = start_inter
    to          = end_inter
    weight      = time_to_travel;
  loadGraph;
run;
%put &_NETWORK_;
```

The macro variable result is written to the log, as shown in [Figure 2.31](#).

Figure 2.31 PROC NETWORK Log: Loading Graph into Memory

```
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 graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 11.
NOTE: The Cloud Analytic Services server processed the request in 0.097988 seconds.
NOTE: The data set MYLIB.OUTGRAPHLIST has 1 observations and 9 variables.
STATUS=OK  PROBLEM_TYPE=LOADGRAPH  SOLUTION_STATUS=OK  GRAPH=0  CREATETIME=17OCT2025:17:16:09
CPU_TIME=2.17  REAL_TIME=0.10
```

The macro variable indicates that the graph identifier (GRAPH) for the in-memory graph is 0. The following statements extract the identifier from the `_NETWORK_` macro variable into the `_GRAPH_` macro variable:

```
%macro GetValue(mac=, item=);
  %let prs = %sysfunc(prxparse(m/\b&item=/i));
  %if %sysfunc(prxmatch(&prs, &&mac)) %then %do;
    %let prs = %sysfunc(prxparse(s/.*\b&item=([^\s]+).*/$1/i));
    %let return_val = %sysfunc(prxchange(&prs, 1, &&mac));
    &return_val
  %end;
  %else %do;
    %put ERROR: Cannot find &item!;
    .
  %end;
%mend GetValue;
%let _GRAPH_ = %GetValue(mac=_NETWORK_, item=GRAPH);
```

The graph identifier and several other pieces of summary information about the in-memory graph are also contained in the output data table `mylib.OutGraphList`, as shown in [Figure 2.32](#).

Figure 2.32 Summary Information about the In-Memory Graphs

graph	createTime	loaded	direction	nodes	links	multiLinks	selfLinks	standardizedLabels
0	17OCT2025:17:16:09	1	Undirected	10	11	1	1	0

Next, the following statements use the in-memory graph that was loaded in the previous PROC NETWORK call to find the route that yields the shortest path between `source="614CapitalBoulevard"` and `sink="SASCampusDrive"`:

```
proc network
  logLevel          = aggressive
  graph              = &_GRAPH_;
  shortestPath
    outPathsLinks = mylib.ShortPath
    source         = "614CapitalBlvd"
    sink           = "SASCampusDrive";
run;
```

These statements point to the in-memory graph that is referenced by identifier `_GRAPH_=0`, build any additional data structures needed (since this is the first call to the shortest path algorithm), and then calculate the shortest path. The progress of the procedure is shown in [Figure 2.33](#).

Figure 2.33 PROC NETWORK Log: Shortest Path Using an In-Memory Graph

```
NOTE: -----
NOTE: -----
NOTE: Running NETWORK.
NOTE: -----
NOTE: -----
NOTE: Building the input graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 11.
NOTE: Processing shortest paths problem using 1200 threads across 15 machines.
NOTE: Processing the shortest paths problem between 1 source nodes and 1 sink nodes.

          Real
Algorithm    Sources    Complete    Time
shortestPath         1      100%      0.01
NOTE: Processing the shortest paths problem used 0.01 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.062842 seconds.
NOTE: The data set MYLIB.SHORTPATH has 5 observations and 6 variables.
```

[Figure 2.34](#) displays the output data table `mylib.ShortPath`, which shows the best route to take to minimize travel time between 614 Capital Boulevard and SAS Campus Drive.

Figure 2.34 Shortest Path for Road Network between 614 Capital Boulevard and SAS Campus Drive

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

Next, to calculate the shortest path between `source="US70W/US440W"` and `sink="SASCampusDrive"`, use the following statements:

```
proc network
  logLevel          = aggressive
  graph             = &_GRAPH_;
  shortestPath
    outPathsLinks = mylib.ShortPath
    source        = "US70W/US440W"
    sink          = "SASCampusDrive";
run;
```

These statements again point to the same in-memory graph (with no additional setup work required) and calculate the shortest path. The progress of the procedure is shown in [Figure 2.35](#).

Figure 2.35 PROC NETWORK Log: Shortest Path Using an In-Memory Graph

```
NOTE: -----
NOTE: -----
NOTE: Running NETWORK.
NOTE: -----
NOTE: -----
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 11.
NOTE: Processing shortest paths problem using 1200 threads across 15 machines.
NOTE: Processing the shortest paths problem between 1 source nodes and 1 sink nodes.

                                     Real
Algorithm          Sources    Complete    Time
shortestPath              1         100%    0.01
NOTE: Processing the shortest paths problem used 0.01 (cpu: 0.01) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.060532 seconds.
NOTE: The data set MYLIB.SHORTPATH has 4 observations and 6 variables.
```

[Figure 2.36](#) displays the output data table `mylib.ShortPath`, which shows the best route to take to minimize travel time between US70W/US440W and SAS Campus Drive.

Figure 2.36 Shortest Path for Road Network between US70W/US440W and SAS Campus Drive

order	start_inter	end_inter	time_to_travel
1	US70W/US440W	US440W/RaleighExpy	2.70000
2	US440W/RaleighExpy	RaleighExpy/US40W	3.00000
3	RaleighExpy/US40W	US40W/HarrisonAve	1.41818
4	US40W/HarrisonAve	SASCampusDrive	1.20000
			8.31818

The following statements load an additional graph, from the table `mylib.LinkSetInRoadNC5pm`:

```
proc network
  logLevel      = aggressive
  links         = mylib.LinkSetInRoadNC5pm
  outGraphList  = mylib.OutGraphList;
  linksVar
    from        = start_inter
    to          = end_inter
    weight      = time_to_travel;
  loadGraph;
run;
```

The summary information about the in-memory graphs is contained in the output data table `mylib.OutGraphList`, as shown in [Figure 2.37](#).

Figure 2.37 Summary Information about the In-memory Graphs

graph	createTime	loaded	direction	nodes	links	multiLinks	selfLinks	standardizedLabels
0	17OCT2025:17:16:09	1	Undirected	10	11	1	1	0
1	17OCT2025:17:16:12	1	Undirected	10	11	1	1	0

Now, you can directly execute any supported algorithm for either graph (**GRAPH=0** or **GRAPH=1**), without reading in the links table again.

Finally, to delete the first in-memory graph (**GRAPH=0**), you can use the following statements:

```
proc network
  graph         = &_GRAPH_
  outGraphList  = mylib.OutGraphList;
  unloadGraph;
run;
```

The output data table `mylib.OutGraphList`, shown in [Figure 2.38](#), now indicates that the in-memory graph that has identifier 0 is no longer loaded.

Figure 2.38 Summary Information about the In-memory Graphs

graph	createTime	loaded	direction	nodes	links	multiLinks	selfLinks	standardizedLabels
0	17OCT2025:17:16:09	0	.	.	.	.	.	.
1	17OCT2025:17:16:12	1	Undirected	10	11	1	1	0

Numeric Limitations

Extremely large or extremely small numerical values might cause computational difficulties for some of the algorithms in PROC NETWORK. For this reason, each algorithm restricts the magnitude of the data values to a particular threshold number. If the user data values exceed this threshold, PROC NETWORK issues an error message. The value of the threshold limit is different for each algorithm and depends on the operating environment. The threshold limits are listed in Table 2.7, where M is defined as the largest number that can be represented by a double.

Table 2.7 Threshold Limits by Statement

Statement (and Options)	Links		Nodes
	weight	auxweight	weight
CENTRALITY			
AUTH=	1E20		
BETWEEN=	$\sqrt{M}$	$\sqrt{M}$	
CLOSE=	$\sqrt{M}$	$\sqrt{M}$	
DEGREE=	$\sqrt{M}$		
EIGEN=	1E20		
HUB=	1E20		
INFLUENCE=	$\sqrt{M}$		$\sqrt{M}$
PAGERANK=	1E20		
CLIQUE	$\sqrt{M}$		$\sqrt{M}$
COMMUNITY	1E50		
CYCLE	$\sqrt{M}$		$\sqrt{M}$
NODESIMILARITY			
VECTOR=TRUE	$\sqrt{M}$		
COSINE=TRUE	1E20		
PATH	$\sqrt{M}$		$\sqrt{M}$
PROJECTION			
COSINE=TRUE	1E20		
REACH			$\sqrt{M}$
SHORTESTPATH	$\sqrt{M}$	$\sqrt{M}$	
SUMMARY			
ASSORTATIVITYNUMERIC=			$\sqrt{M}$
DIAMETERAPPROX=	$\sqrt{M}$		
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;
```

Integer Data Types

PROC NETWORK processes variables that are integer data types in two different ways, depending on whether the variable is a node/link attribute or a label (that is, data in the to, from, queryKey, or node column). The values of node and link attribute variables are converted to doubles, and computations are performed in double precision. Attribute values are also output as double types so that the results of the algorithms are accurately reported. For example, node and link weights, as well as variables that you specify in the VARS= option of the LINKSVAR or NODESVAR statement, are converted to and output as doubles. In contrast, label variables remain as integers when you input them as integer data types. Note that integer values less than -2^{53} or greater than 2^{53} cannot always be converted to doubles without a loss of precision. PROC NETWORK issues a warning when it detects a possible loss of precision due to attribute values being converted from integer data types to doubles.

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, the procedure issues an error message.

Negative Link Weights

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

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

Size Limitations

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

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

Exceptions to this limitation are: the distributed union-find algorithm for finding **connected components**, the parallel label propagation algorithm for **community detection**, and distributed **degree centrality**. 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.

[†] One exception to the limit on the number of supported links is for calculating the eigenvector centrality metric for undirected graphs. In this case, the limit on the number of links is half the standard limit (1,073,741,823 links).

Common Notation and Assumptions

This section briefly introduces some common notation and assumptions that are used throughout the chapter. Let $G = (N, E)$ define a graph (or multigraph) that contains a set N of nodes and a set E of links. Table 2.8 provides a list of commonly used notation.

Table 2.8 Common Notation

Notation	Description
δ_i	The set of links that are incident to node i
δ_i^{out}	The set of outgoing links that are connected from node i
δ_i^{in}	The set of incoming links that are connected to node i
$\text{from}(e)$	The <i>from</i> node of link e
$\text{to}(e)$	The <i>to</i> node of link e
N_i	The set of neighbors (that is, unique nodes that are connected by links incident with node i)
$\bar{N}_i$	The multiset of neighbors (that is, nodes that are connected by links incident with node i)
$\hat{N}_i$	The set of neighbors, excluding node i itself (that is, unique nodes that are connected by links, excluding self-links, incident with node i)
N_i^{out}	The set of out-neighbors (that is, unique nodes that are connected by outgoing links from node i)
$\bar{N}_i^{\text{out}}$	The multiset of out-neighbors (that is, nodes that are connected by outgoing links from node i)
$\hat{N}_i^{\text{out}}$	The set of out-neighbors, excluding node i itself (that is, unique nodes that are connected by outgoing links, excluding self-links, from node i)
N_i^{in}	The set of in-neighbors (that is, unique nodes that are connected by incoming links to node i)
$\hat{N}_i^{\text{in}}$	The set of in-neighbors, excluding node i itself (that is, unique nodes that are connected by incoming links, excluding self-links, to node i)

A *complete graph* on node set N is a simple graph (with no self-links) in which every pair of nodes in N is connected by a link. The number of links in a complete graph on node set N is

$$K(N) = \frac{|N|^2 - |N|}{2}$$

when DIRECTION=UNDIRECTED, or

$$K(N) = |N|^2 - |N|$$

when DIRECTION=DIRECTED.

An *induced subgraph* of a graph $G = (N, E)$ is a subgraph that is defined by a subset of nodes $S \subset N$ and the links of G whose endpoints are both in the subset S .

Examples Statement and Option Cross-Reference

Algorithm Classes

This section provides a cross-reference of which examples use which algorithm classes.

Biconnected components

- “Example 2.1: Articulation Points in a Terrorist Network” on page 268

Centrality

- “Example 2.2: Influence Centrality for Project Groups in a Research Department” on page 271
- “Example 2.3: Betweenness and Closeness Centrality for Computer Network Topology” on page 275
- “Example 2.4: Betweenness and Closeness Centrality for Project Groups in a Research Department” on page 277
- “Example 2.5: Eigenvector Centrality for Word Sense Disambiguation” on page 281
- “Example 2.8: Centrality Metrics for an Undirected Graph by Community” on page 290

Community detection

- “Example 2.6: Community Detection on Zachary’s Karate Club Data” on page 283
- “Example 2.7: Recursive Community Detection on Zachary’s Karate Club Data” on page 289
- “Example 2.8: Centrality Metrics for an Undirected Graph by Community” on page 290
- “Example 2.19: Node Embeddings for Zachary’s Karate Club Data” on page 344

Connected components

- “Example 2.12: Connected Components for US Patent Citations” on page 305

Cycle enumeration

- “Example 2.9: Cycle Enumeration for Kidney Donor Exchange” on page 295
- “Example 2.10: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System” on page 300

Network projection

- “Example 2.22: Using Projection to Recommend Recipe Ingredient Pairings” on page 359

Node similarity

- “Example 2.17: Node Similarity for Link Prediction” on page 335
- “Example 2.18: A Node Similarity Recipe Recommendation Engine” on page 341
- “Example 2.19: Node Embeddings for Zachary’s Karate Club Data” on page 344
- “Example 2.20: Node Embeddings for European Roads Data” on page 349
- “Example 2.21: Learning Word Embeddings from Synonym Pairs” on page 355

Pattern matching

- “Example 2.15: Pattern Matching in a Social Network” on page 312
- “Example 2.16: Detection of Value-Added Tax Carousel Fraud” on page 321

Reach (ego) network

- “Example 2.11: Reach Networks for Computing the Market Coverage of a Terrorist Network” on page 303

Shortest path

- “Example 2.13: Shortest Paths of the New York Road Network” on page 307
- “Example 2.14: Shortest Path in a Road Network by Date and Time” on page 309

Transitive closure

- “Example 2.10: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System” on page 300

Statements and Options

Table 2.9 and Table 2.10 show which examples use each of the statements and options in PROC NETWORK.

Table 2.9 Statements and Options in Examples 1–14

Statements and Options	1	2	3	4	5	6	7	8	9	10	11	12	13	14
<i>PROC Statement</i>														
PROC NETWORK														
DIRECTION=									X	X			X	
LINKS=	X	X	X	X	X	X	X	X	X	X	X	X	X	X
LOGFREQTIME=														
LOGLEVEL=		X		X					X	X			X	
NODES=		X												
OUTLINKS=			X	X				X						
OUTNODES=	X	X	X	X	X	X	X	X			X	X		

Table 2.9 continued

Statements and Options	1	2	3	4	5	6	7	8	9	10	11	12	13	14
<i>Variables Statements</i>														
LINKSVAR										X				X
<i>Algorithm Statements</i>														
CONNECTEDCOMPONENTS OUT=												X X		
BICONNECTEDCOMPONENTS	X		X											
CENTRALITY BETWEEN= CLOSE= DEGREE= EIGEN= INFLUENCE=		X X X	X X X X	X X X	X X			X X X X X						
COMMUNITY OUTCOMMLINKS= OUTCOMMUNITY= OUTLEVEL= OUTOVERLAP= RECURSIVE RESOLUTIONLIST=						X X X X X	X X	X X						
CYCLE MAXCYCLES= MAXLENGTH= MINLENGTH= OUTCYCLESLINKS= OUTCYCLESNODES=									X X X X X	X X				
REACH EACHSOURCE MAXREACH= OUTREACHNODES=											X X X X			
SHORTESTPATH MAXLINKWEIGHT= OUTPATHSLINKS= OUTWEIGHTS= SINK= SOURCE=													X X X X	X X X X X
TRANSITIVECLOSURE OUT=										X X				
<i>Standard Statements</i>														
BY								X						X
DISPLAYOUT								X						X

Table 2.10 Statements and Options in Examples 15–22

Statements and Options	15	16	17	18	19	20	21	22
<i>PROC Statement</i>								
PROC NETWORK								
DIRECTION=	X	X						
LINKS=	X	X	X	X	X	X	X	X
LINKSQUERY=	X	X						
NODES=	X	X			X	X		X
NODESQUERY=	X	X						
NODESSUBSET=			X	X				
OUTNODES=					X	X	X	
<i>Variables Statements</i>								
LINKSQUERYVAR	X	X						
LINKSVAR	X	X						
NODESQUERYVAR	X							
NODESVAR	X	X			X	X		
<i>Algorithm Statements</i>								
COMMUNITY					X			
NODESIMILARITY			X	X	X	X	X	
COSINE=				X				
EMBED=					X	X		
JACCARD=				X	X	X		
NDIMENSIONS=						X	X	
NSAMPLES=					X	X	X	
OUTSIMILARITY=			X	X			X	
PROXIMITYORDER=					X	X	X	
VECTOR=							X	
PATTERNMATCH	X	X						
CODE=	X	X						
LINKFILTER=	X							
LINKPAIRFILTER=		X						
MATCHFILTER=		X						
NODEFILTER=		X						
NODEPAIRFILTER=		X						
OUTMATCHLINKS=	X	X						
OUTMATCHNODES=	X	X						
OUTQUERYLINKS=		X						
OUTQUERYNODES=		X						
OUTSUMMARY=		X						
QUERYKEY=		X						
PROJECTION								X
COMMONNEIGHBORS=								X
OUTNEIGHBORSLIST=								X
OUTPROJECTIONLINKS=								X
PARTITION=								X

Biconnected Components and Articulation Points

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

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

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

The *block-cut tree* of a given connected graph $G = (N, E)$ is an abstract representation of that graph in the form of a tree, where biconnected components, or blocks, are connected through articulation points, or cut points.

The nodes and links information of the block-cut tree is written to two data tables that you specify in the OUTBCTREENODES= and OUTBCTREELINKS= options in the BICONNECTEDCOMPONENTS statement. Additionally, a map from the nodes in the original graph to the nodes in the block-cut tree is written to the bcTree_node column of the OUTNODES= data table.

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

Output Data Tables

Depending on the specified options, the biconnected components algorithm produces an additional output data table as described in the following section.

OUT= Data Table

The OUT= data table describes the number of links in each biconnected component. This data table contains the following columns:

- biconcomp: the biconnected component identifier
- links: the number of links that are contained in the biconnected component

OUTBCTREELINKS= Data Table

The OUTBCTREELINKS= data table describes the links in the block-cut tree. This data table contains the following columns:

- from: the *from* node identifier, prefixed with a_ for articulation points and b_ for biconnected components
- to: the *to* node identifier, prefixed with a_ for articulation points and b_ for biconnected components

OUTBCTREENODES= Data Table

The OUTBCTREENODES= data table describes the nodes in the block-cut tree. This data table contains the following columns:

- node: the node identifier, prefixed with a_ for articulation points and b_ for biconnected components. The articulation points and biconnected components are numbered sequentially, starting from the value that you specify in the INDEXOFFSET= option in the PROC NETWORK statement.
- artpoint: the label of the articulation node in the original graph if the node is an articulation point; otherwise the value is missing.

OUTNODES= Data Table

The results of the biconnected components are added to the output data table that you specify in the OUTNODES= option in the PROC NETWORK statement. Each node in the original graph is listed in the output data table along with the artpoint variable, which indicates whether that node is an articulation point. This data table contains (at least) the following columns:

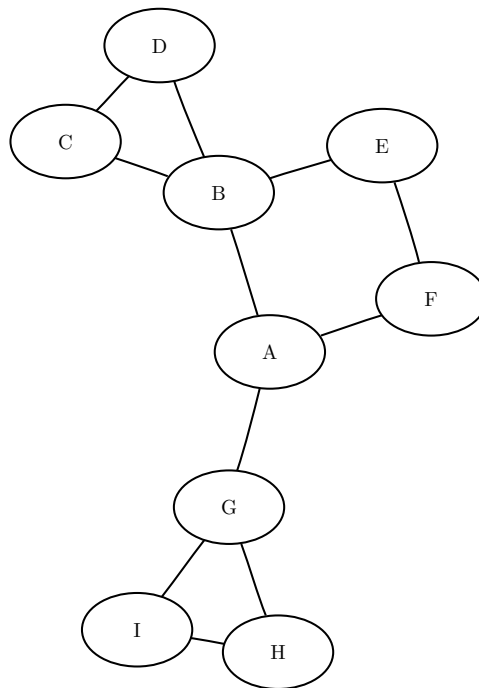
- node: the node label
- artpoint: the articulation point indicator, which is 1 if the node is an articulation point and 0 otherwise

If you specify the OUTBCTREELINKS= or OUTBCTREENODES= option, the output data table also contains the bcTree_node variable, which maps the node to its counterpart in the block-cut tree. In this case, the data table also contains the following column:

- bcTree_node: the node identifier in the block-cut tree

Biconnected Components of an Undirected Graph

This section illustrates the use of the biconnected components algorithm on the undirected graph G that is shown in Figure 2.39.

Figure 2.39 Undirected Graph G 

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

```

data mylib.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 mylib.LinkSetOut, mylib.NodeSetOut, and mylib.BiConCompOut:

```

proc network
  links      = mylib.LinkSetInBiCC
  outLinks   = mylib.LinkSetOut
  outNodes   = mylib.NodeSetOut;
  biconnectedComponents
    out      = mylib.BiConCompOut;
run;

```

The output data table mylib.LinkSetOut contains the biconnected components of the input graph, as shown in Figure 2.40.

Figure 2.40 Biconnected Components of an Undirected Graph

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

The output data table mylib.NodeSetOut contains the articulation points of the input graph, as shown in Figure 2.41.

Figure 2.41 Articulation Points of an Undirected Graph

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

The output data table mylib.BiConCompOut contains the number of links in each biconnected component of the input graph, as shown in Figure 2.42.

Figure 2.42 Summary for the Biconnected Components of an Undirected Graph

biconcomp	links
1	4
2	1
3	3
4	3

The biconnected components are shown graphically in Figure 2.43 and Figure 2.44.

Figure 2.43 Biconnected Components C^1 and C^2

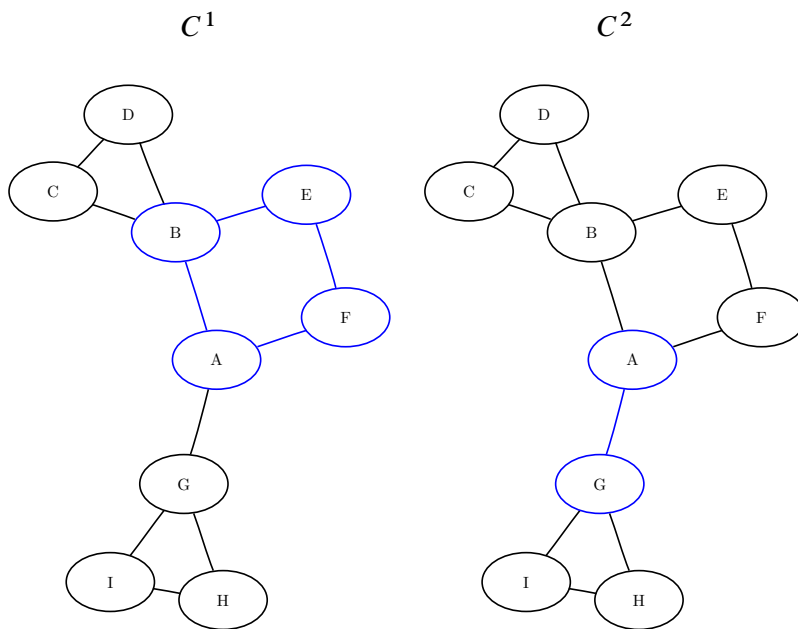
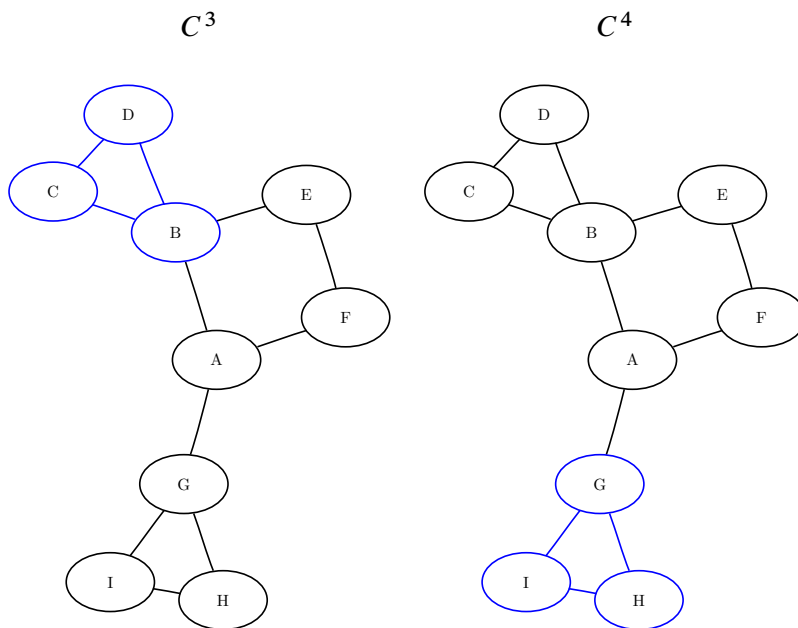


Figure 2.44 Biconnected Components C^3 and C^4



For a more detailed example, see “Example 2.1: Articulation Points in a Terrorist Network” on page 268.

Block-Cut Tree of an Undirected Graph

Consider again the undirected graph in [Figure 2.39](#), which is represented by the links data table `mylib.LinkSetInBiCC`. The following statements calculate the block-cut tree and output the results in the data tables `mylib.BCTreeNodesOut` and `mylib.BCTreeLinksOut`. To indicate the mapping to the original node labels, the `bcTree_node` variable is added to the `mylib.NodeSetOut` data table.

```

proc network
  links      = mylib.LinkSetInBiCC
  outLinks   = mylib.LinkSetOut
  outNodes   = mylib.NodeSetOut;
  biconnectedComponents
    outBCTreeNodes = mylib.BCTreeNodesOut
    outBCTreeLinks = mylib.BCTreeLinksOut;
run;

```

The `mylib.BCTreeNodesOut` output data table contains the nodes in the block-cut tree, as shown in [Figure 2.45](#).

Figure 2.45 Block-Cut Tree Nodes Data Table

node	artpoint
a_1	A
a_2	B
a_3	G
b_1	
b_2	
b_3	
b_4	

The `mylib.BCTreeLinksOut` output data table contains the links in the block-cut tree, as shown in [Figure 2.46](#).

Figure 2.46 Block-Cut Tree Links Data Table

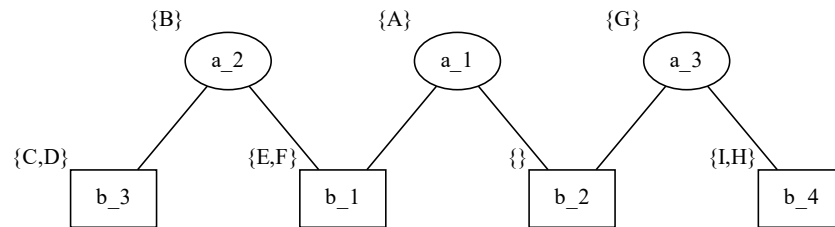
from	to
a_1	b_1
a_1	b_2
a_2	b_1
a_2	b_3
a_3	b_2
a_3	b_4

The `mylib.NodeSetOut` output data table contains the articulation points of the input graph as well as the `bcTree_node` variable, which indicates the map from nodes in the original graph to nodes in the block-cut tree, as shown in [Figure 2.47](#).

Figure 2.47 Articulation Points and Block-Cut Tree Node Mappings

node	artpoint	bcTree_node
A	1	a_1
B	1	a_2
C	0	b_3
D	0	b_3
E	0	b_1
F	0	b_1
G	1	a_3
H	0	b_4
I	0	b_4

The block-cut tree is shown graphically in [Figure 2.48](#).

Figure 2.48 Block-Cut Tree of the Undirected Graph G 

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 30.

Let $G = (N, E)$ define a graph that contains nodes N and links E . The following sections describe each of the possible centrality metrics that you can calculate in PROC NETWORK.

Degree Centrality

The (unweighted) *degree* (*valency*) of a node j in an undirected graph is the number of links that are incident to node j , with self-links counted twice. The *out-degree* of a node in a directed graph is the number of out-links incident to that node; the *in-degree* is the number of in-links incident to that node. For an undirected graph, the terms *degree* and *out-degree* are interchangeable. *Degree centrality* is simply the in- or out-degree of a node and can be interpreted as some form of the node’s relative importance to a network. For example, in a network where nodes are people and you are tracking the flow of a virus, the degree centrality gives

some idea of the magnitude of the risk of spreading the virus. People who have a higher out-degree can lead to faster and more widespread transmission. In a friendship network, in-degree often indicates popularity.

The weighted degree of a node is a generalization of degree; it is the sum of the link weights incident to that node. For unweighted graphs, the default weight of each link is 1.

For undirected graphs, degree centrality is

$$C_d(i) = \sum_{e \in \delta_i} w_e^E$$

where δ_i represents the set of links that are incident to node i , and w_e^E is the weight of link e .

For directed graphs, out-degree centrality of node i is

$$C_d^{\text{out}}(i) = \sum_{e \in \delta_i^{\text{out}}} w_e^E$$

where δ_i^{out} represents the set of outgoing links that are connected from node i .

For directed graphs, in-degree centrality of node i is

$$C_d^{\text{in}}(i) = \sum_{e \in \delta_i^{\text{in}}} w_e^E$$

where δ_i^{in} represents the set of incoming links that are connected to node i .

The overall degree centrality for a directed graph is calculated as

$$C_d(i) = C_d^{\text{out}}(i) + C_d^{\text{in}}(i)$$

Degree centrality is calculated using the value of the **DEGREE=** option in the **CENTRALITY** statement. To specify a distributed graph algorithm, you can set **DISTRIBUTED=TRUE** in the **PROC NETWORK** statement.

The algorithm that **PROC NETWORK** uses to compute degree centrality runs in time $O(|N|)$, and therefore should scale to very large graphs.

OUTNODES= Data Table

The results are provided in the nodes output data table that you specify in the **OUTNODES=** option in the **PROC NETWORK** statement.

The **OUTNODES=** data table contains the degree centrality of each node. This data table contains the following column:

- **node:** the node label

If the graph is undirected and you specify **DEGREE=UNWEIGHT** or **DEGREE=BOTH**, the following column also appears in the nodes output data table:

- `centr_degree_out`: the unweighted degree centrality of the node

If the graph is undirected and you specify `DEGREE=WEIGHT` or `DEGREE=BOTH`, the following column also appears in the nodes output data table:

- `centr_degree_out_wt`: the weighted degree centrality of the node

If the graph is directed and you specify `DEGREE=UNWEIGHT` or `DEGREE=BOTH`, the following columns also appear in the nodes output data table:

- `centr_degree_in`: the unweighted in-degree centrality of the node
- `centr_degree_out`: the unweighted out-degree centrality of the node
- `centr_degree`: the sum of the unweighted in- and out-degree centrality of the node

If the graph is directed and you specify `DEGREE=WEIGHT` or `DEGREE=BOTH`, the following columns also appear in the nodes output data table:

- `centr_degree_in_wt`: the weighted in-degree centrality of the node
- `centr_degree_out_wt`: the weighted out-degree centrality of the node
- `centr_degree_wt`: the sum of the weighted in- and out-degree centrality of the node

Degree Centrality of a Directed Graph

As a simple example, consider again the directed graph in [Figure 2.7](#) with the data table `mylib.LinkSetIn`, which is defined in the section “[Links Input Data](#)” on page 62. The following statements calculate the weighted degree centrality for both in- and out-degrees:

```
proc network
  direction = directed
  links      = mylib.LinkSetIn
  outNodes   = mylib.NodeSetOut;
  centrality
    degree   = weight;
run;
```

The nodes data table `mylib.NodeSetOut` now contains the degree centrality for each node of the input graph. This data table is shown in [Figure 2.49](#).

Figure 2.49 Degree Centrality of a Directed Graph

node	centr_degree_in_wt	centr_degree_out_wt	centr_degree_wt
F	7	6	13
G	8	2	10
B	1	8	9
A	0	7	7
E	4	3	7
D	5	1	6
H	1	5	6
C	3	1	4
I	4	0	4

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*.

For undirected graphs, influence centrality is

$$C_1(i) = \frac{\sum_{e \in \delta_i} w_e^E}{\sum_{j \in N} w_j^N}$$

$$C_2(i) = \sum_{j \in \bar{N}_i} C_1(j)$$

where δ_i represents the set of links that are incident to node i , $\bar{N}_i$ represents the multiset of neighbors (that is, nodes that are connected by links incident with node i), w_e^E is the weight of link e , and w_i^N is the weight of node i .

For directed graphs, influence centrality is

$$C_1(i) = \frac{\sum_{e \in \delta_i^{\text{out}}} w_e^E}{\sum_{j \in N} w_j^N}$$

$$C_2(i) = \sum_{j \in \bar{N}_i^{\text{out}}} C_1(j)$$

where δ_i^{out} represents the set of outgoing links that are connected from node i and $\bar{N}_i^{\text{out}}$ represents the multiset of out-neighbors (that is, nodes that are connected by outgoing links from node i).

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 algorithm that PROC NETWORK uses to compute influence centrality is a simple traversal, runs in time $O(|E|)$, and therefore should scale to very large graphs.

OUTNODES= Data Table

The results are provided in the nodes output data table that you specify in the OUTNODES= option in the PROC NETWORK statement.

The OUTNODES= data table contains the influence centrality of each node. This data table contains the following column:

- node: the node label

If you specify INFLUENCE=UNWEIGHT or INFLUENCE=BOTH, the following columns also appear in the nodes output data table:

- centr_influence1_unwt: the unweighted first-order influence centrality of the node
- centr_influence2_unwt: the unweighted second-order influence centrality of the node

If you specify INFLUENCE=WEIGHT or INFLUENCE=BOTH, the following columns also appear in the nodes output data table:

- centr_influence1_wt: the weighted first-order influence centrality of the node
- centr_influence2_wt: the weighted second-order influence centrality of the node

Influence Centrality of a Directed Graph

Consider again the directed graph in [Figure 2.7](#). Ignore the weights and just calculate the C_1 and C_2 metrics that are 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        = mylib.LinkSetIn
  outNodes     = mylib.NodeSetOut;
  centrality
    influence = unweight;
run;
```

The nodes data table mylib.NodeSetOut now contains the unweighted influence centrality for each node of the input graph. This data table is shown in [Figure 2.50](#).

Figure 2.50 Influence Centrality of a Directed Graph

node	centr_influence1_unwt	centr_influence2_unwt
A	0.33333	0.55556
B	0.33333	0.44444
E	0.22222	0.22222
G	0.22222	0.22222
H	0.22222	0.22222
C	0.11111	0.22222
D	0.11111	0.22222
F	0.11111	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 271.

Clustering Coefficient

The local *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). The local clustering coefficient metric is often referred to as *transitivity*.

The formula for the clustering coefficient is

$$C(i) = \frac{\sum_{\substack{j \in \hat{N}_i, k \in \hat{N}_i: \\ j < k}} b_{jk}}{K(\hat{N}_i)}$$

when DIRECTION=UNDIRECTED, or

$$C(i) = \frac{\sum_{\substack{j \in \hat{N}_i^{\text{out}}, k \in \hat{N}_i^{\text{out}}: \\ j \neq k}} b_{jk}}{K(\hat{N}_i^{\text{out}})}$$

when DIRECTION=DIRECTED, where $\hat{N}_i$ represents the set of neighbors, excluding node i itself (that is, unique nodes that are connected by links, excluding self-links, incident with node i); $\hat{N}_i^{\text{out}}$ represents the set of out-neighbors, excluding node i itself (that is, unique nodes that are connected by outgoing links, excluding self-links, from node i); and b_{jk} specifies whether or not (1 or 0, respectively) a link from node j to node k exists in the link set E , as follows:

$$b_{jk} = \begin{cases} 1, & \text{if there exists } e \in E \text{ such that } \text{from}(e) = j \text{ and } \text{to}(e) = k \\ 0, & \text{otherwise} \end{cases}$$

The definition of the local clustering coefficient for undirected graphs is closely related to the global triangle count, which is given by

$$C_T = \frac{1}{3} \sum_{i \in N} K(\hat{N}_i) C(i) = \sum_{\substack{i \in N, j \in \hat{N}_i, k \in \hat{N}_i: \\ i < j < k}} b_{jk}$$

When `DIRECTION=UNDIRECTED`, this count is reported in the `NUM_TRIANGLES` term of the macro variable `_NETWORK_`.

In `PROC NETWORK`, the calculation of local clustering coefficient considers only binary networks. Weights and multilinks are ignored in the metric.

For a particular node, the clustering coefficient determines how close the subgraph that is induced by its neighbor set is to being a complete subgraph. In social networks, a high clustering coefficient can help predict relationships that might not be known, confirmed, or realized yet. Consider a social network of four people, where person *A* knows persons *B*, *C*, and *D*, person *B* knows person *C*, and person *C* knows person *D*. This does not guarantee that person *B* knows person *D*, but it is more likely that person *B* knows person *D* than that person *B* knows some random person.

The clustering coefficient is calculated when you specify the `CLUSTERINGCOEFFICIENT` option in the `CENTRALITY` 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.

OUTNODES= Data Table

The results are provided in the nodes output data table that you specify in the `OUTNODES=` option in the `PROC NETWORK` statement.

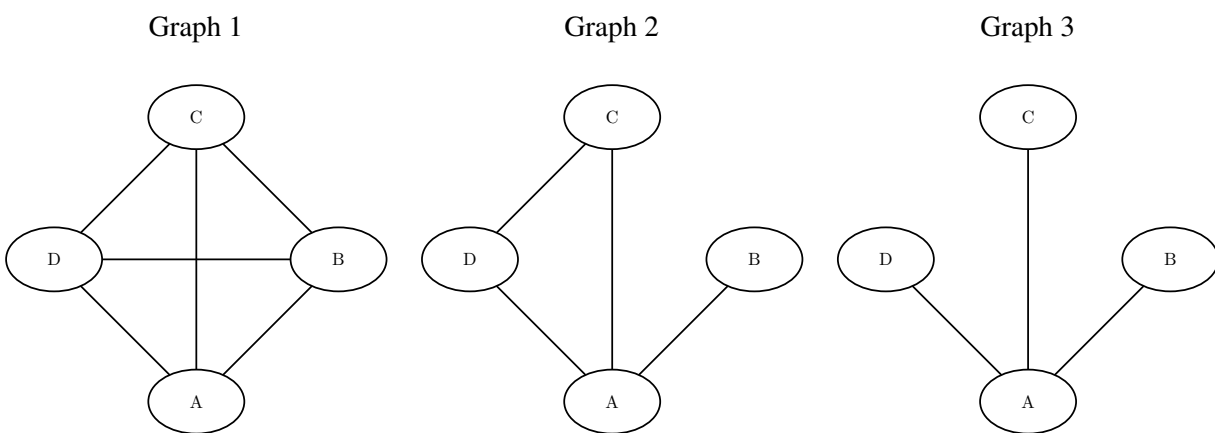
The `OUTNODES=` data table contains the clustering coefficient of each node. This data table contains the following columns:

- `node`: the node label
- `centr_cluster`: the clustering coefficient of the node

Clustering Coefficient of an Undirected Graph

Consider the three undirected graphs on four nodes shown in Figure 2.51.

Figure 2.51 Three Undirected Graphs



Define the three links data tables as follows:

```

data mylib.LinkSetInCC1;
    input from $ to $ @@;
    datalines;
A B  A C  A D
B C  B D  C D
;

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

data mylib.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      = mylib.LinkSetInCC1
    outNodes   = mylib.NodeSetOut1;
    centrality
        clusteringCoef;
run;

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

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

```

The nodes data tables provide the clustering coefficients for each node of each input graph, as shown in Figure 2.52 through Figure 2.54.

Figure 2.52 Clustering Coefficient of an Undirected Graph 1

node	centr_cluster
A	1
B	1
C	1
D	1

Figure 2.53 Clustering Coefficient of an Undirected Graph 2

node	centr_cluster
C	1.00000
D	1.00000
A	0.33333
B	0.00000

Figure 2.54 Clustering Coefficient of an 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_{ij} to be the shortest path distance from node i to node j , 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, $R(i) = \{j \in N : d_{ij} < \infty\}$ is the set of *reachable nodes* from node i . The set of *unreachable nodes* from node i is $N \setminus R(i) = \{j \in N : d_{ij} = \infty\}$. The **CLOSENOPATH=** option specifies how to handle unreachable nodes.

For the special case in which all nodes are unreachable from node i , the closeness centrality is defined as 0. Otherwise, closeness centrality is calculated as

$$C_c(i) = s(i) \left(\frac{n(i)}{\sum_{j \in R(i)} d_{ij} + |N \setminus R(i)| p} \right)$$

where p defines a penalty parameter for unreachable nodes, $n(i)$ defines the number of nodes that are considered in calculating the average, and $s(i)$ is a scaling factor, as shown in Table 2.11.

Table 2.11 Formulas for CLOSENOPATH= Option for Undirected Graphs

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

Closeness Centrality for a Directed Graph

For a directed graph, $R^{\text{out}}(i) = \{j \in N : d_{ij} < \infty\}$ is the set of *reachable nodes* from node i , whereas $R^{\text{in}}(i) = \{j \in N : d_{ji} < \infty\}$ is the set of nodes from which there is a finite path to node i . The set of *unreachable nodes* from node i is $N \setminus R^{\text{out}}(i) = \{j \in N : d_{ij} = \infty\}$, whereas the set of nodes from which there is no finite path to node i is $N \setminus R^{\text{in}}(i) = \{j \in N : d_{ji} = \infty\}$.

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

$$C_c^{\text{out}}(i) = s^{\text{out}}(i) \left(\frac{n^{\text{out}}(i)}{\sum_{j \in R^{\text{out}}(i)} d_{ij} + |N \setminus R^{\text{out}}(i)| p} \right)$$

where $n^{\text{out}}(i)$ defines the number of nodes that are considered in calculating the average and $s^{\text{out}}(i)$ is a scaling factor, as shown in Table 2.12.

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

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

where $n^{\text{in}}(i)$ defines the number of nodes that are considered in calculating the average and $s^{\text{in}}(i)$ is a scaling factor, as shown in Table 2.12.

Table 2.12 Formulas for CLOSENOPATH= Option for Directed Graphs

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

The overall closeness centrality for directed graphs is calculated as

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

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

$$C_h(i) = \frac{1}{|N| - 1} \sum_{j \in N \setminus \{i\}} \frac{1}{d_{ij}}$$

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. 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 $O(|N| \times (|N| \log |N| + |E|))$. 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).

OUTNODES= Data Table

The results are provided in the nodes output data table that you specify in the OUTNODES= option in the PROC NETWORK statement.

The OUTNODES= data table contains the closeness centrality of each node. This data table contains the following column:

- node: the node label

If the graph is undirected and you specify CLOSE=UNWEIGHT or CLOSE=BOTH, the following column also appears in the nodes output data table:

- centr_close_unwt: the unweighted closeness centrality of the node

If the graph is undirected and you specify CLOSE=WEIGHT or CLOSE=BOTH, the following column also appears in the nodes output data table:

- centr_close_wt: the weighted closeness centrality of the node

If the graph is directed and you specify CLOSE=UNWEIGHT or CLOSE=BOTH, the following columns also appear in the nodes output data table:

- centr_close_in_unwt: the unweighted in-closeness centrality of the node
- centr_close_out_unwt: the unweighted out-closeness centrality of the node
- centr_close_unwt: the unweighted closeness centrality of the node

If the graph is directed and you specify CLOSE=WEIGHT or CLOSE=BOTH, the following columns also appear in the nodes output data table:

- centr_close_in_wt: the weighted in-closeness centrality of the node
- centr_close_out_wt: the weighted out-closeness centrality of the node
- centr_close_wt: the weighted closeness centrality of the node

Closeness Centrality of a Directed Graph

Consider again the directed graph in Figure 2.7 with the data table mylib.LinkSetIn, which is defined in the section “Links Input Data” on page 62. The following statements calculate the closeness centrality for both the weighted and unweighted graphs:

```
proc network
  direction = directed
  links      = mylib.LinkSetIn
  outNodes   = mylib.NodeSetOut;
  centrality
    close    = both;
run;
```

The nodes data table mylib.NodeSetOut now contains the weighted and unweighted directed closeness centrality of the input graph. This data table is shown in Figure 2.55.

Figure 2.55 Closeness Centrality of a 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
G	0.56402	0.79470	0.33333	0.32500	0.40000	0.25000
B	0.55134	0.30000	0.80268	0.33333	0.22222	0.44444
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
H	0.41663	0.48290	0.35036	0.27885	0.30769	0.25000
D	0.40113	0.41202	0.39024	0.29178	0.30769	0.27586
A	0.38835	0.00000	0.77670	0.22222	0.00000	0.44444
C	0.38260	0.34043	0.42478	0.27885	0.25000	0.30769
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

$$C_b(i) = \sum_{\substack{(s,t) \in N \times N: \\ s < t, \sigma_{st} > 0}} \frac{\sigma_{st}(i)}{\sigma_{st}}$$

when DIRECTION=UNDIRECTED, or

$$C_b(i) = \sum_{\substack{(s,t) \in N \times N: \\ s \neq t, \sigma_{st} > 0}} \frac{\sigma_{st}(i)}{\sigma_{st}}$$

when DIRECTION=DIRECTED, where σ_{st} is the number of shortest paths from s to t and $\sigma_{st}(i)$ is the number of shortest paths from s to t that pass through node i . As with closeness centrality, the shortest path is calculated with respect to the link weight that is 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 that is 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.

By default, this metric is normalized by dividing by $K(N \setminus \{i\})$, the number of links in the complete graph on node set $N \setminus \{i\}$. You can disable this normalization by using the **BETWEENNORM=** option.

The formula for link betweenness centrality is

$$C_b(i, j) = \sum_{\substack{(s,t) \in N \times N: \\ s < t, \sigma_{st} > 0}} \frac{\sigma_{st}(i, j)}{\sigma_{st}}$$

when DIRECTION=UNDIRECTED, or

$$C_b(i, j) = \sum_{\substack{(s,t) \in N \times N: \\ s \neq t, \sigma_{st} > 0}} \frac{\sigma_{st}(i, j)}{\sigma_{st}}$$

when DIRECTION=DIRECTED, where $\sigma_{st}(i, j)$ is the number of shortest paths from s to t that pass through link (i, j) .

By default, this metric is normalized by dividing by $K(N)$, the number of links in the complete graph on node set N . 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. As with closeness centrality, if **BETWEEN=WEIGHT** (or **BETWEEN=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 $O(|N| \times (|N| \log |N| + |E|))$. 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.

You can compute an approximation of betweenness centrality by using the **SAMPLEPERCENT=** option in the CENTRALITY statement (AlGhamdi et al. 2017). The value that you specify for this option (p) defines the percentage of source nodes to use in calculating shortest paths. In this case the algorithm runs in time $O(\frac{p|N|}{100} \times (|N| \log |N| + |E|))$. If $p = 100$, then PROC NETWORK computes betweenness centrality exactly.

OUTNODES= Data Table

The node betweenness centrality results are provided in the nodes output data table that you specify in the OUTNODES= option in the PROC NETWORK statement.

The OUTNODES= data table contains the betweenness centrality of each node. This data table contains the following column:

- node: the node label

If you specify BETWEEN=UNWEIGHT or BETWEEN=BOTH, the following column also appears in the nodes output data table:

- centr_between_unwt: the unweighted betweenness centrality of the node

If you specify BETWEEN=WEIGHT or BETWEEN=BOTH, the following column also appears in the nodes output data table:

- centr_between_wt: the weighted betweenness centrality of the node

OUTLINKS= Data Table

The link betweenness centrality results are provided in the links output data table that you specify in the OUTLINKS= option in the PROC NETWORK statement.

The OUTLINKS= data table contains the betweenness centrality of each link. This data table contains the following columns:

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

If you specify BETWEEN=UNWEIGHT or BETWEEN=BOTH, the following column also appears in the links output data table:

- centr_between_unwt: the unweighted betweenness centrality of the link

If you specify BETWEEN=WEIGHT or BETWEEN=BOTH, the following column also appears in the links data table:

- centr_between_wt: the weighted betweenness centrality of the link

Betweenness Centrality of a Directed Graph

Consider again the directed graph in [Figure 2.7](#) with data table mylib.LinkSetIn, which is defined in the section “[Links Input Data](#)” on page 62. The following statements calculate the betweenness centrality for both the weighted and unweighted graphs:

```

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

```

The nodes data table mylib.NodeSetOut now contains the weighted and unweighted node betweenness centrality of the input graph. This data table is shown in [Figure 2.56](#).

Figure 2.56 Node Betweenness Centrality of a Directed Graph

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

In addition, the links data table mylib.LinkSetOut contains the weighted and unweighted link betweenness centrality of the input graph. This data table is shown in [Figure 2.57](#).

Figure 2.57 Link Betweenness Centrality of a Directed Graph

from	to	weight	centr_between_wt	centr_between_unwt
F	G	6	0.25000	0.25000
E	F	2	0.16667	0.16667
B	F	5	0.11111	0.11111
G	H	1	0.09722	0.09722
G	I	1	0.09722	0.09722
C	E	1	0.08333	0.08796
D	E	1	0.08333	0.07407
A	B	1	0.06944	0.07407
E	D	1	0.04167	0.04167
A	D	4	0.02778	0.01852
B	E	2	0.02778	0.03241
A	C	2	0.01389	0.01852
B	C	1	0.01389	0.01389
H	G	2	0.01389	0.01389
H	I	3	0.01389	0.01389

For more detailed examples, see “[Example 2.3: Betweenness and Closeness Centrality for Computer Network Topology](#)” on page 275 and “[Example 2.4: Betweenness and Closeness Centrality for Project Groups in a Research Department](#)” on page 277.

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

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

when DIRECTION=UNDIRECTED, or

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

when DIRECTION=DIRECTED, where x_i is the eigenvector centrality of node i , w_{ij} is the sum of the weights of links from node i to node j , and λ is a constant.

The *right* eigenvector equation in matrix form can be written as:

$$\mathbf{A}\mathbf{x} = \lambda\mathbf{x}$$

where, for an undirected graph, $\mathbf{A}$ represents the (symmetric) adjacency matrix that corresponds to the link weights in E , or for a directed graph, $\mathbf{A}$, represents the adjacency matrix that corresponds to the outgoing link weights in E . To calculate the *left eigenvector*, which corresponds to the incoming link weights, switch the orientation of the *from* and *to* variables in the data table by using the LINKSVAR= statement. In constructing the matrix $\mathbf{A}$, the weights of multilinks are aggregated. Eigenvector centrality is defined as the principal eigenvector $\mathbf{x}$ that corresponds to the largest eigenvalue λ .

To solve this eigenvector problem, PROC NETWORK provides two algorithms: the Jacobi-Davidson algorithm and the power method. You can use the EIGENALGORITHM= option in the CENTRALITY statement to specify which algorithm to use. EIGENALGORITHM=JACOBIDAVIDSON, which is the default, specifies the Jacobi-Davidson algorithm 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.

Eigenvector centrality is calculated using the value that you specify in the EIGEN= option in the CENTRALITY statement.

OUTNODES= Data Table

The results are provided in the nodes output data table that you specify in the OUTNODES= option in the PROC NETWORK statement.

The OUTNODES= data table contains the eigenvector centrality of each node. This data table contains the following column:

- **node:** the node label

If you specify EIGEN=UNWEIGHT or EIGEN=BOTH, the following column also appears in the nodes output data table:

- `centr_eigen_unwt`: the unweighted eigenvector centrality of the node

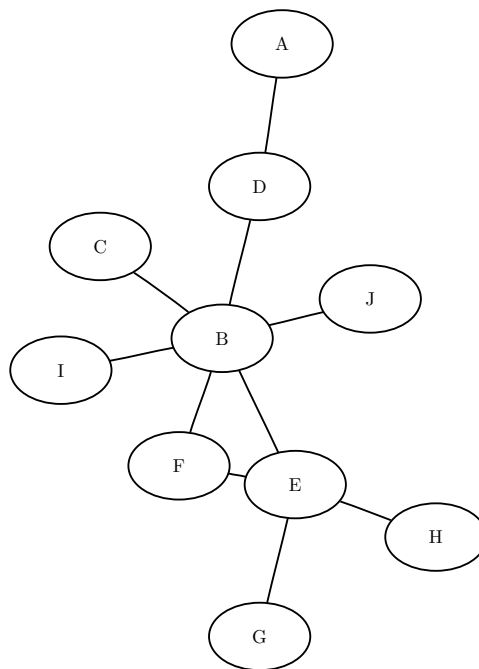
If you specify `EIGEN=WEIGHT` or `EIGEN=BOTH`, the following column also appears in the nodes output data table:

- `centr_eigen_wt`: the weighted eigenvector centrality of the node

Eigenvector Centrality of an Undirected Graph

The following example illustrates the use of eigenvector centrality on the undirected graph G shown in Figure 2.58.

Figure 2.58 Eigenvector Centrality Example of an Undirected Graph



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

```

data mylib.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      = mylib.LinkSetIn
  outNodes   = mylib.NodeSetOut;
  centrality
    eigen    = unweight;
run;

```

The output data table `mylib.NodeSetOut` now contains the eigenvector centrality of each node, as shown in Figure 2.59.

Figure 2.59 Eigenvector Centrality Output

node	centr_eigen_unwt
B	1.00000
E	0.75919
F	0.61981
D	0.40226
I	0.35233
J	0.35233
C	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 281.

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 14 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,

$$x_i = \alpha \sum_{j \in N} w_{ji} y_j$$

$$y_j = \beta \sum_{i \in N} w_{ij} x_i$$

where x_i is the authority centrality of node i , y_j is the hub centrality of node j , w_{ij} is the sum of the weights of links from node i to node j , and α and β are constants.

Similar to eigenvector centrality, the definitions of authority and hub centrality can be written in matrix form as

$$\begin{aligned} \mathbf{x} &= \alpha \mathbf{A}^T \mathbf{y} \\ \mathbf{y} &= \beta \mathbf{A} \mathbf{x} \end{aligned}$$

where $\mathbf{A}$ represents the adjacency matrix that corresponds to the outgoing link weights in E and $\mathbf{A}^T$ represents the adjacency matrix that corresponds to the incoming link weights in E . In constructing the matrices $\mathbf{A}$ and $\mathbf{A}^T$, the weights of multilinks are aggregated.

Combining the two equations results in

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

where $\lambda = (\alpha\beta)^{-1}$. Thus, the authority and hub centralities are the principal eigenvectors of $\mathbf{A}^T \mathbf{A}$ and $\mathbf{A} \mathbf{A}^T$, respectively.

To solve this eigenvector problem, PROC NETWORK provides two algorithms: the Jacobi-Davidson algorithm and the power method. You can use the EIGENALGORITHM= option in the CENTRALITY statement to specify which algorithm to use. EIGENALGORITHM=JACOBIDAVIDSON, which is the default, specifies the Jacobi-Davidson algorithm 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.

OUTNODES= Data Table

The results are provided in the nodes output data table that you specify in the OUTNODES= option in the PROC NETWORK statement.

The OUTNODES= data table contains the hub and authority centralities of each node. This data table contains the following column:

- node: the node label

If you specify HUB=UNWEIGHT or HUB=BOTH, the following column also appears in the nodes output data table:

- centr_hub_unwt: the unweighted hub centrality of the node

If you specify HUB=WEIGHT or HUB=BOTH, the following column also appears in the nodes output data table:

- centr_hub_wt: the weighted hub centrality of the node

If you specify AUTH=UNWEIGHT or AUTH=BOTH, the following column also appears in the nodes output data table:

- centr_auth_unwt: the unweighted authority centrality of the node

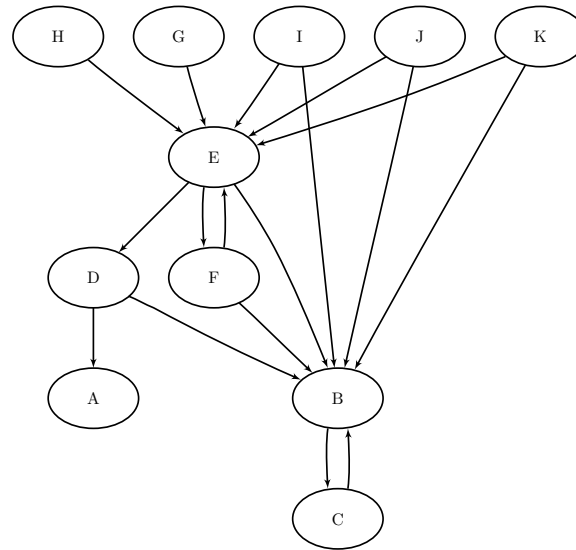
If you specify AUTH=WEIGHT or AUTH=BOTH, the following column also appears in the nodes output data table:

- centr_auth_wt: the weighted authority centrality of the node

Hub and Authority Centrality of a Directed Graph

The following example illustrates the use of hub and authority scoring on the directed graph G shown in Figure 2.60. 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 2.60 Hub and Authority Centrality Example of a Directed Graph



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

```

data mylib.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        = mylib.LinkSetIn
  outNodes     = mylib.NodeSetOut;
  centrality
    hub        = unweight
    auth       = unweight;
run;

```

The output data table `mylib.NodeSetOut` now contains the hub and authority scores of each node, as shown in Figure 2.61.

Figure 2.61 Hub and Authority Centrality Output

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

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.

PageRank Centrality

PageRank centrality, named after Google's cofounder Larry Page, was originally proposed for the application of ranking web pages in a search engine.

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

$$x_i = \frac{1 - \alpha}{|N|} + \alpha \left(\sum_{\substack{j \in N: \\ w_j^{\text{out}} \neq 0}} \frac{w_{ji}}{w_j^{\text{out}}} x_j + \sum_{\substack{j \in N: \\ w_j^{\text{out}} = 0}} \frac{1}{|N|} x_j \right)$$

where x_i is the PageRank score of node i , w_{ij} is the sum of the weights of links from node i to node j , α is a damping factor, and

$$w_i^{\text{out}} = \sum_{j \in N} w_{ij}$$

is the sum of the outbound link weights from node i .

For the calculation of PageRank centrality, PROC NETWORK implements a version of the power method, as described in Langville and Meyer (2006). The damping factor α , which you can specify in the **PAGER-ANKALPHA=** option in the CENTRALITY statement, is an important concept in the PageRank algorithm. In the PageRank equation, $1 - \alpha$ represents the probability of jumping to a random page. Without the damping factor, pages that have no incoming links would have a PageRank score of zero, so these pages cannot be distinguished by PageRank scores. The damping factor also affects the convergence of the power method. Typically, many more iterations are required as the damping factor is set closer to 1.

When the PageRank algorithm is applied to undirected graphs, each undirected link is treated as two directed links that go in both directions.

OUTNODES= Data Table

The results are provided in the nodes output data table that you specify in the OUTNODES= option in the PROC NETWORK statement.

The OUTNODES= data table contains the PageRank centrality of each node. This data table contains the following column:

- node: the node label

If you specify PAGERANK=UNWEIGHT or PAGERANK=BOTH, the following column also appears in the nodes output data table:

- centr_pagerank_unwt: the unweighted PageRank centrality of the node

If you specify PAGERANK=WEIGHT or PAGERANK=BOTH, the following column also appears in the nodes output data table:

- centr_pagerank_wt: the weighted PageRank centrality of the node

PageRank Centrality of an Undirected Graph

This section concludes with an example that illustrates the use of PageRank centrality on the directed graph *G* that is shown in [Figure 2.60](#) in the previous section, “[Hub and Authority Scoring](#)” on page 117.

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

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

The output data table mylib.NodeSetOut now contains the PageRank score of each node, as shown in [Figure 2.62](#).

Figure 2.62 PageRank Centrality of a Directed Graph (Damping Factor = 0.85)

node	centr_pagerank_unwt
B	0.38440
C	0.34291
E	0.08089
F	0.03909
D	0.03909
A	0.03278
K	0.01617
G	0.01617
H	0.01617
I	0.01617
J	0.01617

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

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

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

The new PageRank scores are shown in [Figure 2.63](#).

Figure 2.63 PageRank Centrality of a Directed Graph (Damping Factor = 0.999)

node	centr_pagerank_unwt
B	0.49904
C	0.49867
E	0.00070
D	0.00035
F	0.00035
A	0.00029
G	0.00012
H	0.00012
I	0.00012
J	0.00012
K	0.00012

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

Clique Enumeration

A *clique* of a graph $G = (N, E)$ 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 34. The clique algorithm works only with undirected simple graphs (with no self-links).

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

Output Data Tables

Depending on which options you specify, the clique enumeration algorithm produces additional output data tables, as described in the following sections.

OUT= Data Table

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

- `clique`: the clique identifier
- `node`: the node label

OUTNODES= Data Table

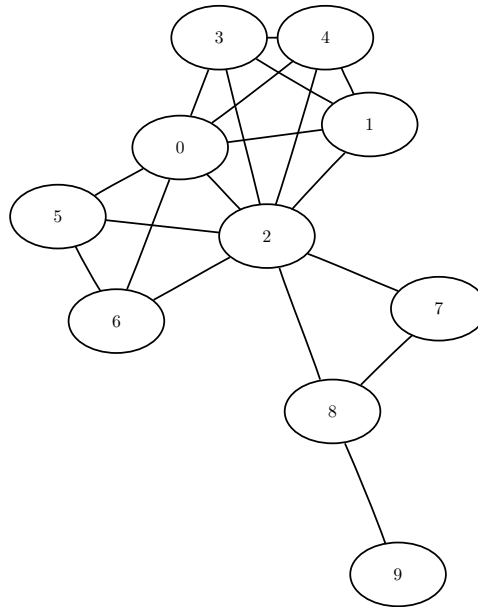
If `CLIQUENUMBER=TRUE`, the results of the clique algorithm are added to the output data table that you specify in the `OUTNODES=` option in the PROC NETWORK statement. Each node is listed in the output data table along with the variable `clique_number` to define the node clique number that the algorithm found for each node. This data table contains (at least) the following columns:

- `node`: the node label
- `clique_number`: the node clique number of each node

Maximal Cliques of an Undirected Graph

This section illustrates the use of the clique algorithm on the undirected graph G shown in Figure 2.64.

Figure 2.64 Undirected Graph G



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

```

data mylib.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 and output the results in the data table mylib.Cliques. They also calculate the node clique numbers for each node and output the results in the data table mylib.NodeSetOut.

```

proc network
  links          = mylib.LinkSetIn
  outNodes       = mylib.NodeSetOut;
  clique
    maxCliques   = all
    cliqueNumber = true
    out          = mylib.Cliques;
run;
%put &_NETWORK_;

```

The progress of the procedure is shown in Figure 2.65.

Figure 2.65 PROC NETWORK Log for Maximal Cliques

```

NOTE: -----
NOTE: Running NETWORK.
NOTE: -----
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 19.
NOTE: Processing clique enumeration using 80 threads across 1 machines.
NOTE: The algorithm found 4 maximal cliques.
NOTE: Processing clique enumeration used 0.01 (cpu: 0.01) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.084763 seconds.
NOTE: The data set MYLIB.NODESETOUT has 10 observations and 2 variables.
NOTE: The data set MYLIB.CLQUES has 14 observations and 2 variables.
STATUS=OK  PROBLEM_TYPE=CLIQUE  SOLUTION_STATUS=OK  NUM_CLQUES=4  CLIQUENUMBER=5
CPU_TIME=0.75  REAL_TIME=0.08

```

The output data table mylib.Clques now contains the maximal cliques of the input graph, as shown in Figure 2.66.

Figure 2.66 Maximal Cliques of an Undirected Graph

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

The output data table mylib.NodeSetOut now contains the node clique number of each node, as shown in Figure 2.67.

Figure 2.67 Node Clique Numbers of an Undirected Graph

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

The following statements use the FEDSQL procedure as a convenient way to create a data table of clique sizes (mylib.CliqueSizes):

```
proc fedsql sessref=my sess;
  create table CliqueSizes as
  select clique, count(*) as size
  from Cliques
  group by clique;
quit;
```

The output data table (mylib.CliqueSizes) now contains the number of nodes in each clique, as shown in Figure 2.68.

Figure 2.68 Sizes of Maximal Cliques of an Undirected Graph

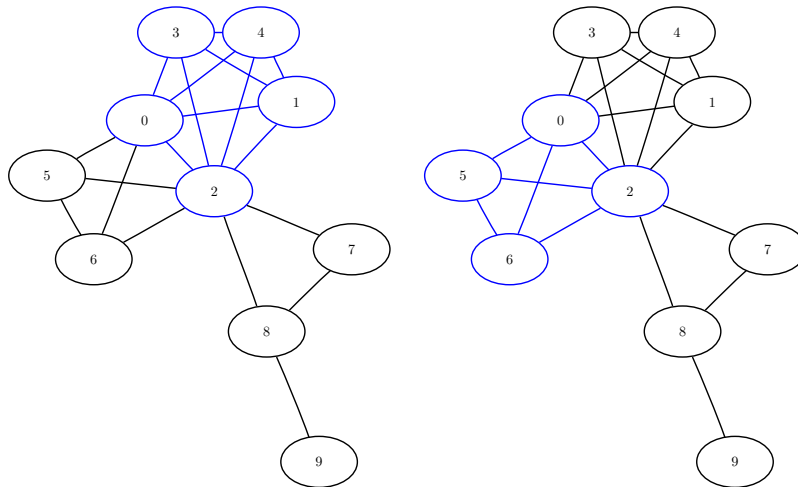
clique	SIZE
1	5
2	4
3	3
4	2

The maximal cliques are shown graphically in Figure 2.69 and Figure 2.70.

Figure 2.69 Maximal Cliques C^1 and C^2

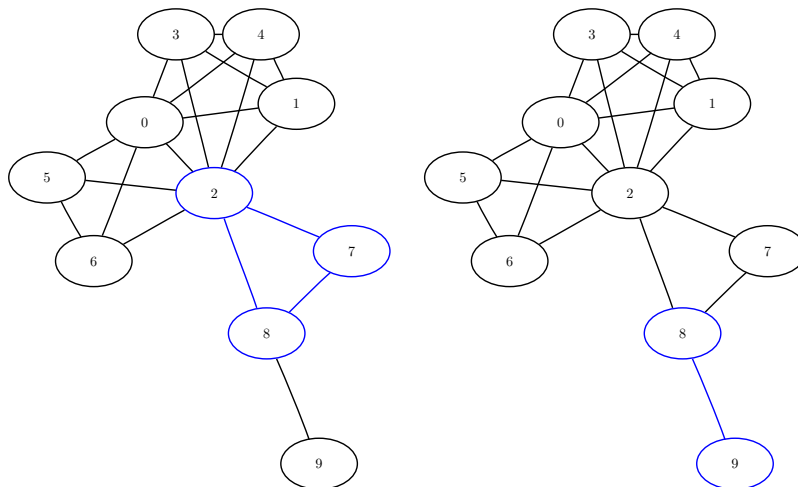
$$C^1 = \{0, 1, 2, 3, 4\}$$

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


Figure 2.70 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 35.

PROC NETWORK implements two heuristic algorithms for finding communities in directed or undirected graphs: the Louvain algorithm proposed by Blondel et al. (2008) and the label propagation algorithm proposed by Raghavan, Albert, and Kumara (2007). Given a graph $G = (N, E)$, the algorithms run in time $O(|N| + |E|)$.

Modularity 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 (weighted) degree of each node. To compute modularity, you can equivalently think of an unweighted graph as a weighted graph in which all weights have a value of 1.

Let k_i represent the weighted degree of node i in an undirected graph, where k_i is defined as the sum of the weights of links incident with node i (with self-links added twice). Then, for an undirected graph $G = (N, E)$, modularity is defined as

$$Q = \frac{1}{2W_E} \sum_{i \in N} \sum_{j \in N} \left(k_{ij} - \frac{k_i k_j}{2W_E} \right) \Delta(c_i, c_j)$$

where W_E is the sum of all link weights in G ; k_{ij} is the sum of the weights of links between node i and node j (with self-links added twice); c_i is the community to which node i belongs; and $\Delta(c_i, c_j)$ is the Kronecker delta symbol, defined as 1 if $c_i = c_j$ and 0 otherwise. If $W_E = 0$, then the modularity is defined as 1.

For a directed graph, modularity is based on weighted out-degrees and weighted in-degrees of the nodes and is defined as

$$Q = \frac{1}{W_E} \sum_{i \in N} \sum_{j \in N} \left(k_{ij}^{\text{out}} - \frac{k_i^{\text{out}} k_j^{\text{in}}}{W_E} \right) \Delta(c_i, c_j)$$

where k_{ij}^{out} is the sum of the weights of links from node i to node j , k_i^{out} is the sum of the weights of outgoing links from node i , and k_j^{in} is the sum of the weights of incoming links to node j .

The Louvain algorithm partitions nodes into communities by heuristically optimizing modularity. 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 positive link weights from the current node; break ties randomly if necessary. For a directed graph, when a node updates its label, it takes the label of one of its out-neighbors. Repeat this step until there are no more movements.

While the label propagation algorithm is running, a node can update its label synchronously or asynchronously. If you specify `LABELUPDATEMODE=ASYNCHRONOUS`, then at each iteration, nodes update their labels according to the current labels of their neighbors. If you specify `LABELUPDATEMODE=SYNCHRONOUS`, then at each iteration, nodes update their labels according to the labels of their neighbors at the previous iteration. You can run the algorithm in parallel when using the synchronous update mode. However, this type of simultaneous updating often leads to oscillating labels because of the bipartite subgraph structure often present in large graphs. To address this issue, the 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.

Fix and Warm Start Community Detection

When you want to perform community detection, you might want to initialize or fix some nodes into prearranged communities on the basis of prior invocations of the `COMMUNITY` statement or prior knowledge about your data. By default, all three community detection algorithms start with each node in its own community and have nodes change communities on the basis of conditions specific to the algorithm being run. In `PROC NETWORK` you can use the `WARMSTART=` option to change this initial configuration for some or all of the nodes. The value that is specified in the `WARMSTART=` option must be the name of a variable in the `NODES=` input table that has nonnegative integer values. Nodes that have the same value are placed in the same community, and each node that has a missing value (or is not specified in the `NODES=` input table) is placed in its own community.

Warm starting can be beneficial if you are performing community detection across multiple snapshots of a network, $(G_0, \dots, G_n)$, where there are relatively few changes (such as link additions or deletions) between each consecutive snapshot. One benefit is that you can improve the run time of the algorithm by warm starting the nodes to a state that is closer to convergence than is the default initialization. For example, if you use the communities that are found in G_{i-1} to warm start many of the nodes for community detection on G_i , you can probably achieve convergence sooner than having each node start in its own community.

Another benefit to warm starting community detection is that you can obtain more stable communities. In general, there can be many solutions that have similar quality in terms of community metrics, so even a small change to the graph can cause significantly different communities to be output. However, by warm starting the nodes, you are likely to see more stable communities, which can be helpful if you want to track communities over time. In general, warm starting community detection on G_i by using exactly the communities that are found in G_{i-1} might hurt the quality of the solution, depending on the amount of change between G_{i-1} and G_i . To alleviate this issue, you can warm start only the nodes that have had no link additions or deletions between G_{i-1} and G_i .

In addition to warm starting community detection, you can also fix nodes in a community by using the `FIX=` option, provided that you specify `ALGORITHM=LOUVAIN` or `ALGORITHM=LABELPROPAGATION` with `LABELUPDATEMODE=ASYNCHRONOUS`. The value that is specified in the `FIX=` option must be the name of a variable in the `NODES=` input table that has nonnegative integer values. In this case, nodes that have the same value are forced into the same community and cannot be split into different communities (unlike warm start, where the nodes can change communities as the algorithm progresses). Nodes that are fixed in same community can still be merged into other communities and become part of a larger community. Each algorithm determines whether to move a fixed group of nodes into another community by treating the fixed group as a unit, which is accomplished by aggregating the fixed group into a supernode.

You can specify both the `FIX=` and `WARMSTART=` options simultaneously. If both options are specified, the values of the variables are assumed to be related. For example, if nodes i and j have a value of 1 for the `FIX` variable and node k has a value of 1 for the `WARMSTART` variable, then nodes i , j , and k are initialized in the same community. During the algorithm, k can move to a different community than do i and j , but i and j are forced to remain in the same community.

One consequence of using the `FIX=` or `WARMSTART=` option is that you are reducing the number of communities that a node can move to during the algorithm. If you initialize many nodes in a small set of communities, then you might end up with fewer communities than if you had run community detection without these options. None of these algorithms perform global optimization, so warm starting or fixing the nodes in a bad initial state might result in low-modularity communities.

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=LABELPROPAGATION` in the `COMMUNITY` statement, the resolution value represents a density threshold. A node cannot take the label of a community if the density of that community, after adding the node, would be less than the resolution value. In this case, the *density* of a community is defined as the number of links inside the community divided by the total number of possible links (even for directed graphs). For more information about resolution values for label propagation, see Traag, Van Dooren, and Nesterov (2011).

If you specify `ALGORITHM=LABELPROPAGATION` and `LABELUPDATEMODE=SYNCHRONOUS`, you can supply one or more resolution values. If you supply multiple resolution values at one time, 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. Also, because the algorithm decides the labels for nodes synchronously, the output communities are not guaranteed to have densities larger than the resolution value. However, in general, a larger resolution value is likely to result in communities that contain fewer nodes and a higher density.

The value that you specify in the `RESOLUTIONLIST=` option can have a major impact on the run time of the label propagation algorithm. When you specify a large resolution value, 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 283 shows the use of the `RESOLUTIONLIST=` option in finding communities.

Large Graphs

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

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

Output Data Tables

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

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

OUTNODES= Data Table

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

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

OUTLINKS= Data Table

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

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

OUTLEVEL= Data Table

The `OUTLEVEL=` data table describes the number of communities and their corresponding modularity values at various resolution levels. It contains the following columns:

- `level`: resolution level number
- `resolution`: resolution value
- `communities`: number of communities at the current resolution level
- `modularity`: modularity value at the current resolution level

OUTCOMMUNITY= Data Table

The OUTCOMMUNITY= data table describes various properties about 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
- intra_links: number of links that connect two nodes in the community
- inter_links: number of links (outgoing links for a directed graph) that connect from a node in the community to a node outside the community
- density: intra_links divided by the total number of possible links in the community
- cut_ratio: inter_links divided by the number of possible links from the community to nodes outside the community
- conductance: the fraction of links from a node in the community that connect to nodes in a different community. This fraction is calculated as follows:
 - for an undirected graph: $\text{inter_links} / (\text{inter_links} + 2 * \text{intra_links})$
 - for a directed graph: $\text{inter_links} / (\text{inter_links} + \text{intra_links})$

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. If a node has no links, it is not included in the table. If a node has a total link weight of 0, then each intensity value is 0. For directed graphs, only the outgoing links of a node are used in the intensity calculation. This data table is computationally expensive to produce, and it requires a large amount of disk space. Therefore, the data table contains only results that correspond to the smallest value of the RESOLUTIONLIST= option. The data table contains the following columns:

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

OUTCOMMLINKS= Data Table

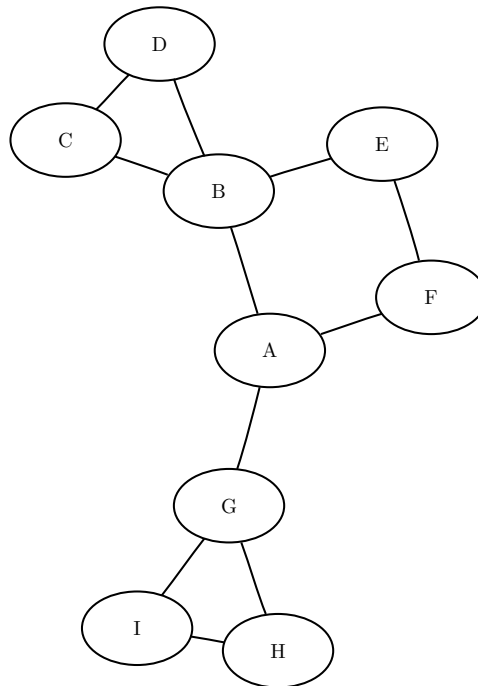
The OUTCOMMLINKS= data table describes how communities are connected. It contains the following columns:

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

Community Detection on an Undirected Graph

This section illustrates the use of the community detection algorithm on the undirected graph G shown in Figure 2.71.

Figure 2.71 Undirected Graph G



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

```

data mylib.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          = mylib.LinkSetIn
  outNodes       = mylib.NodeSetOut;
  community
    resolutionList = 1.0 0.5
    outLevel       = mylib.CommLevelOut
    outCommunity   = mylib.CommOut
    outOverlap     = mylib.CommOverlapOut
    outCommLinks   = mylib.CommLinksOut;
run;
```

The output data table mylib.NodeSetOut contains the community identifier of each node, as shown in Figure 2.72.

Figure 2.72 Community Detection on an Undirected Graph: Nodes Output

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

The output data table mylib.CommLevelOut contains summary information at each resolution level, as shown in Figure 2.73.

Figure 2.73 Community Detection on an Undirected Graph: Level Output

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

The output data table mylib.CommOut contains the number of nodes in each community, as shown in Figure 2.74.

Figure 2.74 Community Detection on an Undirected Graph: Community Summary

level	resolution	community	nodes	intra_links	inter_links	density	cut_ratio	conductance
1	1.0	1	3	2	3	0.66667	0.16667	0.42857
1	1.0	2	3	3	2	1.00000	0.11111	0.25000
1	1.0	3	3	3	1	1.00000	0.05556	0.14286
2	0.5	1	6	7	1	0.46667	0.05556	0.06667
2	0.5	2	3	3	1	1.00000	0.05556	0.14286

The output data table `mylib.CommOverlapOut` contains community overlap information, as shown in Figure 2.75.

Figure 2.75 Community Detection on an Undirected Graph: Community Overlap

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

The output data table `mylib.CommLinksOut` describes how the communities are connected, as shown in Figure 2.76.

Figure 2.76 Community Detection on an Undirected 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

Fixing Nodes for Community Detection

This section continues the example in the section “Community Detection on an Undirected Graph” on page 134 and illustrates fixing nodes together for community detection on the graph shown in Figure 2.71. Suppose you want to fix nodes *A* and *B* in the same community, nodes *C* and *D* in the same community, and nodes *H* and *I* in the same community. In order to do this, you must specify a nodes data table that has a variable that defines these fixed groups. The following DATA step creates the nodes data table:

```
data mylib.NodeSetIn;
    input node $ fixGroup @@;
    datalines;
A 1 B 1
C 2 D 2
H 3 I 3
;
```

The following statements perform community detection by using fixed node groups:

```
proc network
    nodes          = mylib.NodeSetIn
    links          = mylib.LinkSetIn
    outNodes       = mylib.NodeSetOut;
    community
```



```

resolutionList = 1.0
fix            = fixGroup;
run;

```

The output data table `mylib.NodeSetOut` contains the community identifier of each node, as shown in Figure 2.77.

Figure 2.77 Fixing Nodes for Community Detection: Nodes Output

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

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 38.

You can use three algorithms to find connected components in an undirected graph: the depth-first search algorithm (ALGORITHM=DFS); the union-find algorithm (ALGORITHM=UNIONFIND); and the afforest algorithm (ALGORITHM=AFFOREST), which is described in Sutton, Ben-Nun, and Barak (2018). For a graph $G = (N, E)$, each algorithm runs in time $O(|N| + |E|)$ and can usually scale to very large graphs. For undirected graphs, the default automatically selects either the union-find algorithm (ALGORITHM=UNIONFIND) or the afforest algorithm (ALGORITHM=AFFOREST). For directed graphs, only the depth-first search algorithm (ALGORITHM=DFS) is available.

For undirected graphs, you can also execute the union-find algorithm in a distributed manner across multiple machines by specifying `DISTRIBUTED=TRUE` in the PROC NETWORK statement. For directed graphs, you can compute strongly connected components by specifying `DIRECTION=DIRECTED` in that statement. To compute weakly connected components, you can ignore the edge directions by specifying `DIRECTION=UNDIRECTED`.

The results of the CONNECTEDCOMPONENTS statement 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 that 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.

Output Data Tables

Depending on the options selected, the `CONNECTEDCOMPONENTS` statement produces an additional output data table as described in the following section.

OUT= Data Table

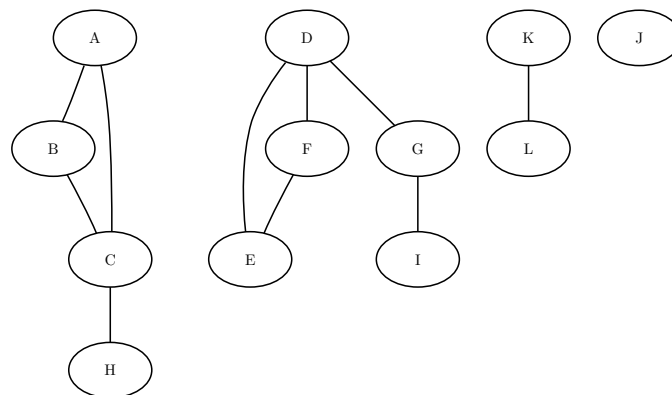
The `OUT=` data table describes the number of nodes in each connected component. This data table contains the following columns:

- `concomp`: connected component identifier
- `nodes`: number of nodes contained in the connected component

Connected Components of an Undirected Graph

This section illustrates the use of the `CONNECTEDCOMPONENTS` statement on the undirected graph G shown in Figure 2.78.

Figure 2.78 Undirected Graph G



The undirected graph G can be represented by the following nodes and links data tables:

```
data mylib.NodeSetIn;
  input node $;
  datalines;
J
;
data mylib.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 `mylib.NodeSetOut`:

```

proc network
  nodes      = mylib.NodeSetIn
  links      = mylib.LinkSetIn
  outNodes   = mylib.NodeSetOut;
  connectedComponents;
run;

```

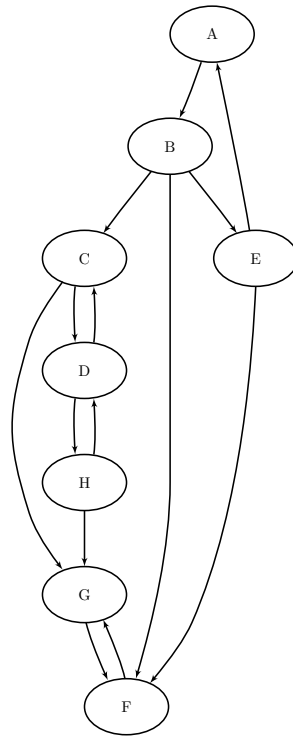
The output data table `mylib.NodeSetOut` contains the connected components of the input graph, as shown in [Figure 2.79](#).

Figure 2.79 Connected Components of an Undirected Graph

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

Strongly Connected Components of a Directed Graph

This section illustrates the use of the `CONNECTEDCOMPONENTS` statement on the directed graph G shown in [Figure 2.80](#) to compute its strongly connected components.

Figure 2.80 Directed Graph *G*

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

```

data mylib.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 strongly connected components and output the results in the data tables mylib.NodeSetOut, mylib.LinkSetOut, and mylib.ConCompOut:

```

proc network
  direction = directed
  links      = mylib.LinkSetIn
  outNodes   = mylib.NodeSetOut
  outLinks   = mylib.LinkSetOut;
  connectedComponents
    out      = mylib.ConCompOut;
run;

```

The output data table mylib.NodeSetOut, shown in Figure 2.81, now contains the node mappings for the strongly connected components of the input graph.

Figure 2.81 Node Mappings for the Strongly Connected Components of a Directed Graph

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

The output data table mylib.LinkSetOut, shown in [Figure 2.82](#), now contains the link mappings for the strongly connected components of the input graph.

Figure 2.82 Links Mappings for the Strongly Connected Components of a Directed Graph

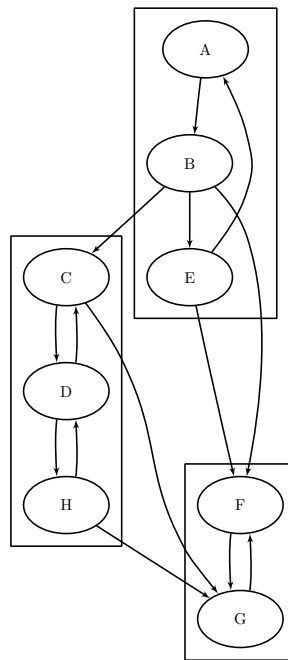
from	to	concomp
H	G	.
B	C	.
B	F	.
C	G	.
E	F	.
A	B	1
B	E	1
E	A	1
C	D	2
D	C	2
D	H	2
H	D	2
F	G	3
G	F	3

The output data table mylib.ConCompOut, shown in [Figure 2.83](#), now contains the number of nodes in each strongly connected component of the input graph.

Figure 2.83 Summary for the Strongly Connected Components of a Directed Graph

concomp	nodes
1	3
2	3
3	2

The strongly connected components are represented graphically in [Figure 2.84](#).

Figure 2.84 Strongly Connected Components of Graph G 

Core Decomposition

An alternative to community detection for finding cohesive subgroups is a method of extracting k -cores known as *core decomposition*. Although this method is generally not as powerful as community detection for extracting a detailed community structure, it can provide a coarse approximation of cohesive structure at a very low computational cost. Let $G = (N, E)$ define a graph that contains nodes N and links E , and let $G_S = (S, E_S)$ be an induced subgraph on node set S . The subgraph G_S is a k -core if and only if for every node $j \in S$, the degree of j 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 39.

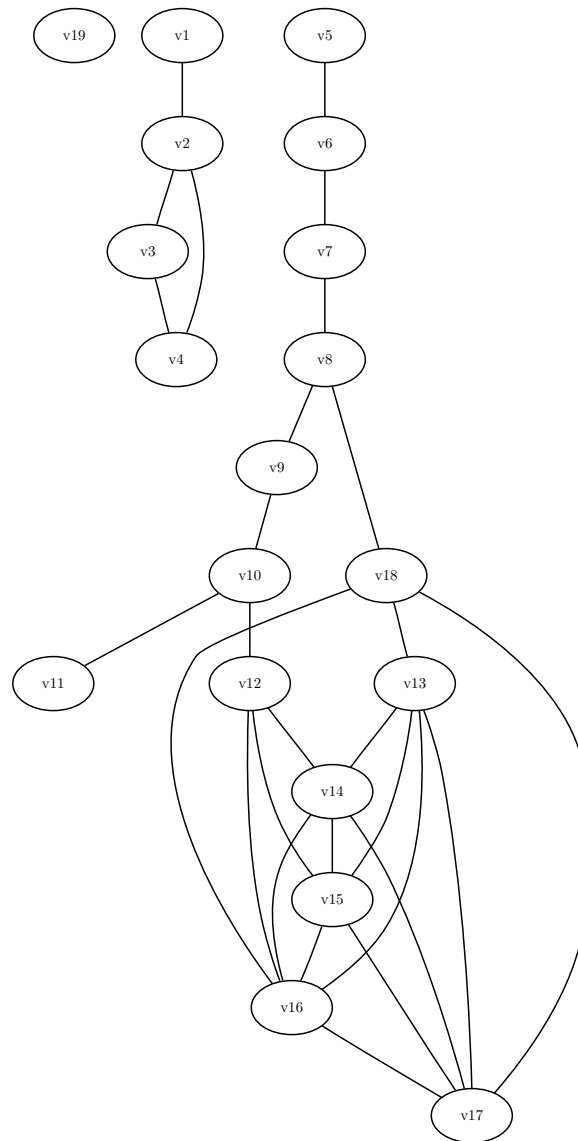
The results of the core decomposition algorithm are given in the output nodes data table that you specify in the OUTNODES= option in the PROC NETWORK statement. For each node in the nodes data table, the variable core_out identifies its *core number*, the highest-order core that contains this node.

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

Core Decomposition of an Undirected Graph

This section illustrates the use of the core decomposition algorithm on the undirected graph G shown in Figure 2.85.

Figure 2.85 Undirected Graph



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

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

```

;

data mylib.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 mylib.NodeSetOut:

```

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

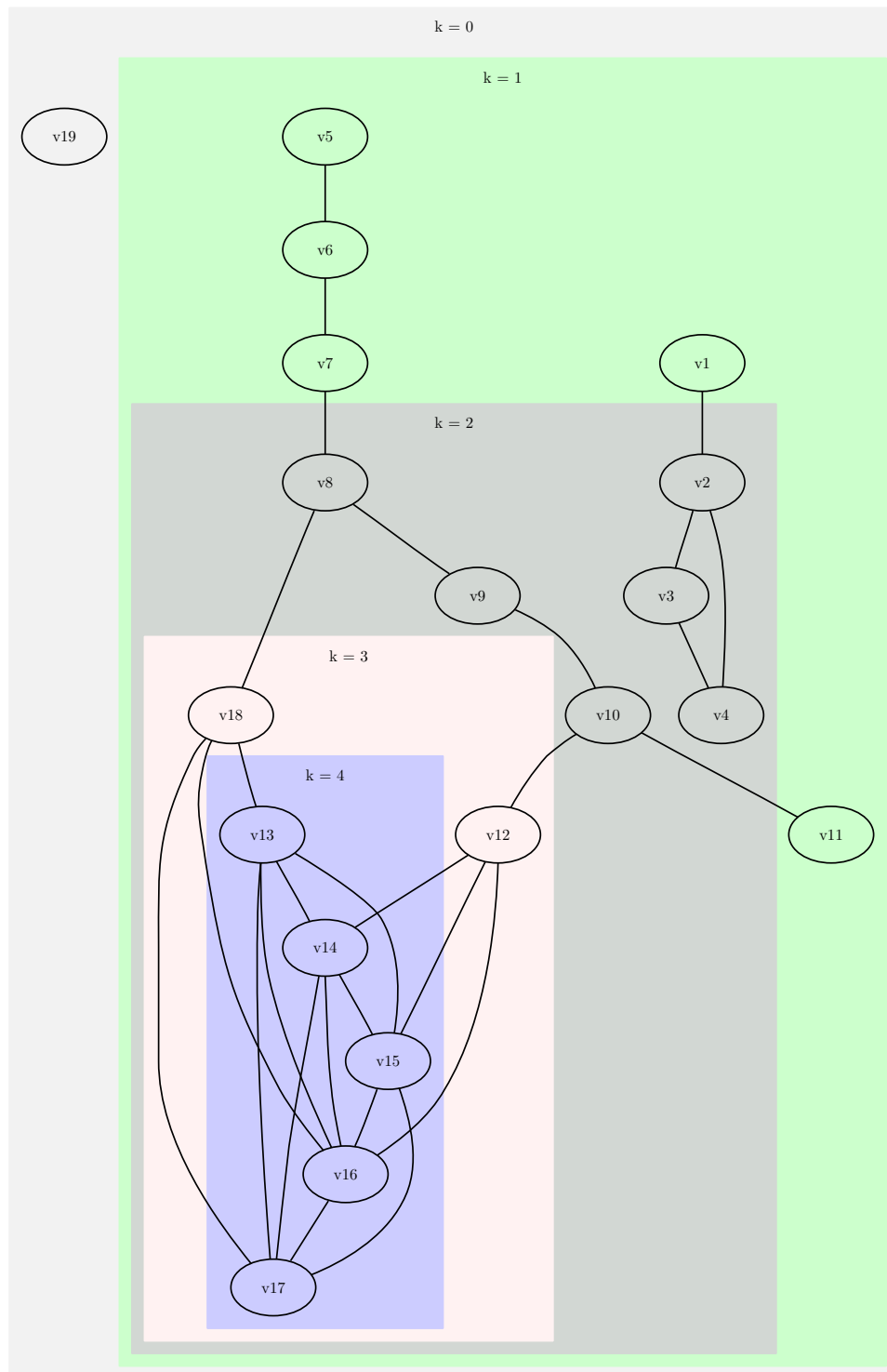
```

The nodes output data table mylib.NodeSetOut contains the core number (variable core_out) for each node, as shown in [Figure 2.86](#).

Figure 2.86 Core Decomposition of an Undirected Graph

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

[Figure 2.87](#) shows the graph layered by its core number.

Figure 2.87 Core Decomposition

Cycle Enumeration

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

In PROC NETWORK, you can find (or just count) the elementary cycles of an input graph by specifying the CYCLE statement. The options for this statement are described in the section “CYCLE Statement” on page 39. To find the cycles and report them in an output data table, use either the OUTCYCLESNODES= option or the OUTCYCLESLINKS= option. To simply count the cycles, omit the output data table options.

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 $A \rightarrow C \rightarrow B \rightarrow A$).

By default, PROC NETWORK uses depth-first search to quickly identify whether a cycle exists. This algorithm runs in time $O(|N| + |E|)$. This algorithm should scale to very large graphs. If filters are used or if the specified maximum number of cycles (MAXCYCLES=) is greater than 1, then enumerative algorithms are used.

The default algorithm that PROC NETWORK uses to enumerate cycles when the value of the MAXLENGTH= option is greater than 20 (ALGORITHM=BACKTRACK) is a variant of the algorithm in Johnson (1975). This algorithm runs in time $O((|N| + |E|)(c + 1))$, where c is the number of elementary cycles in the graph. So the algorithm should scale to large graphs that contain few cycles. However, some graphs can have a very large number of cycles, so the algorithm might not scale. The default when the value of the MAXLENGTH= option is less than or equal to 20 (ALGORITHM=BUILD) is described in Liu and Wang (2006). This algorithm is usually much faster than the backtracking algorithm when the length of the cycles is sufficiently restricted.

If MAXCYCLES=ALL and there are many cycles, the output data tables can become very large. It might be beneficial to check the number of cycles before you try to create an output data table. For more information about these options, see the section “CYCLE Statement” on page 39.

Output Data Tables

Depending on the specified options, the cycle algorithm produces additional output data tables as described in the following section.

OUTCYCLESNODES= Data Table

The OUTCYCLESNODES= data table describes the enumerated cycles as a sequence of nodes. This data table contains the following columns:

- cycle: the cycle identifier
- order: the order (sequence) of the node in the cycle
- node: the node label

OUTCYCLESLINKS= Data Table

The OUTCYCLESLINKS= data table describes the enumerated cycles as a sequence of links. This data table contains the following columns:

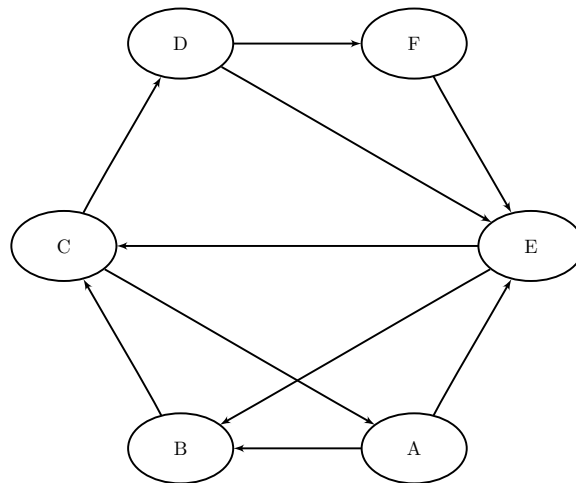
- cycle: the cycle identifier
- order: the order (sequence) of the link in the cycle
- from: the *from* node label
- to: the *to* node label

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

Cycle Enumeration of a Directed Graph

This section provides a simple example of using the cycle enumeration algorithm on the directed graph G shown in Figure 2.88. For a more detailed example involving both cycle enumeration and transitive closure, see “Example 2.10: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System” on page 300.

Figure 2.88 Directed Graph G



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

```

data mylib.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          = mylib.LinkSetIn;
  cycle
    maxCycles = all;
run;
%put &_NETWORK_;

```

The result is written to the log of the NETWORK procedure, as shown in [Figure 2.89](#).

Figure 2.89 PROC NETWORK Log: Count the Number of Cycles in a 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 enumeration using 1 threads across 1 machines.
NOTE: The algorithm found 7 cycles.
NOTE: Processing cycle enumeration used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.04935 seconds.
STATUS=OK  PROBLEM_TYPE=CYCLE  SOLUTION_STATUS=OK  NUM_CYCLES=7  CPU_TIME=0.46  REAL_TIME=0.05

```

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

```

proc network
  direction      = directed
  links          = mylib.LinkSetIn;
  cycle
    outCyclesNodes = mylib.CyclesNodes
    outCyclesLinks = mylib.CyclesLinks;
run;

```

The output data table `mylib.CyclesNodes` now contains the nodes of the first cycle that is found in the input graph, as shown in [Figure 2.90](#).

Figure 2.90 Nodes of the First Cycle Found in a Directed Graph

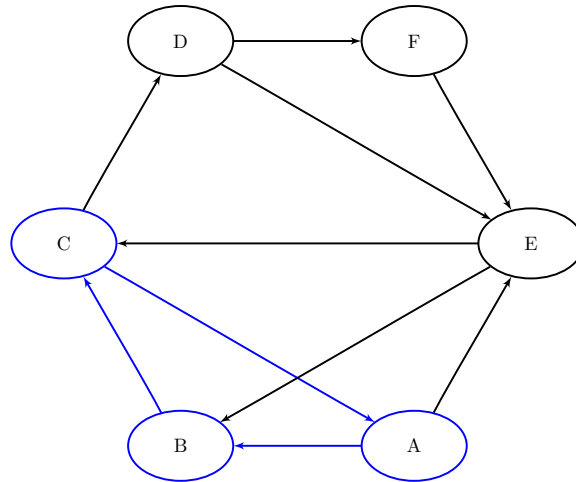
cycle	order	node
1	1	A
1	2	B
1	3	C
1	4	A

The output data table `mylib.CyclesLinks` now contains the links of the first cycle that is found in the input graph, as shown in [Figure 2.91](#).

Figure 2.91 Links of the First Cycle Found in a Directed Graph

cycle	order	from	to
1	1	A	B
1	2	B	C
1	3	C	A

The first cycle that is found in the input graph is shown graphically in [Figure 2.92](#).

Figure 2.92 $A \rightarrow B \rightarrow C \rightarrow A$ 

The following statements return all the cycles in the graph:

```

proc network
  direction      = directed
  links          = mylib.LinkSetIn;
  cycle
    maxCycles     = all
    outCyclesNodes = mylib.CyclesNodes;
run;

```

The output data table mylib.CyclesNodes now contains the nodes of all the cycles in the input graph, as shown in [Figure 2.93](#).

Figure 2.93 Nodes of All Cycles in a Directed Graph

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

The six additional cycles are shown graphically in [Figure 2.94](#) through [Figure 2.96](#).

Figure 2.94 Cycles

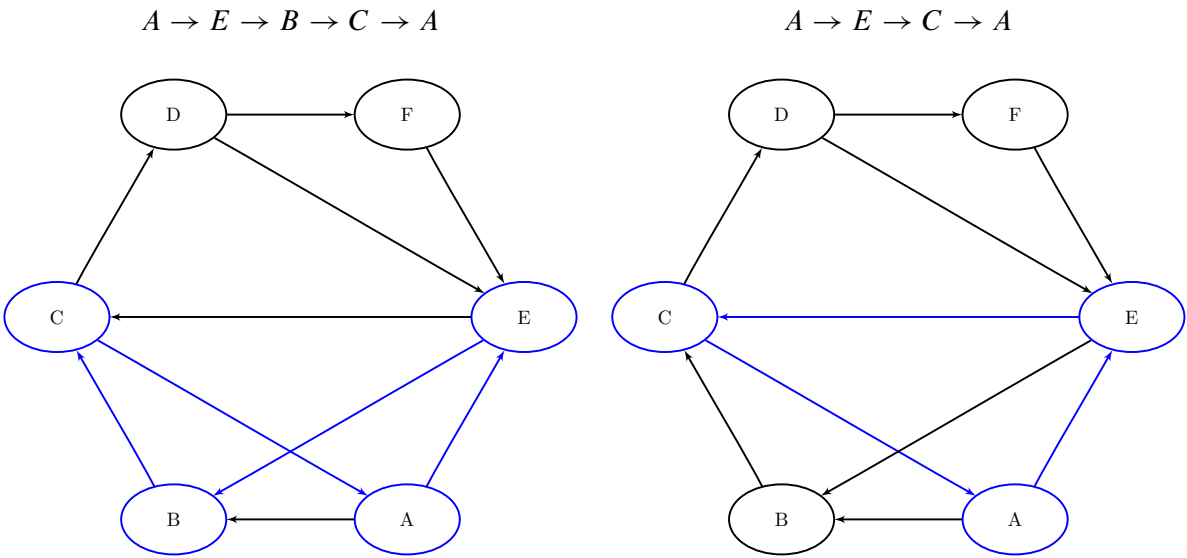


Figure 2.95 Cycles

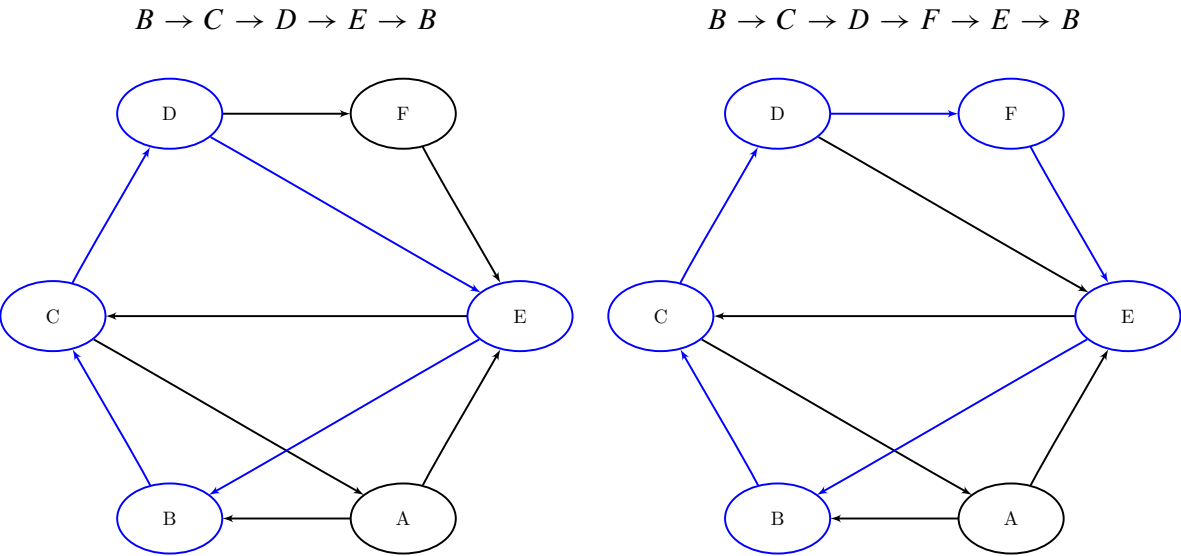
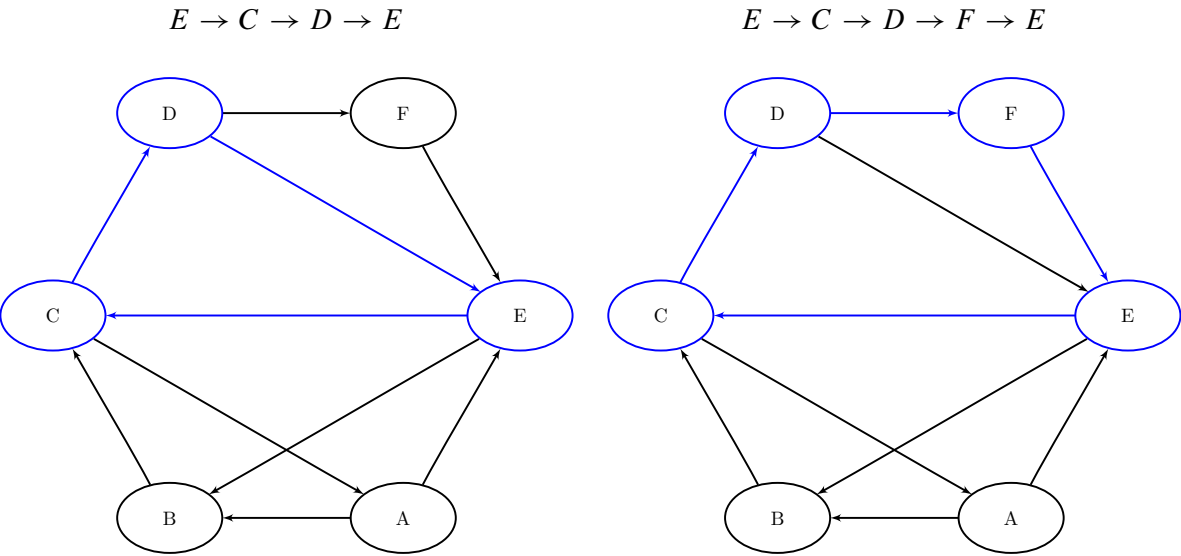


Figure 2.96 Cycles



Node Similarity

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

In PROC NETWORK, you can calculate node similarity by using the NODESIMILARITY statement. The options for this statement are described in the section “[NODESIMILARITY Statement](#)” on page 45.

PROC NETWORK implements seven different measures of node similarity: five measures that are based on the immediate neighborhood (which you specify using the ADAMICADAR=, COMMONNEIGHBORS=, COSINE=, DICE=, and JACCARD= options) and two measures that are based on inner products between vector representations of nodes (first- and second-order proximity, which you specify in the PROXIMITY-ORDER= option when you also specify VECTOR=TRUE). The vector representations are calculated using the link sampling algorithm in Tang et al. (2015).

One simple measure of structural node similarity in a network is the number of neighbors that two nodes have in common. That is, the nodes are similar in proportion to the size of their overlapping neighborhoods.

In an undirected graph, $\hat{N}_i$ represents the set of neighbors, excluding node i itself (that is, unique nodes that are connected by links, excluding self-links, incident with node i). In a directed graph, $\hat{N}_i^{\text{out}}$ represents the set of out-neighbors, excluding node i itself (that is, unique nodes that are connected by outgoing links, excluding self-links, from node i), and $\hat{N}_i^{\text{in}}$ represents the set of in-neighbors, excluding node i itself (that is, unique nodes that are connected by incoming links, excluding self-links, to node i). The common neighbors similarity measure between nodes i and j is defined as

$$S_{ij}^{\text{CN}} = |\hat{N}_i \cap \hat{N}_j|$$

when DIRECTION=UNDIRECTED, or

$$S_{ij}^{\text{CN}} = |\hat{N}_i^{\text{out}} \cap \hat{N}_j^{\text{out}}|$$

when DIRECTION=DIRECTED.

Based on the common neighbors measure, high-degree nodes are likely to attain large similarity scores, even when only a small fraction of their neighbors is shared. One way to correct this bias is to divide by the size of the union of the two neighborhoods. The result is called the Jaccard similarity measure, which is defined as

$$S_{ij}^{\text{J}} = \frac{|\hat{N}_i \cap \hat{N}_j|}{|\hat{N}_i \cup \hat{N}_j|}$$

when DIRECTION=UNDIRECTED, or

$$S_{ij}^{\text{J}} = \frac{|\hat{N}_i^{\text{out}} \cap \hat{N}_j^{\text{out}}|}{|\hat{N}_i^{\text{out}} \cup \hat{N}_j^{\text{out}}|}$$

when DIRECTION=DIRECTED. By convention, when $i = j$, the Jaccard similarity measure is 1. When $i \neq j$ and the union of the two neighborhoods is empty, the Jaccard similarity measure is 0.

A second way to correct this bias is to divide by the average size of the two neighborhoods. The result, which was independently introduced by Dice (1945) and Sørensen (1948), is called the Dice similarity measure (also known as the Sørensen-Dice index) and is defined as

$$S_{ij}^{\text{D}} = \frac{2|\hat{N}_i \cap \hat{N}_j|}{|\hat{N}_i| + |\hat{N}_j|}$$

when DIRECTION=UNDIRECTED, or

$$S_{ij}^D = \frac{2|\hat{N}_i^{\text{out}} \cap \hat{N}_j^{\text{out}}|}{|\hat{N}_i^{\text{out}}| + |\hat{N}_j^{\text{out}}|}$$

when DIRECTION=DIRECTED. Much like the Jaccard similarity measure, the Dice similarity measure is 1 when $i = j$, and it is 0 when $i \neq j$ and the union of the two neighborhoods is empty.

A third way to correct the bias of the common neighbors similarity measure is to use the cosine of the angle between two vectors that represent the neighbor sets of nodes i and j . The result is called the cosine similarity measure, which is defined as

$$S_{ij}^C = \frac{\sum_{k \in \hat{N}_i \cap \hat{N}_j} \bar{w}_{ik} \bar{w}_{jk}}{\sqrt{\sum_{k \in \hat{N}_i} \bar{w}_{ik}^2} \sqrt{\sum_{k \in \hat{N}_j} \bar{w}_{jk}^2}}$$

when DIRECTION=UNDIRECTED, or

$$S_{ij}^C = \frac{\sum_{k \in \hat{N}_i^{\text{out}} \cap \hat{N}_j^{\text{out}}} \bar{w}_{ik} \bar{w}_{jk}}{\sqrt{\sum_{k \in \hat{N}_i^{\text{out}}} \bar{w}_{ik}^2} \sqrt{\sum_{k \in \hat{N}_j^{\text{out}}} \bar{w}_{jk}^2}}$$

when DIRECTION=DIRECTED, where $\bar{w}_{ij}$ is the sum of parallel link weights. It is defined as

$$\bar{w}_{ij} = \sum_{e \in \delta_i \cap \delta_j} w_e$$

when DIRECTION=UNDIRECTED, or

$$\bar{w}_{ij} = \sum_{e \in \delta_i^{\text{out}} \cap \delta_j^{\text{in}}} w_e$$

when DIRECTION=DIRECTED. By convention, $w_e = 1$ for a graph without link weights. When $i = j$, the cosine similarity measure is 1. When $i \neq j$ and either of the two neighborhoods is empty, the cosine similarity measure is 0.

A fourth way to correct the bias of the common neighbors similarity measure is to discount the contribution of common neighbors that have high degree. This technique, formulated by Adamic and Adar (2003), is known as the Adamic-Adar similarity measure. This similarity measure differs from Jaccard, Dice, and cosine similarity. Whereas Jaccard, Dice, and cosine similarity are relative measures that assign a score to node pair i, j in proportion to the fraction of neighborhoods of i and j that overlap, Adamic-Adar similarity is an absolute measure. It assigns to each common neighbor an inverse log weight, and therefore it discounts connections that are very weak, such as a mutual connection to a hub of very high degree. The Adamic-Adar similarity measure is defined as

$$S_{ij}^{\text{AA}} = \sum_{k \in \hat{N}_i \cap \hat{N}_j} \frac{1}{\log |\hat{N}_k|}$$

when DIRECTION=UNDIRECTED, or

$$S_{ij}^{\text{AA}} = \sum_{k \in \hat{N}_i^{\text{out}} \cap \hat{N}_j^{\text{out}}} \frac{1}{\log |\hat{N}_k^{\text{in}}|}$$

when DIRECTION=DIRECTED.

Another way to compare nodes in the graph is to first embed them in a d -dimensional vector space in such a way that the network structure is preserved. The method that is used here was introduced in Tang et al. (2015); it is designed to produce node embeddings that capture the first- or second-order proximity between nodes as proximity between corresponding vector-space embeddings. The first-order proximity in a graph can be represented by w_{ij} , the sum of weights of links from i to j . The second-order proximity between a pair of nodes (i, j) in a graph is the similarity between their neighborhood structures. If the first-order proximity of i to all other nodes is represented as a vector $\mathbf{p}_i = (w_{i1}, \dots, w_{i|N|})$, then the second-order proximity between i and j is determined by the similarity between $\mathbf{p}_i$ and $\mathbf{p}_j$. The second-order proximity assumes that the nodes that share many connections to other nodes are similar to each other. In the method of Tang et al. (2015), a d -dimensional vector $\mathbf{u}_i$ is calculated for each node i by minimizing a corresponding objective function. In the case of first-order proximity, the objective function is

$$O_1 = - \sum_{e \in E} w_e \log p_1(\text{from}(e), \text{to}(e))$$

where $\text{from}(e)$ represents the *from* node of link e , $\text{to}(e)$ represents the *to* node of link e , and p_1 is the joint probability of a link between nodes i and j , which is modeled as

$$p_1(i, j) = \frac{1}{1 + \exp(-\mathbf{u}_i \cdot \mathbf{u}_j)}$$

For second-order proximity, the d -dimensional vector embeddings $\mathbf{u}_i$ and the d -dimensional context embeddings $\mathbf{v}_i$ are calculated by minimization of the objective function,

$$O_2 = - \sum_{e \in E} w_e \log p_2(\text{to}(e) | \text{from}(e))$$

where

$$p_2(j | i) = \frac{\exp(\mathbf{v}_j \cdot \mathbf{u}_i)}{\sum_{k \in N} \exp(\mathbf{v}_k \cdot \mathbf{u}_i)}$$

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

If the stochastic gradient descent updates cease to make any significant change to the embedding vector space, the algorithm has converged and can terminate before reaching the number of samples that is specified in the NSAMPLES= option. The tolerance for this convergence termination criterion can be controlled via the CONVERGENCE THRESHOLD= option. In addition, the convergence values, which represent the rate of change of embedding vectors, are tabulated over time and are written to the OUTCONVERGENCE= data table.

After the embeddings for each node in the graph are calculated, the vector similarity measure between a pair of nodes i and j is calculated as a function of the angle between their embedding vectors,

$$S_{ij}^V = \frac{\cos(\theta_{\mathbf{u}_i \mathbf{u}_j}) + 1}{2}$$

where $\theta_{\mathbf{u}_i \mathbf{u}_j}$ is the angle between $\mathbf{u}_i$ and $\mathbf{u}_j$.

By default, PROC NETWORK finds the node similarity for all pairs of nodes in the input graph. That is, it finds the node similarity for each possible combination of source nodes and sink nodes. Alternatively, you can use the SOURCE= option in the NODESIMILARITY statement to fix a particular source node and find all node similarity scores from the fixed source node to all possible sink nodes. Conversely, by using the SINK= option in the NODESIMILARITY statement, you can fix a sink node and find all node similarity scores from all possible source nodes to the fixed sink node. By using both options together, you can calculate all similarity scores for a specific source-sink pair. In addition, you can use the NODESSUBSET= option in the PROC NETWORK statement to define a list of source-sink pairs to process, as described in the section “Nodes Subset Input Data” on page 67. The following section provides an example of finding the node similarity for all source-sink pairs. You can find additional examples of how to define the specific source-sink pairs of interest in the section “Shortest Path” on page 226.

Output Data Tables

The node similarity algorithm populates up to three output data tables. The output data table that you specify in the OUTSIMILARITY= option in the NODESIMILARITY statement contains the node similarity scores for each source-sink pair. The output data table that you specify in the OUTNODES= option in the PROC NETWORK statement option contains the vector representations of the nodes when you use the VECTOR= or EMBED= option. If you specify VECTOR=TRUE, the pairwise vector similarity scores are calculated in addition to the vector representations. To request the computation of node embeddings without computing pairwise vector node similarity scores, you can specify the EMBED=TRUE option without specifying the VECTOR=TRUE option. The output data table that you specify in the OUTCONVERGENCE= option in the NODESIMILARITY statement contains the convergence curves for vector embeddings training.

OUTSIMILARITY= Data Table

By default, the OUTSIMILARITY= data table contains the similarity scores for each node pair. Unless you request particular sources or sinks, the output table lists all pairs whose source node label is less than or equal to the sink node label. For large graphs and a large requested number of source-sink pairs, this output data table can be extremely large. Generating the output can sometimes take longer than computing the similarity. When you are running on multiple machines, this output data table is a distributed table. The only restriction is the total available cache disk space that is enabled by your configuration, as described in *SAS Cloud Analytic Services: User's Guide*. If you specify SPARSE=TRUE, the output data table excludes node pairs that have no common neighbors or whose source node label is equal to the sink node label. By specifying the SPARSE=TRUE option, you can greatly reduce the size of this output data table. However, you cannot specify SPARSE=TRUE in combination with the VECTOR=TRUE option, because vector node similarity is inherently dense.

The OUTSIMILARITY= data table contains the following columns:

- source: the source node label of this pair
- sink: the sink node label of this pair
- link: the flag indicating whether this pair exists in the LINKS= data table (1 if exists, 0 otherwise)
- order: the ranking of each pair ordered by decreasing similarity score, if you specify the TOPK= or BOTTOMK= option

- jaccard: the Jaccard similarity score, if you specify JACCARD=TRUE
- dice: the Dice similarity score, if you specify DICE=TRUE
- commonNeighbors: the common neighbors similarity score, if you specify COMMONNEIGHBORS=TRUE
- cosine: the cosine similarity score, if you specify COSINE=TRUE
- adamicAdar: the Adamic-Adar similarity score, if you specify ADAMICADAR=TRUE
- vector: the vector similarity score, if you specify VECTOR=TRUE

OUTNODES= Data Table

When VECTOR=TRUE, the output table that is specified in the OUTNODES= option contains the vector representations of each node in the input graph. The vector embedding columns are defined as `vec_1` through `vec_d`, where d is specified by the NDIMENSIONS= option in the NODESIMILARITY statement. When PROXIMITYORDER=SECOND, the context embedding columns, defined as `ctx_1` through `ctx_d`, are also written to this output table.

OUTCONVERGENCE= Data Table

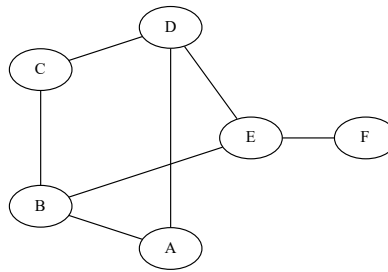
The OUTCONVERGENCE= data table contains the convergence curves for vector embeddings training. When this table has been specified, the convergence value of the vector embeddings training algorithm is periodically written to the table. In multiple machine processing, each machine trains its own embedding vectors and writes its convergence values independently. In order to limit the size of output table to no more than 10,000 observations per processing machine, the algorithm reports only at certain sample intervals. For example, if you specify NSAMPLES=8,000,000, the algorithm might report only every 1,000th sample.

The OUTCONVERGENCE= data table contains the following columns:

- worker: the identifier for the particular machine that reports the convergence value
- sample: the number of samples that have been completed when the convergence value is reported
- convergence: the reported convergence value

Node Similarity in an Undirected Graph

This section provides a simple example of calculation of node similarity in the undirected graph G that is shown in [Figure 2.97](#).

Figure 2.97 Undirected Graph G 

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

```

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

```

The following statements find the node pairs that have the top 21 vector node similarity scores:

```

proc network
    links          = mylib.LinkSetIn;
    nodeSimilarity
        vector      = true
        nDimensions = 10
        proximityOrder = second
        topK         = 21
        orderBy      = vector
        outSimilarity = mylib.NodeSim;
run;

```

The output data table `mylib.NodeSim`, as shown in [Figure 2.98](#), contains the pairs of nodes that are ranked highest according to vector node similarity.

Figure 2.98 Node Pairs ranked by Second Order Vector Similarity

source	sink	link	order	vector	jaccard
A	A	0	1	1.00000	1.00000
B	B	0	2	1.00000	1.00000
C	C	0	3	1.00000	1.00000
D	D	0	4	1.00000	1.00000
E	E	0	5	1.00000	1.00000
F	F	0	6	1.00000	1.00000
C	E	0	7	0.77859	0.66667
A	C	0	8	0.76196	1.00000
B	F	0	9	0.62101	0.33333
B	D	0	10	0.60832	1.00000
E	F	1	11	0.60548	0.00000
C	D	1	12	0.57327	0.00000
B	C	1	13	0.51978	0.00000
A	E	0	14	0.51242	0.66667
A	F	0	15	0.48257	0.00000
C	F	0	16	0.46406	0.00000
A	B	1	17	0.45554	0.00000
B	E	1	18	0.37221	0.00000
A	D	1	19	0.37144	0.00000
D	E	1	20	0.33226	0.00000
D	F	0	21	0.26633	0.33333

This example illustrates the intuition behind node similarity scores. The nodes A and C have the same set of nearest neighbors (see Figure 2.97), B and D, which results in the Jaccard similarity score of 1 and a vector similarity score very close to 1; thus the (A,C) pair is ranked first in the output. The pair (B,F) has lower Jaccard and vector similarity scores relative to (A,C). Lower similarity scores are reasonable in this cases because B and F share only one neighbor out of their three total neighbors.

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

```
proc network
  links          = mylib.LinkSetIn
  outNodes       = mylib.NodeVecs;
  nodeSimilarity
    jaccard      = false
    vector       = true
    nDimensions  = 10
    proximityOrder = first
    topK         = 21
    outSimilarity = mylib.NodeSim
    nSamples     = 10000;
run;
```

The output data table mylib.NodeSim, as shown in Figure 2.99, contains the pairs of nodes and similarity scores for the first-order proximity. The example also demonstrates the intuition behind similarity that is based on first-order proximity and how it is different from Jaccard-based proximity and the vector for the second-order proximity. Intuitively, the first-order vector similarity score between the nodes that are directly connected is higher than the first-order vector similarity score between the nodes that are separated by a

longer path in a graph, whereas the Jaccard similarity and second-order vector similarity scores are higher between the nodes that share more neighbors. The first-order vector similarity is more closely related to results that are produced by label propagation community detection algorithms, in the sense that the pairs of nodes that have high first-order vector similarity scores between them can be placed in one community.

Figure 2.99 Node Pairs ranked by First Order Vector Similarity

source	sink	link	order	vector
A	A	0	1	1.00000
B	B	0	2	1.00000
C	C	0	3	1.00000
D	D	0	4	1.00000
E	E	0	5	1.00000
F	F	0	6	1.00000
E	F	1	7	0.92055
B	D	0	8	0.89666
C	D	1	9	0.88843
B	C	1	10	0.88664
A	B	1	11	0.85652
A	D	1	12	0.83266
B	E	1	13	0.79937
D	E	1	14	0.79041
A	E	0	15	0.69844
C	E	0	16	0.65832
A	C	0	17	0.63142
D	F	0	18	0.57707
B	F	0	19	0.57462
A	F	0	20	0.48810
C	F	0	21	0.47847

The output data table `mylib.NodeVecs` contains node-embedding vectors, shown in [Figure 2.100](#).

Figure 2.100 Node-Embedding Vectors

node	vec_1	vec_2	vec_3	vec_4	vec_5	vec_6	vec_7	vec_8	vec_9	vec_10
A	-0.19710	-0.38469	-0.27657	-0.38787	0.11955	-0.20777	-0.51566	-0.44830	-0.13436	-0.20935
B	-0.49498	-0.19376	-0.42566	0.02023	0.02954	0.18430	-0.58490	-0.29728	0.13032	-0.23138
C	-0.31226	-0.21749	-0.44807	0.16253	-0.22939	0.38317	-0.42860	0.16318	0.35491	-0.30379
D	-0.09487	-0.31820	-0.38099	-0.24722	-0.17666	0.13038	-0.67262	-0.09681	0.38597	-0.15657
E	-0.22454	0.20090	-0.08974	-0.05951	0.06625	0.07396	-0.62022	-0.44412	0.43843	0.33709
F	-0.14305	0.34788	0.12668	0.12927	0.03250	-0.14591	-0.34150	-0.32373	0.48543	0.58851

Warm Start for Vector Node Similarity

This section continues the example in the section “Node Similarity in an Undirected Graph” on page 156 and illustrates warm start processing for vector node similarity. You can resume training of the previously learned node embeddings by providing the previous output as the initialization of a subsequent call to PROC NETWORK. After producing the result shown in Figure 2.100, you can warm start the training of the node-embedding vectors by providing the output table that is specified in the OUTNODES= option from the previous run as the NODES= input option. In addition, you must use the EMBEDDINGS= option to specify the variables within the NODES= input table to be used for the vector embeddings.

The following statements resume the training and produce refined node embeddings:

```
proc network
  links          = mylib.LinkSetIn
  nodes          = mylib.NodeVecs
  outNodes       = mylib.NodeVecs2;
  nodeSimilarity
    jaccard      = false
    vector       = true
    nDimensions  = 10
    proximityOrder = first
    topK         = 21
    outSimilarity = mylib.NodeSim2
    nSamples     = 10000
    embeddings   = (vec_1-vec_10);
run;
```

Figure 2.101 Refined First Order Vector Similarity Scores after Warm Start

source	sink	link	order	vector
A	A	0	1	1.00000
B	B	0	2	1.00000
C	C	0	3	1.00000
D	D	0	4	1.00000
E	E	0	5	1.00000
F	F	0	6	1.00000
E	F	1	7	0.93355
A	B	1	8	0.85003
C	D	1	9	0.84375
A	D	1	10	0.83993
B	C	1	11	0.82293
B	E	1	12	0.81907
B	D	0	13	0.81035
D	E	1	14	0.77810
A	E	0	15	0.65711
A	C	0	16	0.64834
C	E	0	17	0.63155
B	F	0	18	0.62078
D	F	0	19	0.57353
A	F	0	20	0.44830
C	F	0	21	0.44332

The new similarity scores in `mylib.NodeSim2`, as shown in [Figure 2.101](#), are based on the refined node embeddings that are in the data table `mylib.NodeVecs2`. [Figure 2.102](#) shows the refined node embeddings.

Figure 2.102 Refined Node-Embedding Vectors after Warm Start

node	vec_1	vec_2	vec_3	vec_4	vec_5	vec_6	vec_7	vec_8	vec_9	vec_10
A	-0.18436	-0.41537	-0.30140	-0.39302	0.09739	-0.17285	-0.50923	-0.40402	-0.13839	-0.25905
B	-0.55210	-0.12434	-0.38498	0.06085	0.09233	0.14987	-0.56194	-0.37868	0.08927	-0.17228
C	-0.28980	-0.23347	-0.44923	0.15612	-0.23854	0.39641	-0.39910	0.19715	0.33376	-0.33426
D	-0.00629	-0.32079	-0.34184	-0.31514	-0.18833	0.09409	-0.67127	-0.09145	0.40476	-0.11735
E	-0.24929	0.20076	-0.08939	-0.01119	0.04952	0.04527	-0.60522	-0.42347	0.46139	0.35558
F	-0.15244	0.37843	0.13235	0.13185	0.05622	-0.10786	-0.33469	-0.34178	0.46337	0.58337

Vector Node Similarity Convergence

This section demonstrates the convergence options for vector node similarity.

This example uses an undirected graph that is represented by the links data table, `mylib.LinkSetIn`, which is created by the following DATA step:

```
data mylib.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 compute vector node embeddings and plot the convergence curves for each worker:

```
proc network
    links                = mylib.LinkSetIn;
    nodeSimilarity
        jaccard          = false
        vector           = true
        convergenceThreshold = 0.025
        outConvergence   = mylib.OutConv1;
run;
```

The output data table `mylib.OutConv1` contains the convergence curve for each worker, as shown in [Figure 2.103](#). The training of the vector node embeddings halts after the desired convergence threshold of 0.025 is reached.

Figure 2.103 Convergence Output Table

sample	worker	convergence
10	0	0.78430
20	0	0.58833
30	0	0.44144
40	0	0.33133
50	0	0.24877
230	0	0.03850
240	0	0.03480
250	0	0.03142
260	0	0.02840
270	0	0.02564
280	0	0.02317
10	1	0.78172
20	1	0.58637
30	1	0.43993
40	1	0.33020
50	1	0.24798
230	1	0.03908
240	1	0.03534
250	1	0.03204
260	1	0.02901
270	1	0.02631
280	1	0.02381

sample	worker	convergence
10	2	0.78479
20	2	0.58872
30	2	0.44173
40	2	0.33151
50	2	0.24886
230	2	0.03923
240	2	0.03549
250	2	0.03217
260	2	0.02914
270	2	0.02640
280	2	0.02396
10	3	0.77974
20	3	0.58492
30	3	0.43896
40	3	0.32955
50	3	0.24753
230	3	0.03889
240	3	0.03516
250	3	0.03183
260	3	0.02886
270	3	0.02613
280	3	0.02369

sample	worker	convergence
10	4	0.78425
20	4	0.58841
30	4	0.44165
40	4	0.33147
50	4	0.24889
230	4	0.03927
240	4	0.03556
250	4	0.03223
260	4	0.02921
270	4	0.02647
280	4	0.02396
10	5	0.78281
20	5	0.58719
30	5	0.44051
40	5	0.33048
50	5	0.24808
230	5	0.03823
240	5	0.03453
250	5	0.03118
260	5	0.02817
270	5	0.02540
280	5	0.02296

sample	worker	convergence
10	6	0.78375
20	6	0.58796
30	6	0.44112
40	6	0.33102
50	6	0.24859
230	6	0.03868
240	6	0.03498
250	6	0.03159
260	6	0.02861
270	6	0.02594
280	6	0.02351
10	7	0.77993
20	7	0.58513
30	7	0.43898
40	7	0.32946
50	7	0.24744
230	7	0.03869
240	7	0.03503
250	7	0.03164
260	7	0.02870
270	7	0.02592
280	7	0.02346

Figure 2.103 *continued*

sample	worker	convergence
10	8	0.78087
20	8	0.58578
30	8	0.43942
40	8	0.32976
50	8	0.24755
230	8	0.03864
240	8	0.03485
250	8	0.03148
260	8	0.02850
270	8	0.02575
280	8	0.02326
10	9	0.78447
20	9	0.58840
30	9	0.44150
40	9	0.33132
50	9	0.24865
230	9	0.03870
240	9	0.03492
250	9	0.03154
260	9	0.02856
270	9	0.02584
280	9	0.02340

sample	worker	convergence
10	12	0.78397
20	12	0.58828
30	12	0.44143
40	12	0.33144
50	12	0.24909
230	12	0.03918
240	12	0.03544
250	12	0.03201
260	12	0.02899
270	12	0.02625
280	12	0.02384
10	13	0.78086
20	13	0.58578
30	13	0.43945
40	13	0.32994
50	13	0.24771
230	13	0.03899
240	13	0.03529
250	13	0.03200
260	13	0.02903
270	13	0.02628
280	13	0.02381

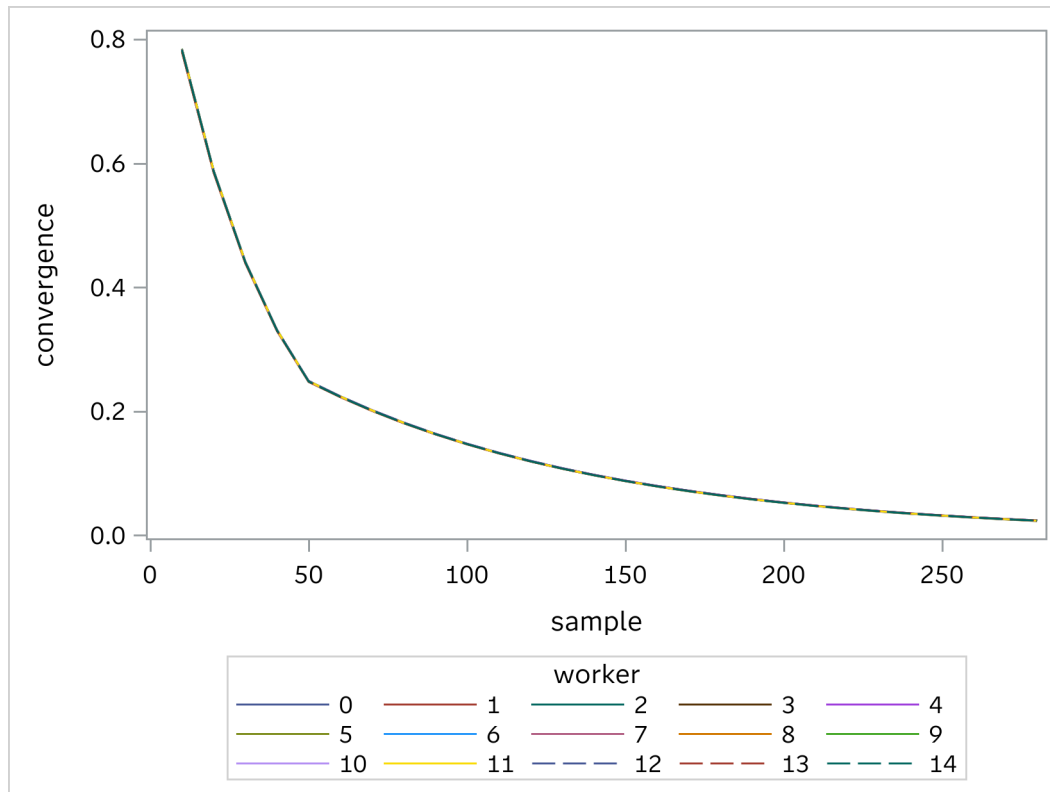
sample	worker	convergence
10	10	0.78318
20	10	0.58750
30	10	0.44074
40	10	0.33071
50	10	0.24826
230	10	0.03847
240	10	0.03478
250	10	0.03145
260	10	0.02843
270	10	0.02573
280	10	0.02331
10	11	0.78444
20	11	0.58842
30	11	0.44144
40	11	0.33130
50	11	0.24870
230	11	0.03881
240	11	0.03508
250	11	0.03178
260	11	0.02879
270	11	0.02609
280	11	0.02362

sample	worker	convergence
10	14	0.78247
20	14	0.58696
30	14	0.44039
40	14	0.33042
50	14	0.24795
230	14	0.03872
240	14	0.03497
250	14	0.03175
260	14	0.02868
270	14	0.02599
280	14	0.02356

The following statements produce a plot of the convergence curves from each processing worker:

```
proc sgplot data=mylib.OutConv1;
  series x=sample y=convergence / group=worker;
run;
```

Figure 2.104 Node-Embedding Vector Convergence Curves



Each plotted series represents a different worker, each of which trains embeddings independently and in parallel. The output is displayed in Figure 2.104.

Path Enumeration

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

In PROC NETWORK, you can find the elementary paths of an input graph by specifying the PATH statement. The options for this statement are described in the section “PATH Statement” on page 50.

By default, PROC NETWORK finds paths for all pairs of nodes in the input graph. That is, it finds all paths for each possible combination of source nodes and sink nodes. Alternatively, you can use the SOURCE= option to fix a particular source node and find all paths from the fixed source node to all possible sink nodes. Conversely, by using the SINK= option, you can fix a sink node and find all paths from all possible source nodes to the fixed sink node. By using both options together, you can request all paths for a specific

source-sink pair. In addition, you can use the `NODESSUBSET=` option to define a list of source-sink pairs to process, as described in the section “[Nodes Subset Input Data](#)” on page 67. The following section provides an example of how to use one of these options. Additional examples that show how to define the source-sink pairs of interest are found in the section “[Shortest Path](#)” on page 226.

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

Output Data Tables

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

OUTPATHSLINKS= Data Table

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

The `OUTPATHSLINKS=` data table contains the following columns:

- `source`: the source node label of this path
- `sink`: the sink node label of this path
- `path`: for this source-sink pair, the path identifier of this path
- `order`: for this source-sink pair, the order of this link in this path
- `from`: the *from* node label of this link in this path
- `to`: the *to* node label of this link in this path
- `weight`: the weight of this link in this path
- `column`: the auxiliary weight of this link (if the `AUXWEIGHT=column` is defined in the `LINKSVAR` statement)

OUTPATHSNODES= Data Table

The `OUTPATHSNODES=` data table contains the nodes present in each path. This output data table can also be extremely large. This output data table is a distributed table when you are running on multiple machines.

The `OUTPATHSNODES=` data table contains the following columns:

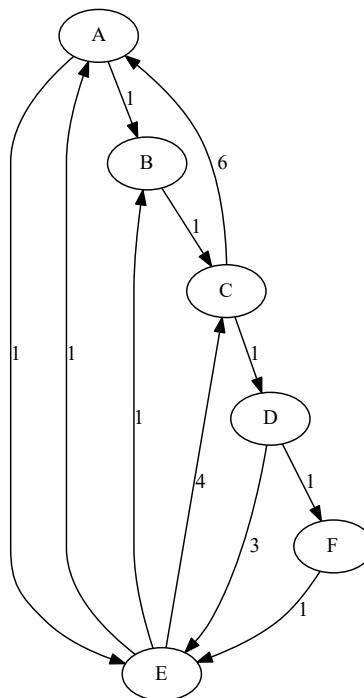
- `source`: the source node label of this path
- `sink`: the sink node label of this path

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

Path Enumeration for One Source-Sink Pair

This section provides a simple example of using the path enumeration algorithm on the directed graph G shown in Figure 2.105 to find all paths between one source-sink pair by using the SOURCE= and SINK= options.

Figure 2.105 Directed Graph G



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

```

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

```

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

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

The output data table mylib.PathLinks contains the links of the three paths from *D* to *A* whose path link weight is less than or equal to 10, as shown in [Figure 2.106](#).

Figure 2.106 Links for All (Short) Paths in a Directed Graph

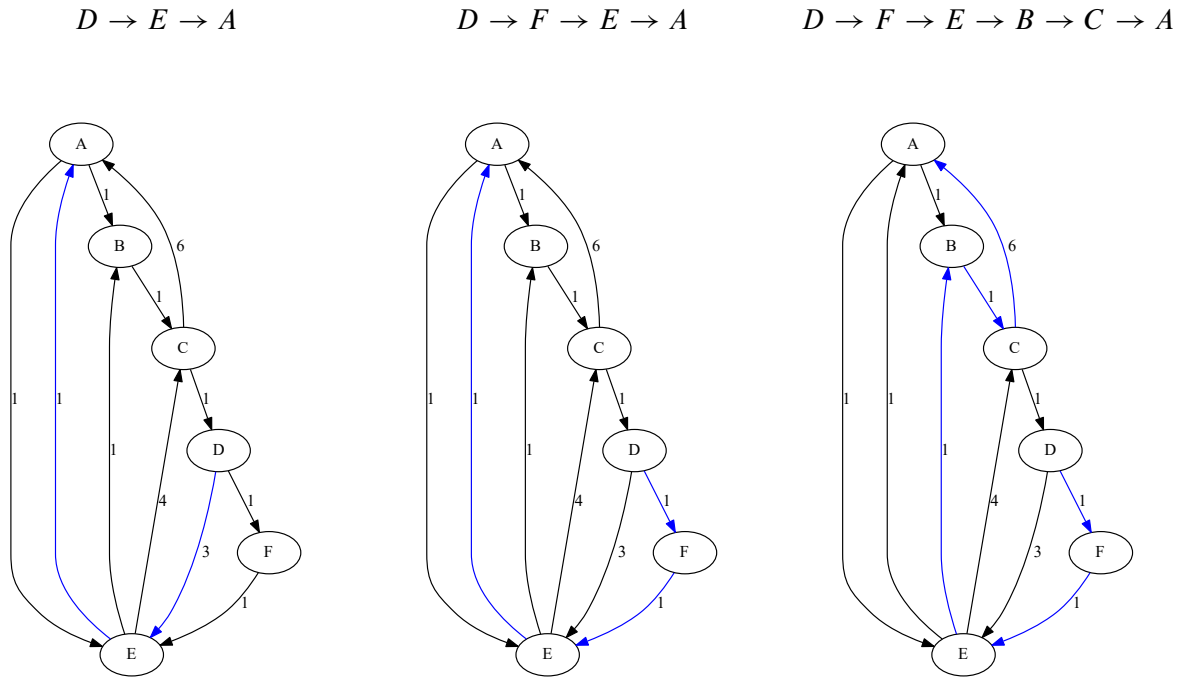
source	sink	path	order	from	to	weight
D	A	1	1	D	E	3
D	A	1	2	E	A	1
D	A	2	1	D	F	1
D	A	2	2	F	E	1
D	A	2	3	E	A	1
D	A	3	1	D	F	1
D	A	3	2	F	E	1
D	A	3	3	E	B	1
D	A	3	4	B	C	1
D	A	3	5	C	A	6

The output data table mylib.PathNodes contains the nodes of the three paths, as shown in [Figure 2.107](#).

Figure 2.107 Nodes for All (Short) Paths in a Directed Graph

source	sink	path	order	node
D	A	1	1	D
D	A	1	2	E
D	A	1	3	A
D	A	2	1	D
D	A	2	2	F
D	A	2	3	E
D	A	2	4	A
D	A	3	1	D
D	A	3	2	F
D	A	3	3	E
D	A	3	4	B
D	A	3	5	C
D	A	3	6	A

The three (short) paths are shown graphically in [Figure 2.108](#).

Figure 2.108 Paths

Pattern Matching

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

Subgraph isomorphism and pattern matching have applications in many areas, including social network analysis, fraud detection, pattern recognition, data mining, chemistry, and biology. See, for example, Aggarwal and Wang (2010); Gallagher (2006); Conte et al. (2004); Shasha, Wang, and Giugno (2002).

In PROC NETWORK, you can find pattern matches by using the PATTERNMATCH statement. The options for this statement are described in the section “[PATTERNMATCH Statement](#)” on page 51. The search algorithm is a multithreaded implementation that is based conceptually on the algorithms proposed in Saltz (2013).

The query graph is specified in the [LINKSQUERY=](#) option or the [NODESQUERY=](#) option (or both) in the PROC NETWORK statement. The specification of the query graph and its attributes works the same way that it does for the main graph, which is described in more detail in the section “[Graph Input and Output Data](#)” on page 61. One exception is the concept of a *wildcard* when an attribute’s value is missing. A missing value in the query graph indicates that the attribute value in the matching subgraph can be any value. An example of this is shown in “[Example 2.15: Pattern Matching in a Social Network](#)” on page 312.

Output Data Tables

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

OUTMATCHGRAPHLINKS= Data Table

The OUTMATCHGRAPHLINKS= data table describes the links in the [induced subgraph](#) of matches. The induced subgraph is constructed from the union of the set of nodes in all matches for each query key (if QUERYKEYAGGREGATE=FALSE) or across all query keys (if QUERYAGGREGATE=TRUE).

- queryKey: the query key (if you specify the QUERYKEY= option and QUERYKEYAGGREGATE=FALSE)
- from: the *from* node label for each link in the main graph that is in the induced subgraph of matches
- to: the *to* node label for each link in the main graph that is in the induced subgraph of matches

OUTMATCHGRAPHNODES= Data Table

The OUTMATCHGRAPHNODES= data table describes the nodes in the [induced subgraph](#) of matches. The induced subgraph is constructed from the union of the set of nodes in all matches for each query key (if QUERYKEYAGGREGATE=FALSE) or across all query keys (if QUERYAGGREGATE=TRUE).

- queryKey: the query key (if you specify the QUERYKEY= option and QUERYKEYAGGREGATE=FALSE)
- node: node label for each node in the main graph that is in the induced subgraph of matches

OUTMATCHLINKS= Data Table

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

- queryKey: the query key (if you specify the QUERYKEY= option)
- match: match identifier
- from: the *from* node label for each link in the main graph
- to: the *to* node label for each link in the main graph

OUTMATCHNODES= Data Table

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

- queryKey: the query key (if you specify the QUERYKEY= option)
- match: match identifier
- nodeQ: node label for each node in the query graph
- node: node label for each node in the main graph

OUTQUERYLINKS= Data Table

The OUTQUERYLINKS= data table describes the links for the generated queries. This data table contains the following columns:

- queryKey: the query key
- from: the *from* node label for each link in the query graph
- to: the *to* node label for each link in the query graph

OUTQUERYNODES= Data Table

The OUTQUERYNODES= data table describes the nodes for the generated queries. This data table contains the following columns:

- queryKey: the query key
- node: the node label for each node in the query graph

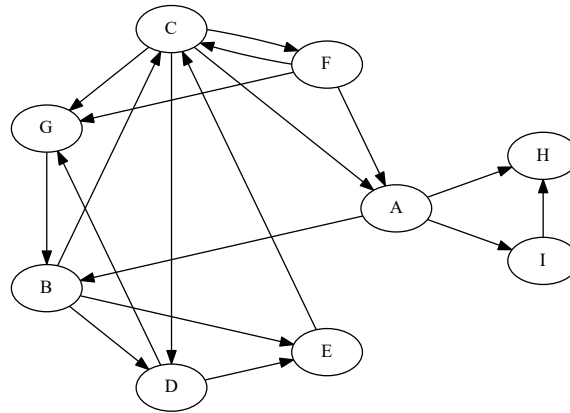
OUTSUMMARY= Data Table

The OUTSUMMARY= data table summarizes the executed queries, the number of matches that are found, and the elapsed time. This data table contains the following columns:

- queryKey: the query key (if the QUERYKEY= option is specified)
- nodes: the number of nodes in the query graph
- links: the number of links in the query graph
- matches: the number of matches that are found
- realTime: the elapsed time (in seconds) for the pattern match search

Pattern Matching of a Directed Graph

This section illustrates the use of the pattern matching algorithm on the directed graph G that is shown in Figure 2.109.

Figure 2.109 Directed Graph G 

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

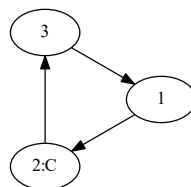
```

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

```

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

In this example, you want to find all subgraphs that are directed cycles of length 3 that pass through node C. The query graph Q that defines the pattern to search for is shown in Figure 2.110.

Figure 2.110 Query Graph Q 

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

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

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

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

The output data table `mylib.OutMatchNodes` now contains the mapping from nodes in the query graph to nodes in the main graph for each pattern match, as shown in [Figure 2.111](#).

Figure 2.111 Node Mappings for Pattern Matches

match	nodeQ	node	label
1	1	B	B
1	2	C	C
1	3	A	A
2	1	B	B
2	2	C	C
2	3	G	G
3	1	E	E
3	2	C	C
3	3	D	D

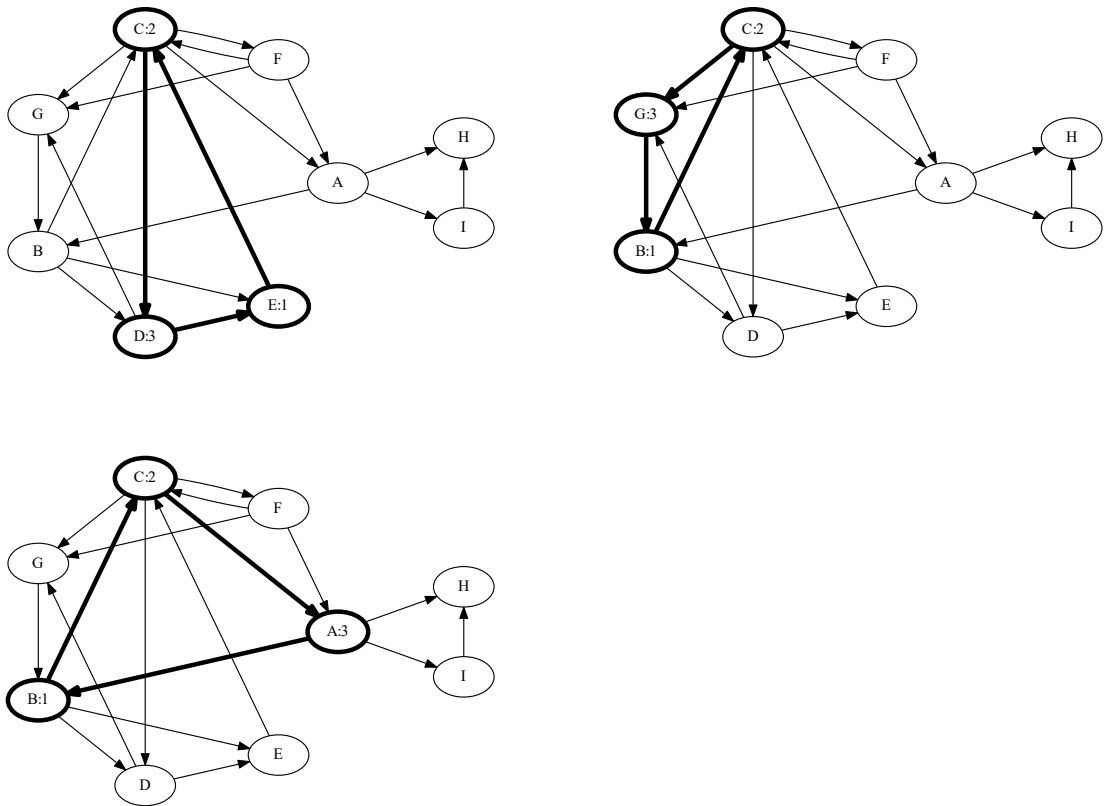
The output data table `mylib.OutMatchLinks` now contains the subgraphs for each pattern match, as shown in [Figure 2.112](#).

Figure 2.112 Subgraphs for Pattern Matches

match from to		
1	A	B
1	B	C
1	C	A
2	B	C
2	C	G
2	G	B
3	C	D
3	D	E
3	E	C

The results are displayed graphically in Figure 2.113.

Figure 2.113 Subgraphs



The output data table mylib.OutMatchGraphNodes now contains the nodes in the induced subgraph of matches, as shown in Figure 2.114.

Figure 2.114 Nodes of the Induced Subgraph of Matches

node	label
A	A
B	B
C	C
D	D
E	E
G	G

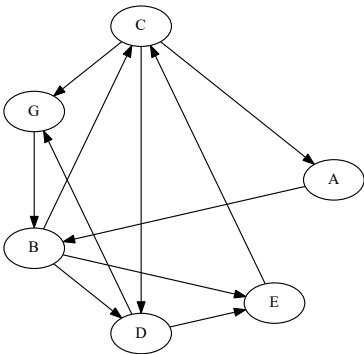
The output data table mylib.OutMatchGraphLinks now contains the links in the induced subgraph of matches, as shown in [Figure 2.115](#).

Figure 2.115 Links of the Induced Subgraph of Matches

from	to
A	B
B	C
B	D
B	E
C	A
C	D
C	G
D	E
D	G
E	C
G	B

The induced subgraph of matches is displayed graphically in [Figure 2.116](#).

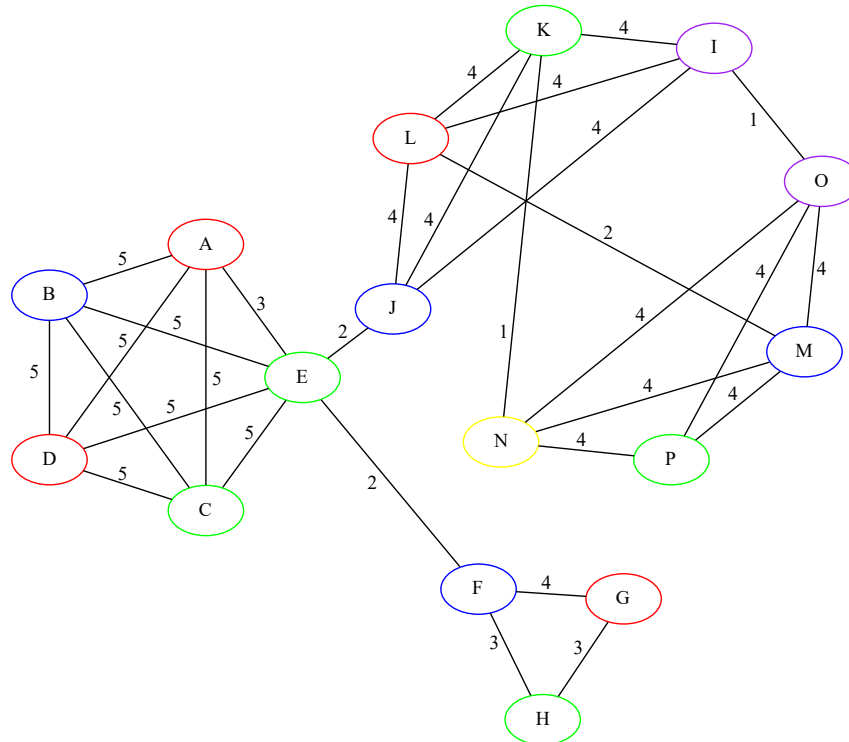
Figure 2.116 Induced Subgraph



Pattern Matching of an Undirected Graph

This section illustrates the use of the pattern matching algorithm on the undirected graph G that is shown in Figure 2.117.

Figure 2.117 Undirected Graph G



The undirected graph G has one link attribute (weight) and one node attribute (color). You can represent the graph by using the nodes data table, `mylib.Nodes`, and links data table, `mylib.Links`, that are created by the following DATA steps:

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

```

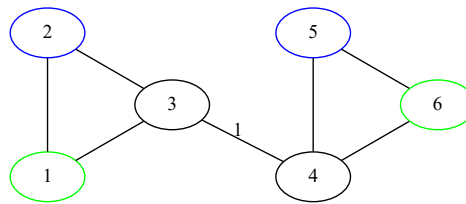
K L 4 I O 1 K N 1 L M 2 M N 4
M O 4 M P 4 N P 4 N O 4 P O 4
;

```

Each node has a node attribute color, and each link has a link attribute weight.

In this example, you want to find pairs of triangles (cliques of size 3) that are connected by a link whose weight is 1. In addition, you want the outer nodes of the triangle to have the colors blue and green. The query graph Q that defines the pattern to search for is shown in Figure 2.118.

Figure 2.118 Query Graph Q



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

```

data mylib.NodesQuery;
    input node color $;
    datalines;
1 green
2 blue
5 blue
6 green
;
data mylib.LinksQuery;
    input from to weight;
    datalines;
1 2 .
1 3 .
2 3 .
3 4 1
4 5 .
5 6 .
4 6 .
;

```

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

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


```

proc network
  nodes          = mylib.Nodes
  links          = mylib.Links
  nodesQuery     = mylib.NodesQuery
  linksQuery     = mylib.LinksQuery;
  nodesVar
    vars        = (color);
  linksVar
    vars        = (weight);
  nodesQueryVar
    vars        = (color);
  linksQueryVar
    vars        = (weight);
  patternMatch
    outMatchNodes = mylib.OutMatchNodes
    outMatchLinks = mylib.OutMatchLinks;
run;

```

The output data table `mylib.OutMatchNodes` now contains the mapping from nodes in the query graph to nodes in the main graph for each pattern match, as shown in [Figure 2.119](#).

Figure 2.119 Node Mappings for Pattern Matches

match	nodeQ	node	color
1	1	K	green
1	2	J	blue
1	3	I	purple
1	4	O	purple
1	5	M	blue
1	6	P	green
2	1	P	green
2	2	M	blue
2	3	O	purple
2	4	I	purple
2	5	J	blue
2	6	K	green

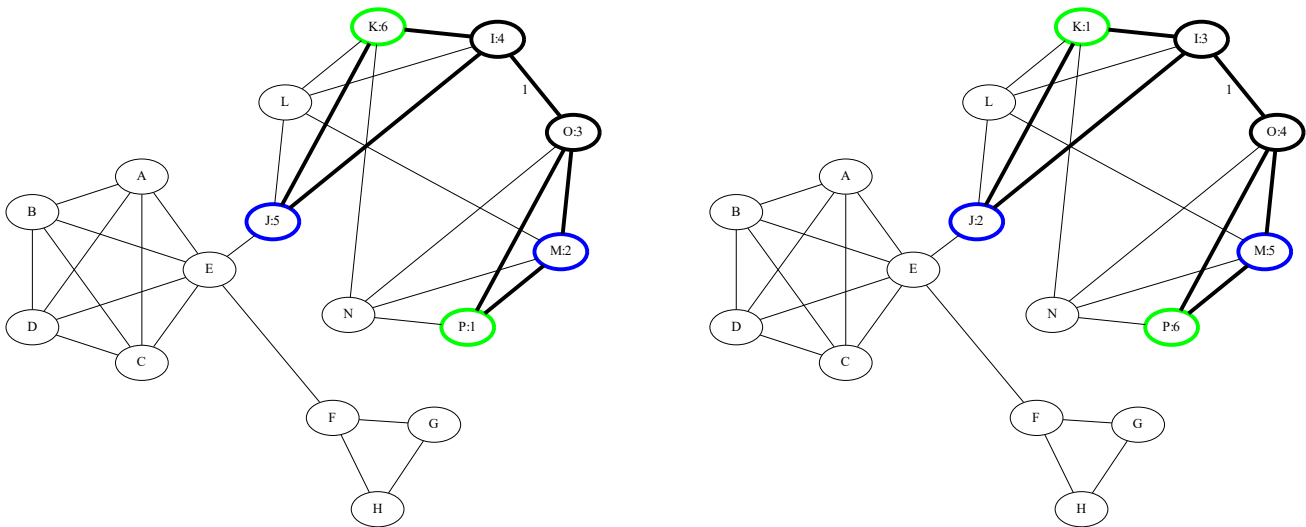
The output data table `mylib.OutMatchLinks` now contains the subgraphs for each pattern match, as shown in [Figure 2.120](#).

Figure 2.120 Subgraphs for Pattern Matches

match	from	to	weight
1	I	K	4
1	I	O	1
1	J	I	4
1	J	K	4
1	M	O	4
1	M	P	4
1	P	O	4
2	I	K	4
2	I	O	1
2	J	I	4
2	J	K	4
2	M	O	4
2	M	P	4
2	P	O	4

Only one set of links (one topology) matches the specified pattern. However, there are two isomorphic mappings. The results are displayed graphically in Figure 2.121.

Figure 2.121 Subgraphs



Using the SAS Function Compiler to Create Refined Query Filters

As is shown in the previous examples, using attributes in the query data tables returns subgraphs that match the specified values exactly in addition to matching the defined graph structure (topology).

You can further customize the query by using the SAS Function Compiler (FCMP), which provides greater flexibility than using query data tables provides. For a general overview of FCMP and examples of its use in creating user-defined functions, see the chapter “The FCMP Procedure” in *Base SAS Procedures Guide*. In PROC NETWORK, you can use FCMP to provide a set of Boolean functions, each of which, when associated

with a specific pattern match query, defines an additional condition that a subgraph from the main graph must satisfy in order to be considered a match. You can use FCMP syntax and programming statements in addition to the attributes of the query (or main) graph in defining such a condition. The flexibility that you have in defining the FCMP function permits both exact and inexact attribute matching for individual nodes and links, and these can be further tailored to specific nodes and links in the query graph. The additional ability to use functions that apply to pairs of nodes and links enables you to define conditions whose scope is more global than that of conditions on attributes of individual nodes and links. Refinement using pair filters is described in the section “[Using the SAS Function Compiler to Create Refined Query Pair Filters](#)” on page 181. Furthermore, you can define a filter globally on the set of candidate matching nodes and links. Refinement using a global match filter is described in the section “[Using the SAS Function Compiler to Create Refined Query Match Filters](#)” on page 185.

When you specify filters, you can use FCMP functions that you have defined in a library that is specified by the CMPLIB= system option, or you can use the CODE= option to define FCMP functions inline.

Arguments to the defined functions must either correspond to columns in the input data tables or define scalars. The expected format for defining a column argument in the function option is **table.column**, where **table** represents the appropriate table indicator as described in [Table 2.13](#) and **column** is the name of the column in the referenced table.

Table 2.13 FCMP Function Argument Table Indicators

Table	Table Indicators	Example
LINKS=	links or l	links.var specifies the var variable in the links table
LINKSQUERY=	linksQuery , linksQ , or lq	linksQuery.var specifies the var variable in the links query table
NODES=	nodes or n	nodes.var specifies the var variable in the nodes table
NODESQUERY=	nodesQuery , nodesQ , or nq	nodesQuery.var specifies the var variable in the nodes query table

You can specify FCMP functions and their associated arguments in the following options to define a node or link filter function:

- NODEFILTER= option, which specifies the FCMP function for a node filter
- LINKFILTER= option, which specifies the FCMP function for a link filter

During the search process, a node filter function is called for a particular mapping from some node in the query graph to some node in the main graph. The node filter function evaluates whether this node mapping is valid. A link filter function is called for a particular mapping from some link in the query graph to some link in the main graph. The link filter function evaluates whether this link mapping is valid.

Consider again the example in the section “[Pattern Matching of an Undirected Graph](#)” on page 175. Rather than providing the color variable in the mylib.NodesQuery table, the same refinement is accomplished using the following node filter function and associated function call:

```

function myNodeFilter(nodeQ, color $);
  if (nodeQ in (1,6)) then return (color='green');
  else if (nodeQ in (2,5)) then return (color='blue');
  else return (1);
endsub;
patternMatch
  nodeFilter = myNodeFilter(nodesQuery.node,nodes.color)

```

For query nodes 1, 2, 5, and 6, the function **myNodeFilter** checks the color attribute of a candidate node in the main graph. Query nodes 1 and 6 must map to a node in the main graph whose color is green. Query nodes 2 and 5 must map to a node in the main graph whose color is blue. For all other query nodes, the color of the mapped node in the main graph can take any value.

Similarly, rather than providing the weight variable in the **mylib.LinksQuery** table, the same refinement is accomplished using the following link filter function and associated function call:

```

function myLinkFilter(fromQ, toQ, weight);
  if (fromQ=3 and toQ=4) then return (weight=1);
  else return (1);
endsub;
patternMatch
  linkFilter = myLinkFilter(linksQuery.from,linksQuery.to,links.weight)

```

For query link {3,4}, the function **myLinkFilter** checks the weight attribute of a candidate link in the main graph. Query link {3,4} must map to a link in the main graph whose weight is 1. For all other query links, the weight of the mapped link in the main graph can take any value.

Pattern Matching of an Undirected Graph Using FCMP Filter Functions

This section shows how you can use the FCMP filter functions instead of defining attributes in the query data tables. Using attributes in the query data tables allows only for exact attribute matching. The filter functions extend the query capabilities to inexact attribute matching. Any method or mathematical operator that is supported by FCMP can be used in the definition of the filter function. For an example of using inexact attribute matching, see “[Example 2.15: Pattern Matching in a Social Network](#)” on page 312.

The following DATA steps create the same query tables as are created in the section “[Pattern Matching of an Undirected Graph](#)” on page 175 but with no additional attribute specifications.

```

data mylib.NodesQuery;
  input node @@;
  datalines;
1 2 5 6
;
data mylib.LinksQuery;
  input from to @@;
  datalines;
1 2 1 3 2 3 3 4 4 5
5 6 4 6
;

```

The following statements use PROC FCMP to specify the filter functions **myLinkFilter** and **myNodeFilter** and then set the CMPLIB to specify where to look for the compiled functions. For more information about PROC FCMP, see the chapter “FCMP Procedure” in *Base SAS Procedures Guide*.

```

proc fcmp casoutlib=casuser.functions.filters;
  function myNodeFilter(nodeQ, color $);
    if (nodeQ in (1,6)) then return (color='green');
    else if (nodeQ in (2,5)) then return (color='blue');
    else return (1);
  endsub;
  function myLinkFilter(fromQ, toQ, weight);
    if (fromQ=3 and toQ=4) then return (weight=1);
    else return (1);
  endsub;
quit;
proc fcmp setcascmplib="casuser.functions" getcascmplib; run;

```

Next, the following statements find all subgraphs that have the pattern that is specified by the query input data tables and the FCMP filter functions:

```

proc network
  nodes          = mylib.Nodes
  links          = mylib.Links
  nodesQuery     = mylib.NodesQuery
  linksQuery     = mylib.LinksQuery;
  nodesVar
    vars         = (color);
  linksVar
    vars         = (weight);
  patternMatch
    nodeFilter    = myNodeFilter(nodesQuery.node,nodes.color)
    linkFilter    = myLinkFilter(linksQuery.from,linksQuery.to,links.weight)
    outMatchNodes = mylib.OutMatchNodes
    outMatchLinks = mylib.OutMatchLinks;
run;

```

Using the SAS Function Compiler to Create Refined Query Pair Filters

Further refinement of queries can be done by providing functions that consider pairs of nodes (or links) and their associated mappings.

You can specify FCMP functions and their associated arguments in the following options to define a node-pair or link-pair filter function:

- **NODEPAIRFILTER=** option, which specifies the FCMP function for a filter that is based on a pair of nodes
- **LINKPAIRFILTER=** option, which specifies the FCMP function for a filter that is based on a pair of links

During the search process (or when the validity of a node mapping is considered), a node-pair filter function is called for a particular mapping of a pair of nodes in the query graph to a pair of nodes in the main graph. The node-pair filter function evaluates whether this node-pair mapping is valid. Similarly, when the subgraph is considered for a mapping, a link-pair filter function is called for each pair of links in the query graph to a pair of links in the main graph. The link-pair filter function evaluates whether this link-pair mapping is valid.

All arguments to the pair filter functions are arrays of length two (for the two elements of the pair). As in the filter functions, the arguments must correspond to columns in the input data tables or they must define scalars.

Consider again the example in the section “[Pattern Matching of an Undirected Graph](#)” on page 175. The following node-pair filter function enforces that the colors of the nodes that are mapped to query nodes 1 and 6 are the same and that the colors of the nodes that are mapped to query nodes 2 and 5 are the same:

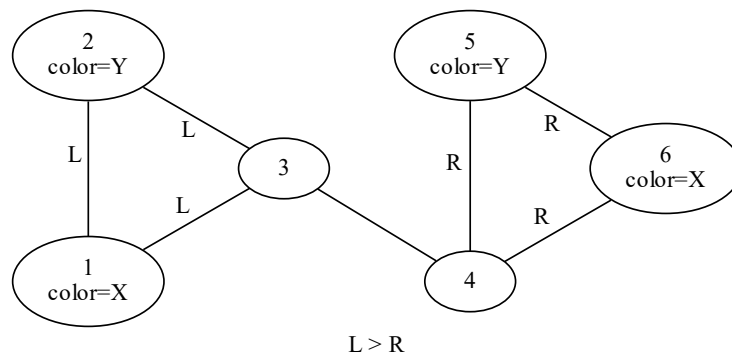
```
%macro nodePairFilterCode();
  function myNodePairFilter(nodeQ[*], color[*] $);
    if ((nodeQ[1]=1 and nodeQ[2]=6) or
        (nodeQ[1]=2 and nodeQ[2]=5)) then return (color[1]=color[2]);
    else return (1);
  endsub;
%mend nodePairFilterCode;
patternMatch
  nodePairFilter = myNodePairFilter(nodesQuery.node,nodes.color)
```

The following link-pair filter function enforces that the weights of the links that are mapped to the triangles (1,2,3 and 4,5,6) are inclusively equal and that the weights of the links of the left triangle are greater than the weights of the links of the right triangle:

```
%macro linkPairFilterCode();
  function myLinkPairFilter(fromQ[*], toQ[*], weight[*]);
    if ((fromQ[1]<=3 and toQ[1]<=3 and fromQ[2]<=3 and toQ[2]<=3) or
        (fromQ[1]>=4 and toQ[1]>=4 and fromQ[2]>=4 and toQ[2]>=4))
    then return (weight[1]=weight[2]);
    else
      if (fromQ[1]=1 and toQ[1]=3 and fromQ[2]=4 and toQ[2]=6)
      then return (weight[1]>weight[2]);
      else return (1);
    endsub;
%mend linkPairFilterCode;
patternMatch
  linkPairFilter = myLinkPairFilter(linksQuery.from,linksQuery.to,links.weight)
```

Pattern Matching of an Undirected Graph Using FCMP Pair Filter Functions

The query graph Q that defines the pattern to search for is shown in [Figure 2.122](#).

Figure 2.122 Query Graph Q 

The topology of the query graph Q can be represented using the nodes data table (mylib.NodesQuery) and links data table (mylib.LinksQuery) that are created by the following DATA steps:

```
data mylib.NodesQuery;
  input node @@;
  datalines;
1 2 5 6
;
data mylib.LinksQuery;
  input from to @@;
  datalines;
1 2 1 3 2 3 3 4 4 5
5 6 4 6
;
```

You can provide the FCMP code for the customized filter functions by using macro substitution and the CODE= option. The following statements find all subgraphs that have the pattern that is specified by the query input data tables and the FCMP pair filter functions:

```
proc network
  nodes          = mylib.Nodes
  links          = mylib.Links
  nodesQuery     = mylib.NodesQuery
  linksQuery     = mylib.LinksQuery;
  nodesVar
    vars         = (color);
  linksVar
    vars         = (weight);
  patternMatch
    code         =
    "
      %nodePairFilterCode()
      %linkPairFilterCode()
    "
    nodePairFilter = myNodePairFilter(nodesQuery.node,nodes.color)
```

```

linkPairFilter = myLinkPairFilter(linksQuery.from, linksQuery.to, links.weight)
outMatchNodes  = mylib.OutMatchNodes
outMatchLinks  = mylib.OutMatchLinks;

run;

```

The output data table `mylib.OutMatchNodes` now contains the mapping from nodes in the query graph to nodes in the main graph for each pattern match, as shown in [Figure 2.123](#).

Figure 2.123 Node Mappings for Pattern Matches

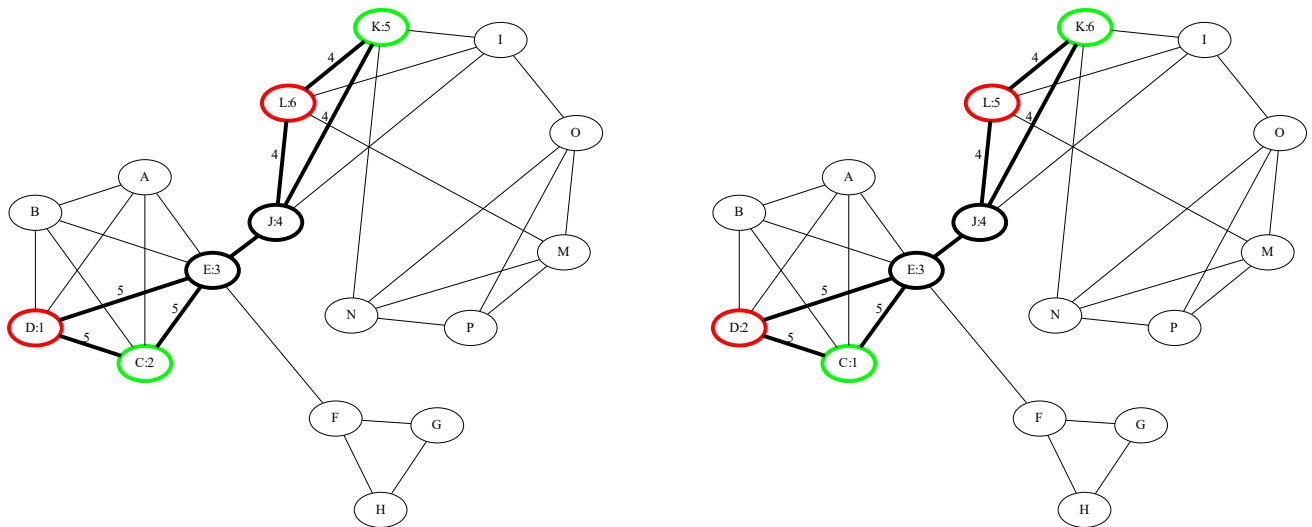
match	nodeQ	node	color
1	1	C	green
1	2	D	red
1	3	E	green
1	4	J	blue
1	5	L	red
1	6	K	green
2	1	D	red
2	2	C	green
2	3	E	green
2	4	J	blue
2	5	K	green
2	6	L	red

The output data table `mylib.OutMatchLinks` now contains the subgraphs for each pattern match, as shown in [Figure 2.124](#).

Figure 2.124 Subgraphs for Pattern Matches

match	from	to	weight
1	C	D	5
1	C	E	5
1	D	E	5
1	E	J	2
1	J	K	4
1	J	L	4
1	K	L	4
2	C	D	5
2	C	E	5
2	D	E	5
2	E	J	2
2	J	K	4
2	J	L	4
2	K	L	4

Only one set of links (one topology) matches the specified pattern. However, there are two isomorphic mappings. The results are displayed graphically in [Figure 2.125](#).

Figure 2.125 Subgraphs

For an example of using node and link-pair filters to identify value-added tax carousel fraud, see “[Example 2.16: Detection of Value-Added Tax Carousel Fraud](#)” on page 321.

Using the SAS Function Compiler to Create Refined Query Match Filters

You can do global refinement of queries by providing functions that consider any subset of nodes (or links or both) and their associated mappings.

The `MATCHFILTER=` option specifies the FCMP function for a filter that is based on a potential match (that is, any subset of nodes or links or both).

During the search process (or when the validity of a node mapping or subgraph is considered), a match filter function is called for a particular mapping of a subset of nodes (or links or both) in the query graph to a subset of nodes (or links or both) in the main graph. The match filter function evaluates whether this node (or link or both) mapping is valid.

All arguments to the match filter functions are arrays. As in the filter functions, the arguments must correspond to columns in the input data tables or they must define scalars. Arguments that are associated with links are arrays whose length is equal to the number of links in the query graph. Arguments that are associated with nodes are arrays whose length is equal to the number of nodes in the query graph.

Consider again the example in the section “[Pattern Matching of an Undirected Graph Using FCMP Pair Filter Functions](#)” on page 182. The following match filter function enforces that the sum of weights of the links of the left triangle (1,2,3) is at least 30% greater than the sum of the weights of the links of the right triangle (4,5,6):

```
%macro matchFilterCode();
  function myMatchFilter(fromQ[*], toQ[*], weight[*], limit);
    sumLeft  = 0;
    sumRight = 0;
    nLinks   = dim(weight);
```

```

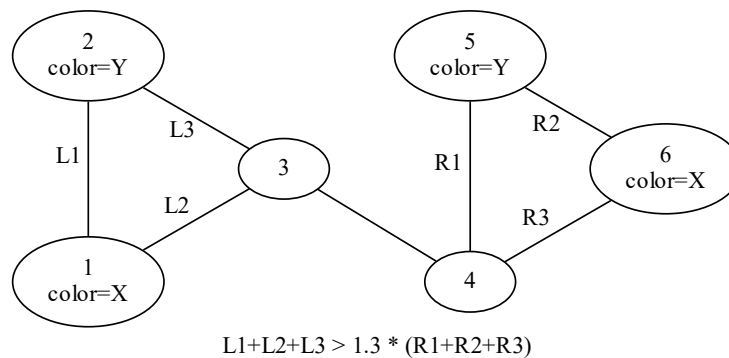
do i=1 to nLinks;
  if(fromQ[i]<=3 and toQ[i]<=3) then sumLeft = sumLeft + weight[i];
  if(fromQ[i]<=3 and toQ[i]<=3) then sumLeft = sumLeft + weight[i];
  if(fromQ[i]<=3 and toQ[i]<=3) then sumLeft = sumLeft + weight[i];
  if(fromQ[i]<=3 and toQ[i]<=3) then sumLeft = sumLeft + weight[i];
  if(fromQ[i]<=3 and toQ[i]<=3) then sumLeft = sumLeft + weight[i];
  else
  else
  else
  else
    if(fromQ[i]>=4 and toQ[i]>=4) then sumRight = sumRight + weight[i];
  end;
  return (sumLeft>(limit*sumRight));
endsub;
%mend matchFilterCode;
patternMatch
  matchFilter = myMatchFilter(linksQuery.from,linksQuery.to,links.weight,1.3)

```

Pattern Matching of an Undirected Graph Using FCMP Match Filter Functions

The query graph Q that defines the pattern to search for is shown in Figure 2.126.

Figure 2.126 Query Graph Q



The topology of the query graph Q can be represented using the nodes data table (mylib.NodesQuery) and links data table (mylib.LinksQuery) that are created by the following DATA steps:

```

data mylib.NodesQuery;
  input node @@;
  datalines;
1 2 5 6
;
data mylib.LinksQuery;
  input from to @@;
  datalines;
1 2 1 3 2 3 3 4 4 5

```

```
5 6 4 6
;
```

Next, the following statements find all subgraphs that have the pattern that is specified by the query input data tables and the FCMP filter functions:

```
proc network
  nodes          = mylib.Nodes
  links          = mylib.Links
  nodesQuery     = mylib.NodesQuery
  linksQuery     = mylib.LinksQuery;
  nodesVar
    vars        = (color);
  linksVar
    vars        = (weight);
  patternMatch
    code        =
    "
      %nodePairFilterCode()
      %matchFilterCode()
    "
    nodePairFilter = myNodePairFilter(nodesQuery.node,nodes.color)
    matchFilter    = myMatchFilter(linksQuery.from,linksQuery.to,links.weight,1.3)
    outMatchNodes  = mylib.OutMatchNodes
    outMatchLinks  = mylib.OutMatchLinks;
run;
```

The output data table mylib.OutMatchNodes now contains the mapping from nodes in the query graph to nodes in the main graph for each pattern match, as shown in [Figure 2.127](#).

Figure 2.127 Node Mappings for Pattern Matches

match	nodeQ	node	color
1	1	C	green
1	2	D	red
1	3	E	green
1	4	F	blue
1	5	G	red
1	6	H	green
2	1	D	red
2	2	C	green
2	3	E	green
2	4	F	blue
2	5	H	green
2	6	G	red

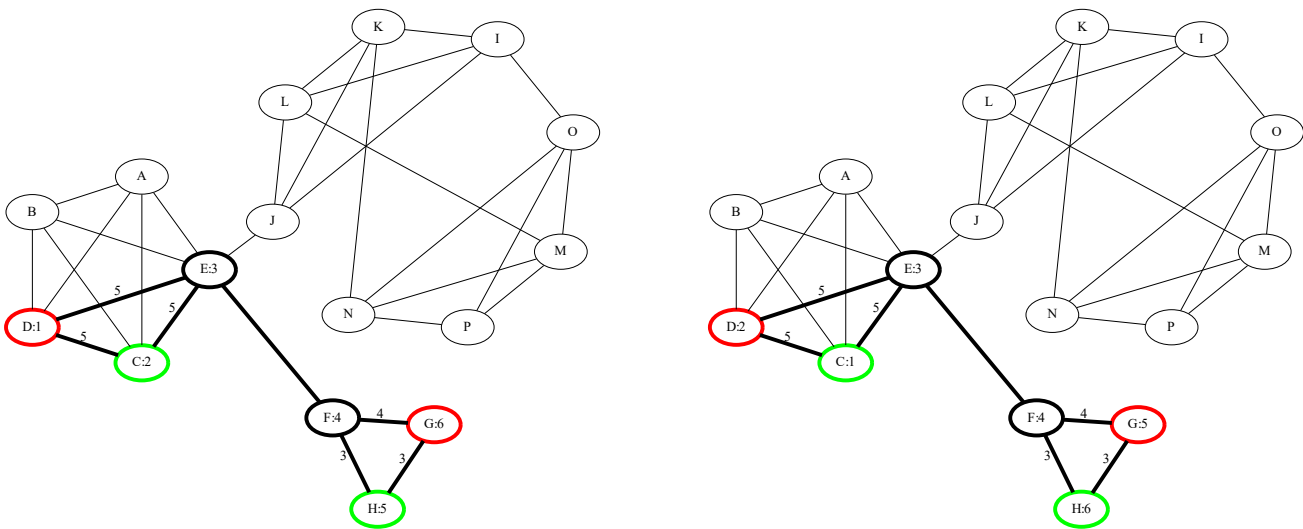
The output data table mylib.OutMatchLinks now contains the subgraphs for each pattern match, as shown in [Figure 2.128](#).

Figure 2.128 Subgraphs for Pattern Matches

match	from	to	weight
1	C	D	5
1	C	E	5
1	D	E	5
1	E	F	2
1	F	G	4
1	F	H	3
1	G	H	3
2	C	D	5
2	C	E	5
2	D	E	5
2	E	F	2
2	F	G	4
2	F	H	3
2	G	H	3

Only one set of links (one topology) matches the specified pattern. However, there are two isomorphic mappings. The results are displayed graphically in [Figure 2.129](#).

Figure 2.129 Subgraphs



For an example of using a match filter to identify value-added tax carousel fraud, see “[Example 2.16: Detection of Value-Added Tax Carousel Fraud](#)” on page 321.

Using a Node Pair Filter to Break Symmetry

In the example in the section “Pattern Matching of an Undirected Graph Using FCMP Match Filter Functions” on page 186, you might be interested in finding the distinct subgraphs, not all the isomorphic mappings. In this case, it is necessary to break the symmetry in the query graph, so that out of multiple isomorphic mappings, only one match is returned. To do this, you can add a condition to the node pair filter that breaks the symmetry by allowing only one of the isomorphic alternatives.

In the case of the query graph of Figure 2.126, relabeling the nodes 1, 2, 5, and 6 as 2, 1, 6, and 5, respectively, produces an isomorphism. To break the symmetry, you can add a clause requiring that the node mapped to 1 has a label in the main graph that comes lexicographically before that of the node mapped to 2.

To represent this constraint by using FCMP, you can add logic to the node pair filter function to ensure that the nodes mapped to 1 and 2 have node labels that are in lexicographic order. The following statements demonstrate the modified logic:

```
%macro myFilterCode();
  function myNodePairFilter(node[*] $, nodeQ[*], color[*] $);
    if(nodeQ[1]=1 and nodeQ[2]=2 and node[1]>node[2]) then return (0);
    if ((nodeQ[1]=1 and nodeQ[2]=6) or
        (nodeQ[1]=2 and nodeQ[2]=5)) then return (color[1]=color[2]);
    else return (1);
  endsub;
  function myMatchFilter(fromQ[*], toQ[*], weight[*], limit);
    sumLeft = 0;
    sumRight = 0;
    nLinks = dim(weight);
    do i=1 to nLinks;
      if(fromQ[i]<=3 and toQ[i]<=3) then sumLeft = sumLeft + weight[i];
      else
        if(fromQ[i]>=4 and toQ[i]>=4) then sumRight = sumRight + weight[i];
      end;
    end;
    return (sumLeft>(limit*sumRight));
  endsub;
%mend myFilterCode;
```

Next, the following statements find all subgraphs that have the pattern that is specified by the modified query input data tables and FCMP filter functions:

```
proc network
  nodes          = mylib.Nodes
  links          = mylib.Links
  nodesQuery     = mylib.NodesQuery
  linksQuery     = mylib.LinksQuery;
  nodesVar
    vars         = (color);
  linksVar
    vars         = (weight);
  patternMatch
    code         = "%myFilterCode()"
    nodePairFilter = myNodePairFilter(nodes.node, nodesQuery.node, nodes.color)
    matchFilter   = myMatchFilter(linksQuery.from, linksQuery.to, links.weight, 1.3)
    outMatchNodes = mylib.OutMatchNodes
    outMatchLinks = mylib.OutMatchLinks;
```

```
run;
```

The output data table `mylib.OutMatchNodes` now contains the mapping from nodes in the query graph to nodes in the main graph for each pattern match, as shown in [Figure 2.130](#).

Figure 2.130 Node Mappings for Pattern Matches

match	nodeQ	node	color
1	1	C	green
1	2	D	red
1	3	E	green
1	4	F	blue
1	5	G	red
1	6	H	green

The output data table `mylib.OutMatchLinks` now contains the subgraphs for the single match, as shown in [Figure 2.131](#).

Figure 2.131 Subgraphs for Pattern Matches

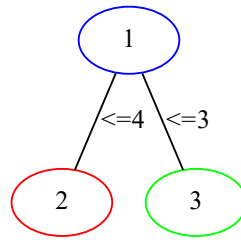
match	from	to	weight
1	C	D	5
1	C	E	5
1	D	E	5
1	E	F	2
1	F	G	4
1	F	H	3
1	G	H	3

Now, only one of the two isomorphic matches that are found in the section “[Pattern Matching of an Undirected Graph Using FCMP Match Filter Functions](#)” on page 186 is returned.

Specifying the Variables to Match

In some applications, you might want to consider an attribute that is associated with a query graph but not match its value exactly. By default, the variables to match (`VARSMATCH=`) are the same as the variables to consider (`VAR=`). The `VARSMATCH=` option in the `LINKSQUERYVAR` and `NODESQUERYVAR` statements enables you to specify the variables to match exactly.

Consider again the example in the section “[Pattern Matching of an Undirected Graph](#)” on page 175, which is displayed graphically in [Figure 2.117](#). The pattern that you want to find is a simple two-edge, three-node graph where the middle node has the color blue and the other two nodes have colors red and green. In addition, the link connecting nodes 1 and 2 should have weight at most 4 and the link connecting nodes 1 and 3 should have weight at most 3. This is shown graphically in [Figure 2.132](#).

Figure 2.132 Query Graph Q 

The topology of the query graph Q can be represented using the nodes query table (mylib.NodesQuery) and links query table (mylib.LinksQuery) that are created by the following DATA steps:

```

data mylib.NodesQuery;
    input node color $;
    datalines;
1 blue
2 red
3 green
;
data mylib.LinksQuery;
    input from to limit;
    datalines;
1 2 4
1 3 3
;
  
```

The restriction on the node's exact color is imposed as before using the nodes query table. The weight restrictions (limit) are not exact, although the data that are needed to apply the restriction are contained in the links query table. The following link filter function enforces that the weight of each link does not exceed the specified limit:

```

%macro linkFilterCode();
    function myLinkFilter(weight, limitQ);
        return (weight<=limitQ);
    endsub;
%mend linkFilterCode;
  
```

Next, the following statements find all subgraphs that have the patterns that are specified by the query input data tables and the FCMP filter functions:

```

proc network
    nodes          = mylib.Nodes
    links          = mylib.Links
    nodesQuery     = mylib.NodesQuery
    linksQuery     = mylib.LinksQuery;
    nodesVar
        vars      = (color);
    linksVar
  
```

```

        vars                = (weight);
nodesQueryVar
        vars                = (color);
linksQueryVar
        vars                = (limit)
        varsMatch           = ();
patternMatch
        code                = "%linkFilterCode()"
        linkFilter           = myLinkFilter(links.weight,linksQuery.limit)
        outMatchNodes        = mylib.OutMatchNodes
        outMatchLinks        = mylib.OutMatchLinks;
run;

```

In this case, you need the VARSMATCH= option in the LINKSQUERYVAR statement to indicate that you do not want to match the limit values exactly, but those values need to be included in the graph attributes (VAR=) for use in the FCMP link filter function.

The output data table mylib.OutMatchNodes now contains the mapping from nodes in the query graph to nodes in the main graph for each match, as shown in [Figure 2.133](#).

Figure 2.133 Node Mappings for Pattern Matches

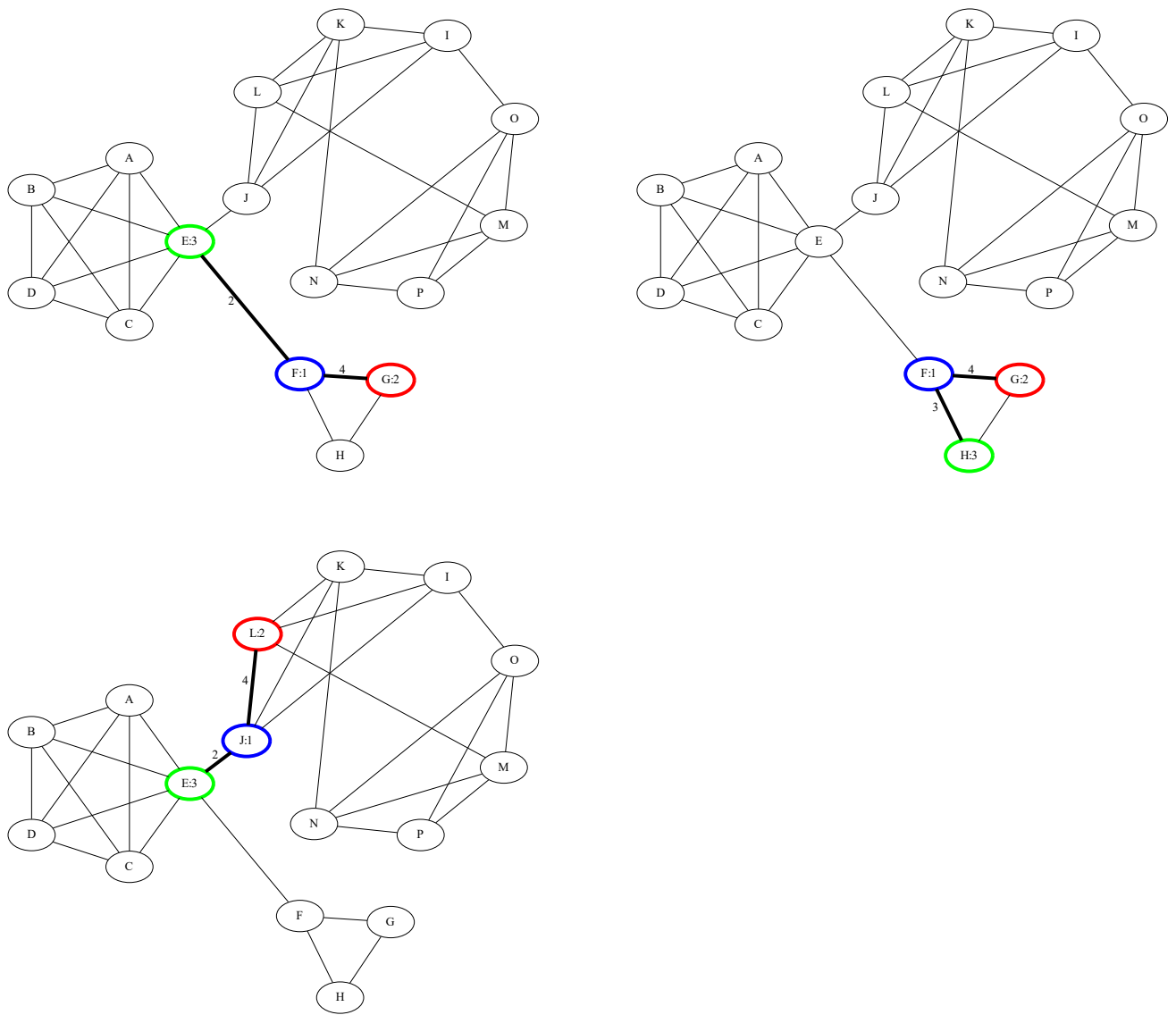
match	nodeQ	node	color
1	1	F	blue
1	2	G	red
1	3	E	green
2	1	F	blue
2	2	G	red
2	3	H	green
3	1	J	blue
3	2	L	red
3	3	E	green

The output data table mylib.OutMatchLinks now contains the subgraphs for each match, as shown in [Figure 2.134](#).

Figure 2.134 Subgraphs for Pattern Matches

match	from	to	weight
1	E	F	2
1	F	G	4
2	F	G	4
2	F	H	3
3	E	J	2
3	J	L	4

The results are displayed graphically in [Figure 2.135](#).

Figure 2.135 Subgraphs

Processing Multiple Queries in One Pass

For some applications, you might want to search for more than one pattern in the main graph. The `QUERYKEY=` option enables you to define all the patterns of interest in one pass. This can help improve performance because the processing is done in parallel across machines and threads (depending on your session configuration).

As an example, consider the following set of time-stamped transactions:

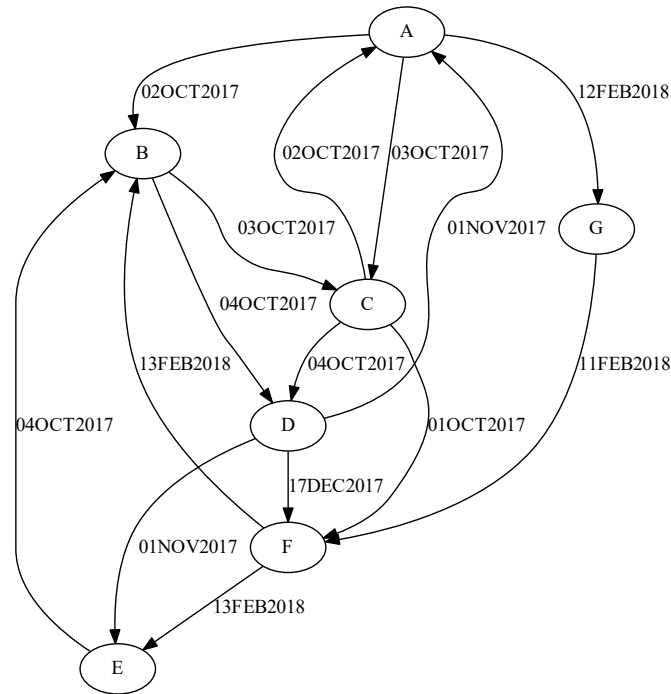
```

data mylib.Nodes;
    input node $ @@;
    label = node;
    datalines;
A B C D E F G
;
data mylib.Links;
    format time DATE9.;
    input from $ to $ time DATE9.;
    datalines;
A B 02OCT2017
A C 03OCT2017
A G 12FEB2018
B C 03OCT2017
B D 04OCT2017
C A 02OCT2017
C D 04OCT2017
C F 01OCT2017
D A 01NOV2017
D E 01NOV2017
D F 17DEC2017
E B 04OCT2017
F B 13FEB2018
F E 13FEB2018
G F 11FEB2018
;

```

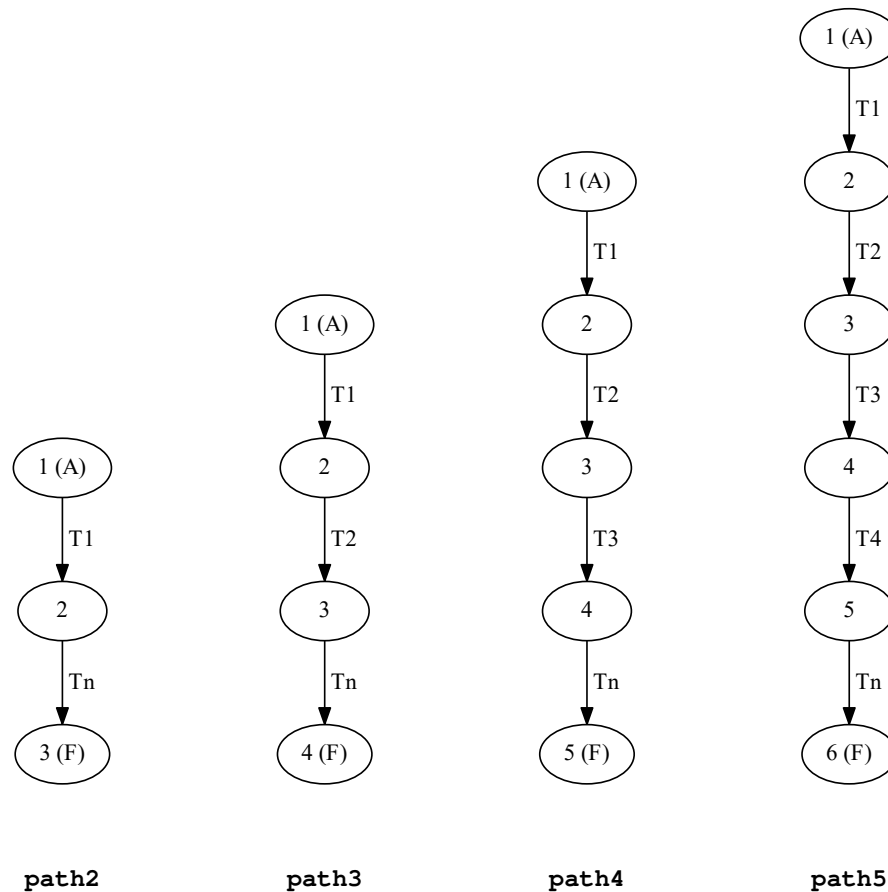
Each node has a node attribute label, and each link has a link attribute time.

The data are shown graphically in [Figure 2.136](#).

Figure 2.136 Directed Graph *G*

The pattern you want to find is a path of sequential transactions that starts at a node with label A and ends at a node with label F. The length of the path can be between two and five links. One approach to finding this set of patterns is to construct four query graphs and call the pattern-matching algorithm separately four times. A more efficient approach is to use the `QUERYKEY=` option and process these four queries in one pass.

The query graphs that define the patterns to search for are shown in Figure 2.137. In addition, the pattern requires that $T_1 < T_2 < \dots < T_n$.

Figure 2.137 Query Graphs

In order to construct this set of patterns (**path2** through **path5**), the query graphs can be represented using the data that are created by the following DATA steps:

```
data mylib.NodesQuery;
  input node $ label $ key $ @@;
  datalines;
1 A path2 3 F path2
1 A path3 4 F path3
1 A path4 5 F path4
1 A path5 6 F path5
;
data mylib.LinksQuery;
  input from $ to $ key $ @@;
  datalines;
1 2 path2 2 3 path2
1 2 path3 2 3 path3 3 4 path3
1 2 path4 2 3 path4 3 4 path4 4 5 path4
1 2 path5 2 3 path5 3 4 path5 4 5 path5 5 6 path5
;
```

The following statements specify the link-pair filter function. The link-pair filter function

`myLinkPairFilter` enforces that the sequence of transactions in the path has increasing timestamps.

```
%macro linkPairFilterCode();
  function myLinkPairFilter(fromQ[*] $, toQ[*] $, time[*]);
    if (toQ[1] = fromQ[2]) then
      return (time[1]<time[2]);
    else
      return (1);
    endsub;
%mend linkPairFilterCode;
```

Next, the following statements find all subgraphs that have the patterns that are specified by the query input data tables and the FCMP filter functions:

```
proc network
  direction          = directed
  nodes              = mylib.Nodes
  links              = mylib.Links
  nodesQuery         = mylib.NodesQuery
  linksQuery         = mylib.LinksQuery;
  nodesVar
    vars             = (label);
  nodesQueryVar
    vars             = (label);
  linksVar
    vars             = (time);
  patternMatch
    code             = "%linkPairFilterCode()"
    queryKey         = key
    queryKeyAggregate = true
    linkPairFilter   = myLinkPairFilter(linksQuery.from,linksQuery.to,links.time)
    outMatchNodes    = mylib.OutMatchNodes
    outMatchLinks    = mylib.OutMatchLinks
    outMatchGraphNodes = mylib.OutMatchGraphNodes
    outMatchGraphLinks = mylib.OutMatchGraphLinks
    outSummary       = mylib.OutMatchSummary;
run;
```

There are two matches with path length 3 and one match with path length 4. The output data table `mylib.OutSummary` contains the summary information about the executed queries, as shown in [Figure 2.138](#).

Figure 2.138 Summary Information for Executed Queries

key	nodes	links	matches	realTime
path2	3	2	0	0.007599
path3	4	3	2	0.028552
path4	5	4	1	0.025240
path5	6	5	0	0.029413

The output data table `mylib.OutMatchNodes` now contains the mapping from nodes in the query graph to nodes in the main graph for each pattern type (key) and each match, as shown in [Figure 2.139](#).

Figure 2.139 Node Mappings for Pattern Matches

key	match	nodeQ	node	label
path3	1	1	A	A
path3	1	2	B	B
path3	1	3	D	D
path3	1	4	F	F
path3	2	1	A	A
path3	2	2	C	C
path3	2	3	D	D
path3	2	4	F	F
path4	1	1	A	A
path4	1	2	B	B
path4	1	3	C	C
path4	1	4	D	D
path4	1	5	F	F

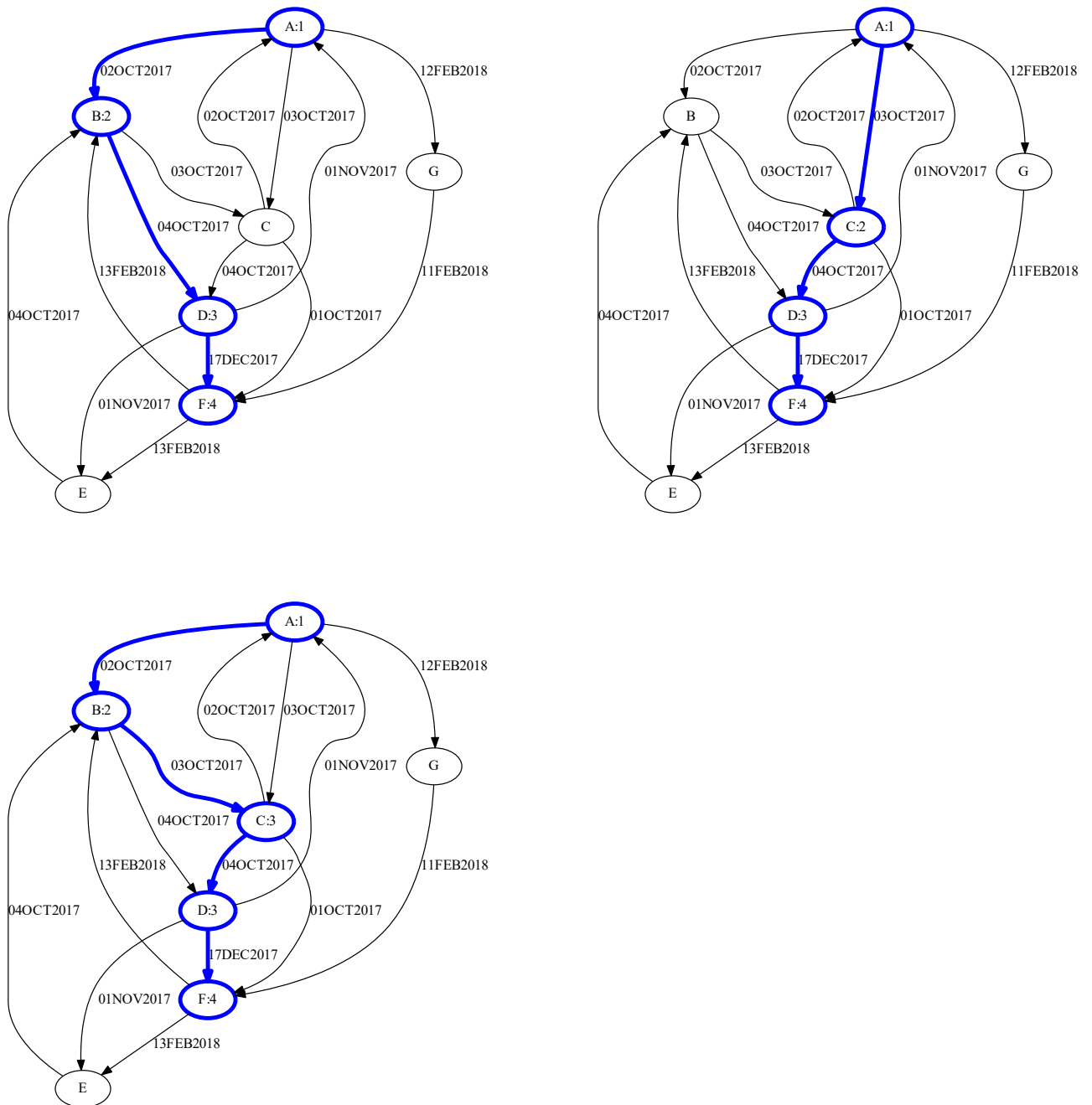
The output data table `mylib.OutMatchLinks` now contains the subgraphs for each pattern type (`key`) and each match, as shown in [Figure 2.140](#).

Figure 2.140 Subgraphs for Pattern Matches

key	match	from	to	time
path3	1	A	B	02OCT2017
path3	1	B	D	04OCT2017
path3	1	D	F	17DEC2017
path3	2	A	C	03OCT2017
path3	2	C	D	04OCT2017
path3	2	D	F	17DEC2017
path4	1	A	B	02OCT2017
path4	1	B	C	03OCT2017
path4	1	C	D	04OCT2017
path4	1	D	F	17DEC2017

The results are displayed graphically in [Figure 2.141](#).

Figure 2.141 Subgraphs



The output data table `mylib.OutMatchGraphNodes` now contains the nodes in the induced subgraph of matches (across query keys), as shown in Figure 2.142.

Figure 2.142 Nodes of the Induced Subgraph of Matches

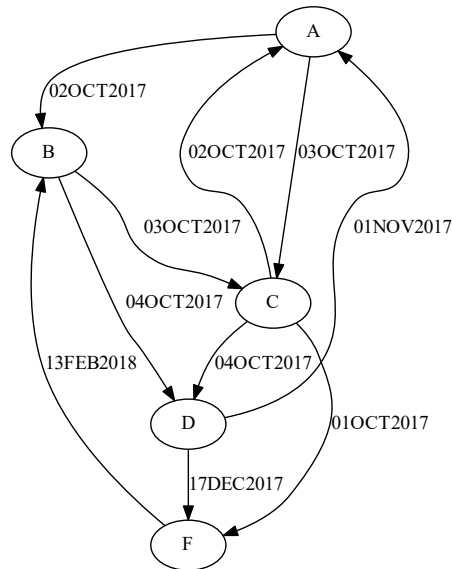
node	label
A	A
B	B
C	C
D	D
F	F

The output data table `mylib.OutMatchGraphLinks` now contains the links in the induced subgraph of matches, as shown in [Figure 2.143](#).

Figure 2.143 Links of the Induced Subgraph of Matches

from	to	time
A	B	02OCT2017
A	C	03OCT2017
B	C	03OCT2017
B	D	04OCT2017
C	A	02OCT2017
C	D	04OCT2017
C	F	01OCT2017
D	A	01NOV2017
D	F	17DEC2017
F	B	13FEB2018

The induced subgraph of matches is displayed graphically in [Figure 2.144](#).

Figure 2.144 Induced Subgraph

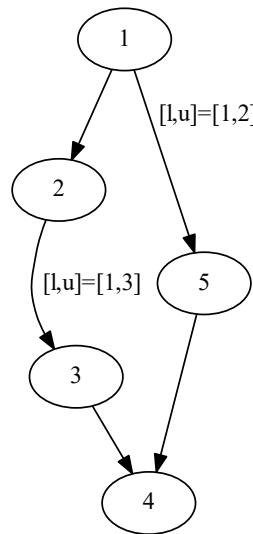
For an example of executing multiple queries in one pass to identify value-added tax carousel fraud, see “[Example 2.16: Detection of Value-Added Tax Carousel Fraud](#)” on page 321.

Generating and Processing Multiple Queries Using Path Expansion

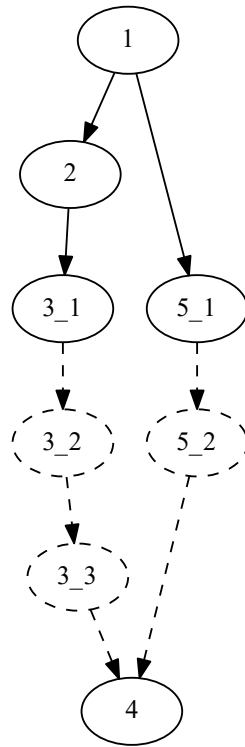
The example in the section “[Processing Multiple Queries in One Pass](#)” on page 193 uses the `QUERYKEY=` option to search for paths of sequential transactions whose path length ranges from three to six links. To construct these patterns, you explicitly define all the nodes and links for all four query graphs. Alternatively, you can use the `EXPANDLOWER=` and `EXPANDUPPER=` options in the `LINKSQUERYVAR` statement to automatically generate the query graphs that represent the four paths from one base query graph.

For each link in the links query table, the expansion lower bound (l) and the expansion upper bound (u) specify the number of copies of the *to* node to repeat in the generated paths.

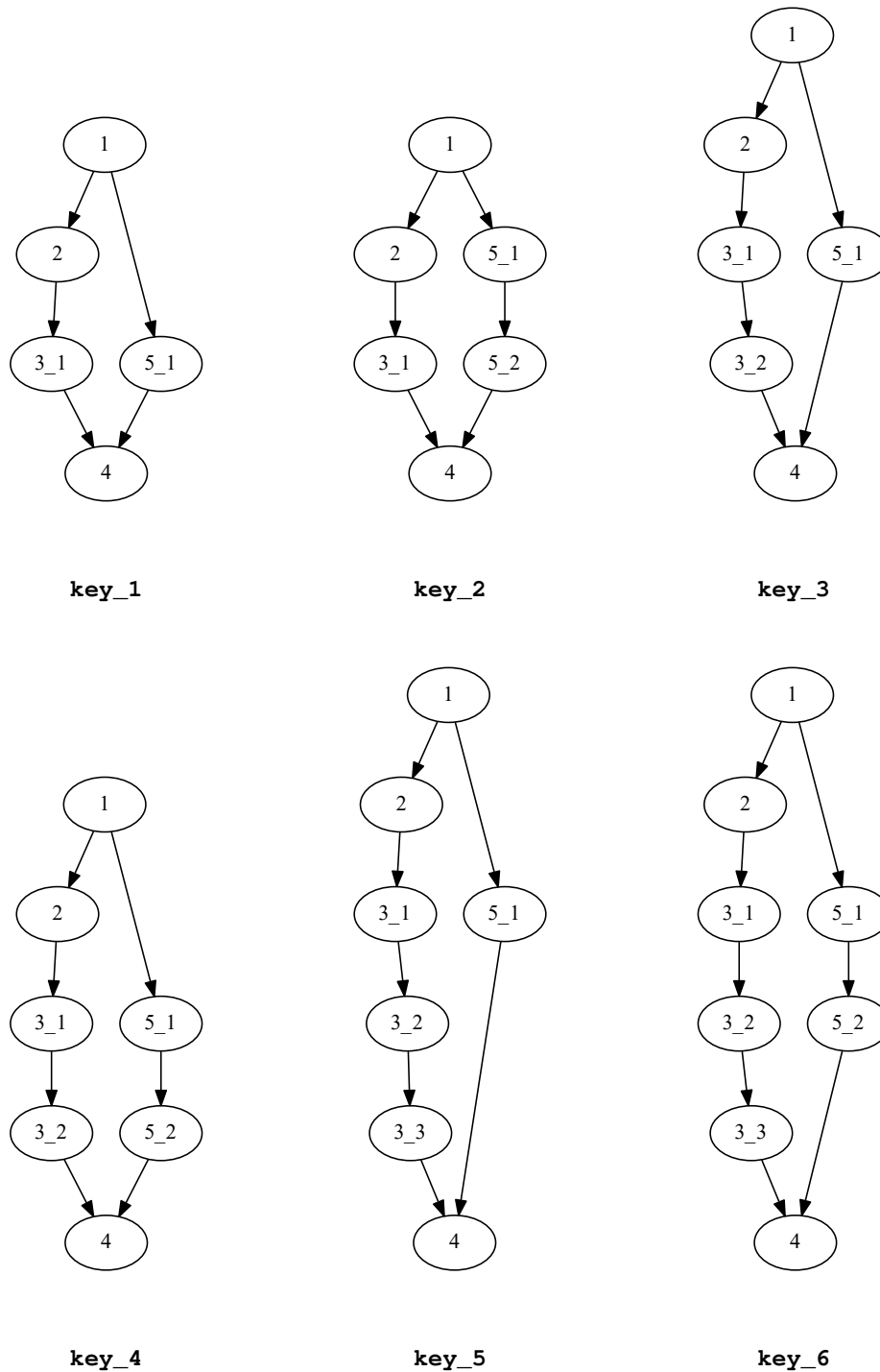
As a simple example, consider the base query graph in [Figure 2.145](#).

Figure 2.145 Base Query Graph

The expansion bounds $[l, u] = [1, 3]$ on link $2 \rightarrow 3$ imply that node 3 is to be expanded into a path whose length is one to three links, whereas the expansion bounds $[l, u] = [1, 2]$ on link $1 \rightarrow 5$ imply that node 5 is to be expanded into a path whose length is one to two links. In [Figure 2.146](#), the query graph is represented graphically; the dashed nodes and links are *optional* expansions of the associated paths.

Figure 2.146 Expanded Query Graph

There are three paths that involve node 3 and two paths that involve node 5. This yields six generated query graphs, as shown in [Figure 2.147](#).

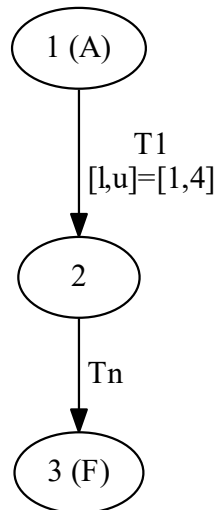
Figure 2.147 Generated Query Graphs

The generated nodes are labeled using a standard schema that appends "_" and the sequence number starting with the value that you specify in the [INDEXOFFSET=](#) option. The node labels in the base query graph must be character type. The generated query key values are labeled in a similar manner, starting with the prefix "key_".

In the case where path expansion leads to two generated nodes that have the same label, the generator produces an error message.

Consider again the base query graph, shown in [Figure 2.148](#), from the section “[Processing Multiple Queries in One Pass](#)” on page 193.

Figure 2.148 Base Query Graph for Sequential Transactions



Recall that the pattern you want to find is a path of sequential transactions that starts at a node with the label A and ends at a node with the label F. The length of the path can be between two and five links. The pattern requires that $T1 < T2 < \dots < Tn$.

To construct this set of patterns by using path expansion, you can use the following DATA steps to represent the query graphs:

```

data mylib.NodesQuery;
    input node $ label $;
    datalines;
1 A
3 F
;
data mylib.LinksQuery;
    input from $ to $ expandL expandU;
    datalines;
1 2 1 4
2 3 . .
;
  
```

Next, the following statements find all subgraphs that have the patterns that are specified by the query input data tables and the FCMP filter functions:

```

proc network
  direction          = directed
  nodes              = mylib.Nodes
  links              = mylib.Links
  nodesQuery         = mylib.NodesQuery
  linksQuery         = mylib.LinksQuery;
  nodesVar
    vars             = (label);
  nodesQueryVar
    vars             = (label);
  linksVar
    vars             = (time);
  linksQueryVar
    expandLower       = expandL
    expandUpper       = expandU;
  patternMatch
    code              = "%linkPairFilterCode()"
    linkPairFilter    = myLinkPairFilter(linksQuery.from,linksQuery.to,links.time)
    outMatchNodes     = mylib.OutMatchNodes
    outMatchLinks     = mylib.OutMatchLinks
    outQueryNodes     = mylib.OutQueryNodes
    outQueryLinks     = mylib.OutQueryLinks
    outSummary        = mylib.OutMatchSummary;
run;

```

The output data table `mylib.OutQueryNodes` contains the nodes of the query graphs that are constructed from the expansion of the base query graph, as shown in [Figure 2.149](#).

Figure 2.149 Query Graph Nodes

queryKey	node	label
key_1	1	A
key_1	2_1	
key_1	3	F
key_2	1	A
key_2	2_1	
key_2	2_2	
key_2	3	F
key_3	1	A
key_3	2_1	
key_3	2_2	
key_3	2_3	
key_3	3	F
key_4	1	A
key_4	2_1	
key_4	2_2	
key_4	2_3	
key_4	2_4	
key_4	3	F

The output data table `mylib.OutQueryLinks` contains the links of the query graphs that are constructed from

the expansion of the base query graph, as shown in [Figure 2.150](#).

Figure 2.150 Query Graph Links

queryKey	from	to
key_1	1	2_1
key_1	2_1	3
key_2	1	2_1
key_2	2_1	2_2
key_2	2_2	3
key_3	1	2_1
key_3	2_1	2_2
key_3	2_2	2_3
key_3	2_3	3
key_4	1	2_1
key_4	2_1	2_2
key_4	2_2	2_3
key_4	2_3	2_4
key_4	2_4	3

There are two matches whose path length is 3 and one match whose path length is 4. The output data table `mylib.OutSummary` contains the summary information about the executed queries, as shown in [Figure 2.151](#).

Figure 2.151 Summary Information for Executed Queries

queryKey	nodes	links	matches	realTime
key_1	3	2	0	0.029932
key_2	4	3	2	0.022594
key_3	5	4	1	0.007028
key_4	6	5	0	0.024964

The output data table `mylib.OutMatchNodes` contains the mapping from nodes in the query graph to nodes in the main graph for each pattern type (key) and each match, as shown in [Figure 2.152](#).

Figure 2.152 Node Mappings for Pattern Matches

queryKey	match	nodeQ	node	label
key_2	1	1	A	A
key_2	1	2_1	B	B
key_2	1	2_2	D	D
key_2	1	3	F	F
key_2	2	1	A	A
key_2	2	2_1	C	C
key_2	2	2_2	D	D
key_2	2	3	F	F
key_3	1	1	A	A
key_3	1	2_1	B	B
key_3	1	2_2	C	C
key_3	1	2_3	D	D
key_3	1	3	F	F

The output data table `mylib.OutMatchLinks` contains the subgraphs for each pattern type (key) and each match, as shown in [Figure 2.153](#).

Figure 2.153 Subgraphs for Pattern Matches

queryKey	match	from	to	time
key_2	1	A	B	02OCT2017
key_2	1	B	D	04OCT2017
key_2	1	D	F	17DEC2017
key_2	2	A	C	03OCT2017
key_2	2	C	D	04OCT2017
key_2	2	D	F	17DEC2017
key_3	1	A	B	02OCT2017
key_3	1	B	C	03OCT2017
key_3	1	C	D	04OCT2017
key_3	1	D	F	17DEC2017

For an example of generating and processing multiple queries by using path expansion to identify value-added tax carousel fraud, see “[Example 2.16: Detection of Value-Added Tax Carousel Fraud](#)” on page 321.

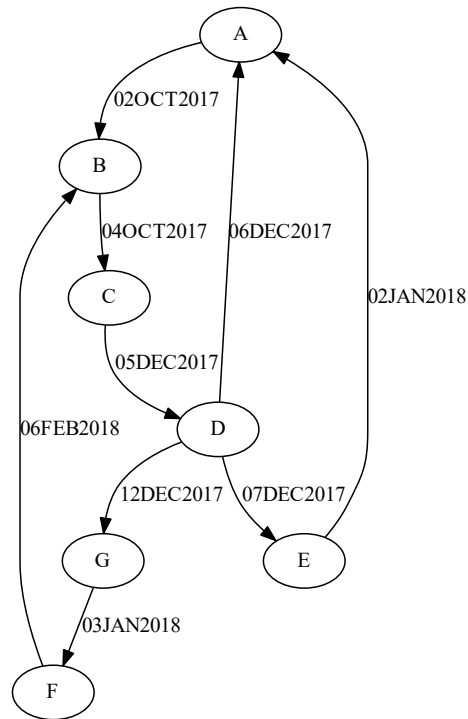
Pattern Matching of a Directed Graph Restricted to Induced Matches

By default, the pattern matching algorithm returns all subgraphs isomorphic to the defined query graph Q . For some applications, you might want to exclude matches whose [induced subgraph](#) is not topologically equivalent to the query graph. The `INDUCED=` option specifies whether to filter matches by using the induced subgraph of the match.

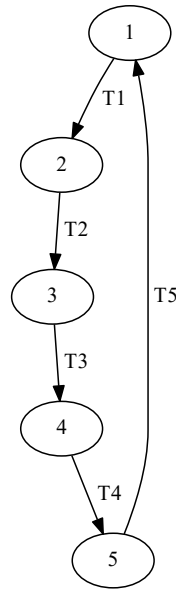
As an example, consider the following set of timestamped transactions:

```
data mylib.Links;
  format time DATE9.;
  input from $ to $ time DATE9.;
  datalines;
A B 02OCT2017
B C 04OCT2017
C D 05DEC2017
D A 06DEC2017
D E 07DEC2017
E A 02JAN2018
F B 06FEB2018
D G 12DEC2017
G F 03JAN2018
;
```

The data are shown graphically in [Figure 2.154](#).

Figure 2.154 Directed Graph *G*

The pattern that you want to find is a cycle of sequential transactions of length 5. The query graph that defines the patterns to search for is shown in [Figure 2.155](#). In addition, the pattern requires that $T1 < T2 < \dots < T5$.

Figure 2.155 Query Graph Q 

The query graph can be represented using the data that are created by the following DATA step:

```

data mylib.LinksQuery;
  input from to;
  datalines;
1 2
2 3
3 4
4 5
5 1
;

```

The following statements specify the link-pair filter function. The link-pair filter function `myLinkPairFilter` enforces that the sequence of transactions in the cycle has increasing timestamps.

```

%macro linkPairFilterCode();
  function myLinkPairFilter(fromQ[*], toQ[*], time[*]);
    if (toQ[1] = 1) then
      return (1);
    else if (toQ[1] = fromQ[2]) then
      return (time[1] < time[2]);
    else return (1);
  endsub;
%mend linkPairFilterCode;

```

Next, the following statements find all induced subgraphs that have the patterns that are specified by the query input data tables and the FCMP filter functions:

```

proc network
  direction      = directed
  links          = mylib.Links
  linksQuery     = mylib.LinksQuery;
  linksVar
    vars        = (time);
  patternMatch
    code        = "%linkPairFilterCode()"
    induced     = true
    linkPairFilter = myLinkPairFilter(linksQuery.from, linksQuery.to, links.time)
    outMatchLinks = mylib.OutMatchLinks
    outMatchNodes = mylib.OutMatchNodes;
run;

```

When you use default settings (INDUCED=FALSE), the cycles $B \rightarrow C \rightarrow D \rightarrow G \rightarrow F \rightarrow B$ and $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E \rightarrow A$ both match the requested query. However, in the latter case, the induced subgraph on the subset $\{A, B, C, D, E\}$ includes the link (D, A) , forming a chorded cycle. When you specify INDUCED=TRUE, the second candidate match is rejected because its induced subgraph is not topologically equivalent to the query graph.

The output data table mylib.OutMatchNodes now contains the mapping from nodes in the query graph to nodes in the main graph for the match, as shown in [Figure 2.156](#).

Figure 2.156 Node Mappings for Induced Pattern Matches

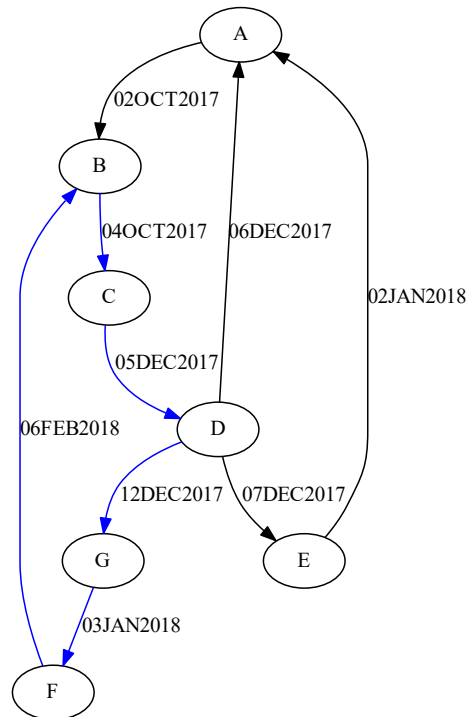
match	nodeQ	node
1	1	B
1	2	C
1	3	D
1	4	G
1	5	F

The output data table mylib.OutMatchLinks now contains the subgraphs for the match, as shown in [Figure 2.157](#).

Figure 2.157 Subgraphs for Induced Pattern Matches

match	from	to	time
1	B	C	04OCT2017
1	C	D	05DEC2017
1	D	G	12DEC2017
1	G	F	03JAN2018
1	F	B	06FEB2018

The results are displayed graphically in [Figure 2.158](#).

Figure 2.158 Subgraph

Network Projection

Network projection is the process of reducing a multiple-partition network to a single-partition network. When a network is projected onto one subset of nodes S through another subset of nodes T , the resulting network contains only the nodes of S . In the resulting network, links are present between any node pair that shares a common neighbor among the nodes of T in the original network. The algorithm assumes that the graph that is induced by $S \cup T$ in the original network is bipartite, so any links that exist among the nodes of S or among the links of T are ignored. In PROC NETWORK, you can calculate network projection by specifying the **PROJECTION** statement.

PROC NETWORK implements network projection according to partitions that are defined in the NODES= data table. You must use the PARTITION= option to specify the data variable that defines the partition of each node. The values of this variable must be either 1, 0, or missing. To find the projection onto node set S through node set T , assign the partition value 1 to nodes in set S and the partition value 0 to nodes in the set T . Nodes that have a missing indicator (.) for the partition flag or nodes that are absent from the NODES= data table are excluded from the projection computation.

In addition, the extent to which two nodes that are joined by a link in the projected graph are connected can be quantified on the basis of how many neighbors the nodes have in common. In PROC NETWORK, you can quantify each projected link by using one of the following similarity measures:

- Adamic-Adar, by specifying ADAMICADAR=TRUE

- common neighbors, by specifying `COMMONNEIGHBORS=TRUE`
- cosine, by specifying `COSINE=TRUE`
- Dice, by specifying `DICE=TRUE`
- Jaccard, by specifying `JACCARD=TRUE`

For the formal definition of each of these similarity measures, see the section “[Node Similarity](#)” on page 152.

When `DIRECTION=DIRECTED`, three methods are provided for handling directed links. When `DIRECTEDMETHOD=CONVERGING`, only out-neighbors of nodes in S are considered (that is, nodes in T that are connected by outgoing links). When `DIRECTEDMETHOD=DIVERGING`, only in-neighbors of nodes in S are considered (that is, nodes in T that are connected by ingoing links). When `DIRECTEDMETHOD=TRANSITIVE`, only paths of length two are considered (that is, paths that start at some node in S , traverse through some node in T , and end at some node in S). By default, `DIRECTEDMETHOD=CONVERGING`.

You can also produce the lists of neighbors that are common to each node pair that is joined by a link in the projected graph. These lists are written to the output data table that you specify in the `OUTNEIGHBORSLIST=` option.

Output Data Tables

The network projection algorithm populates up to three output data tables, which are described in the following subsections.

OUTPROJECTIONLINKS= Data Table

The output data table that you specify in the `OUTPROJECTIONLINKS=` option in the `PROJECTION` statement contains the links of the projected graph. Optionally, this table also reports the requested similarity scores that are computed between pairs of nodes that are represented by a link in the projected graph. These scores quantify the extent to which the node pair is connected through nodes that are indicated by a partition flag of 0 in the input graph.

The `OUTPROJECTIONLINKS=` data table contains the following columns:

- *from*: the *from* node label of this link
- *to*: the *to* node label of this link
- *adamicAdar*: the Adamic-Adar similarity score between the *from* and *to* nodes, if you specify `ADAMIC-ADAR=TRUE`
- *commonNeighbors*: the common neighbors similarity score between the *from* and *to* nodes, if you specify `COMMONNEIGHBORS=TRUE`
- *cosine*: the cosine similarity score between the *from* and *to* nodes, if you specify `COSINE=TRUE`
- *dice*: the Dice similarity score between the *from* and *to* nodes, if you specify `DICE=TRUE`
- *jaccard*: the Jaccard similarity score between the *from* and *to* nodes, if you specify `JACCARD=TRUE`

OUTPROJECTIONNODES= Data Table

The output data table that you specify in the **OUTPROJECTIONNODES=** option in the **PROJECTION** statement contains the nodes of the projected graph. This data table contains the following column:

- **node:** the node label

OUTNEIGHBORSLIST= Data Table

The output data table that you specify in the **OUTNEIGHBORSLIST=** option in the **PROJECTION** statement contains the lists of common neighbors between pairs of nodes. For each link that is listed in the **OUTPROJECTIONLINKS=** data table, each neighbor that is common to both the *from* and *to* nodes is listed in this table. There is one observation for each neighbor.

The **OUTNEIGHBORSLIST=** data table contains the following columns:

- **from:** the *from* node label
- **to:** the *to* node label
- **neighbor_id:** the sequential index of the neighbor
- **neighbor:** the node label of the common neighbor of the *from* and *to* nodes

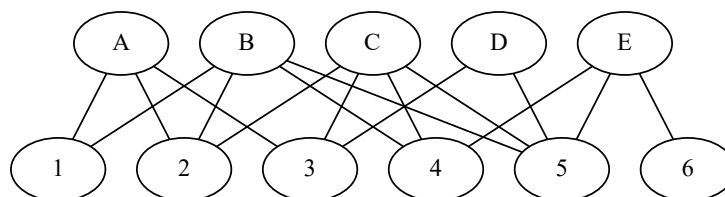
The **neighbor_id** identifiers are numbered sequentially, starting from the value of the **INDEXOFFSET=** option in the **PROC NETWORK** statement.

By default, the output data tables **OUTPROJECTIONLINKS=** and **OUTNEIGHBORSLIST=** represent undirected graphs, and only the observations in which *from* is less than *to* are output. However, when **DIRECTION=DIRECTED** and **DIRECTEDMETHOD=TRANSITIVE**, the output is a directed graph, and the output data tables might contain relationships in both directions.

Network Projection of a Bipartite Graph

This section provides a simple example that calculates a network projection of the bipartite graph *G* shown in Figure 2.159.

Figure 2.159 Undirected Graph *G*



The bipartite graph *G* can be represented by a links data table. You can create this table, **mylib.LinkSetIn**, by using the following **DATA** step:

```

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

```

In order to identify the partition of each node, you must also specify a nodes data table, mylib.NodeSetIn, which you can create by using the following DATA step:

```

data mylib.NodeSetIn;
    input node $ partitionFlag @@;
    datalines;
A 1  B 1  C 1  D 1  E 1
1 0  2 0  3 0  4 0  5 0  6 0
;

```

The following statements find the projection of the network onto nodes A through E:

```

proc network
    links          = mylib.LinkSetIn
    nodes          = mylib.NodeSetIn;
    projection
        partition  = partitionFlag
        outProjectionLinks = mylib.ProjLinkSetOut
        commonNeighbors = true;
run;

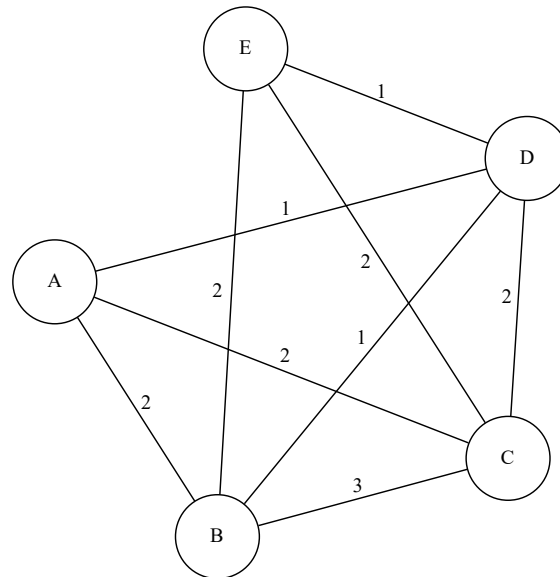
```

The output data table mylib.ProjLinkSetOut, shown in [Figure 2.160](#), contains the links of the projected graph. For each link, the number of neighbors that connect the *from* and *to* nodes is contained in the commonNeighbors variable.

Figure 2.160 Links of the Projected Graph

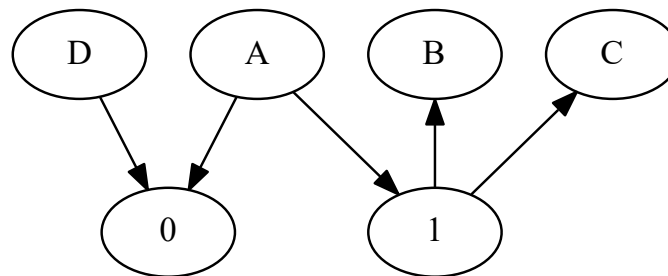
from	to	commonNeighbors
A	B	2
A	C	2
A	D	1
B	C	3
B	D	1
B	E	2
C	D	2
C	E	2
D	E	1

The projected graph is shown in [Figure 2.161](#).

Figure 2.161 Projected Graph

Network Projection of a Directed Bipartite Graph

This section provides a simple example that calculates a network projection of the directed bipartite graph G shown in Figure 2.159 using three projection methods.

Figure 2.162 Directed Graph G 

The bipartite graph G can be represented by a links data table. You can create this table, `mylib.LinkSetIn`, by using the following DATA step:


```

data mylib.LinkSetIn;
  input from $ to $ @@;
  datalines;
1 B 1 C A 0 A 1 D 0
;

```

In order to identify the partition of each node, you must also specify a nodes data table, `mylib.NodeSetIn`, which you can create by using the following DATA step:

```

data mylib.NodeSetIn;
  input node $ partitionFlag @@;
  datalines;
0 0 1 0 A 1 B 1 C 1 D 1
;

```

The following statements find the projection of the network onto nodes A through D via their out-neighbors:

```

proc network
  direction          = directed
  links              = mylib.LinkSetIn
  nodes              = mylib.NodeSetIn;
  projection
    directedMethod   = converging
    partition         = partitionFlag
    outProjectionLinks = mylib.ProjLinkSetOut
    commonNeighbors   = true;
run;

```

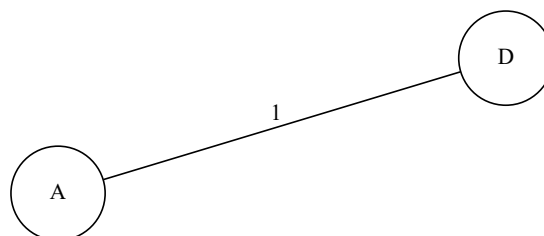
The output data table `mylib.ProjLinkSetOut`, shown in [Figure 2.163](#), contains the links of the projected graph. For each link, the number of neighbors that connect the *from* and *to* nodes is contained in the `commonNeighbors` variable.

Figure 2.163 Links of the Projected Graph

from to commonNeighbors		
A	D	1

The projected graph is shown in [Figure 2.164](#).

Figure 2.164 Projected Graph



The following statements find the projection of the network onto nodes A through D via their in-neighbors:

```
proc network
  direction          = directed
  links              = mylib.LinkSetIn
  nodes              = mylib.NodeSetIn;
  projection
    directedMethod    = diverging
    partition          = partitionFlag
    outProjectionLinks = mylib.ProjLinkSetOut
    commonNeighbors    = true;
run;
```

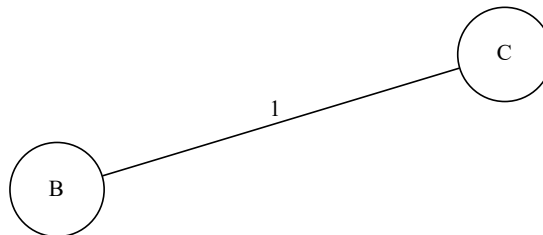
The output data table `mylib.ProjLinkSetOut`, shown in Figure 2.165, contains the links of the projected graph. For each link, the number of neighbors that connect the *from* and *to* nodes is contained in the `commonNeighbors` variable.

Figure 2.165 Links of the Projected Graph

from to		commonNeighbors
B	C	1

The projected graph is shown in Figure 2.166.

Figure 2.166 Projected Graph



The following statements find the projection of the network onto nodes A through D via directed paths of length 2:

```
proc network
  direction          = directed
  links              = mylib.LinkSetIn
  nodes              = mylib.NodeSetIn;
  projection
    directedMethod    = transitive
    partition          = partitionFlag
    outProjectionLinks = mylib.ProjLinkSetOut
    commonNeighbors    = true;
run;
```

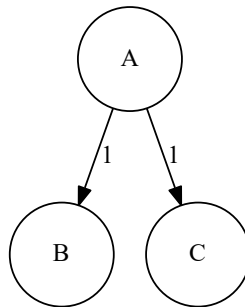
The output data table `mylib.ProjLinkSetOut`, shown in [Figure 2.167](#), contains the links of the projected graph. For each link, the number of neighbors that connect the *from* and *to* nodes is contained in the `commonNeighbors` variable.

Figure 2.167 Links of the Projected Graph

from	to	commonNeighbors
A	B	1
A	C	1

The projected graph is shown in [Figure 2.168](#).

Figure 2.168 Projected Graph



Reach (Ego) Network

The *reach network* of a graph $G = (N, E)$ is a graph $G_L^R = (N_L^R, E_L^R)$ 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*. In the context of social networks, reach networks are often referred to as *ego networks*, because they focus on the neighbors of one particular individual (or more than one).

For a directed graph, the set of reachable nodes, N_L^R is defined using outgoing links and its [induced subgraph](#) (using outgoing links only) is called the *out-reach network*.

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 56.

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 67. The nodes subset data table can be used to define several sets of source 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 that is specified in the OUTNODES= option in the PROC NETWORK statement.

You specify the number L of steps by using the **MAXREACH=** option in the REACH statement. If MAXREACH=0, then the reach algorithm produces the induced subgraph of the defined set of source nodes.

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 in the (out-)reach network from the set of source nodes
- count_not: the number of nodes not in the (out-)reach network from the set of source nodes

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

- count_wt: the sum of the weights of the nodes counted in count
- count_not_wt: the sum of the weights of the nodes counted in count_not

If the graph is directed and you specify the **DIGRAPH** option (and MAXREACH=1), then the OUTCOUNTS= data table contains the following additional columns:

- `count_in`: the number of nodes that reach some node in the set of source nodes using at most one step
- `count_out`: the number of nodes in the out-reach network from the set of source nodes (equivalent to `count`)
- `count_in_or_out`: the number of nodes in the union of the set of nodes counted in `count_in` and `count_out`
- `count_in_and_out`: the number of nodes in the intersection of the set of nodes counted in `count_in` and `count_out`

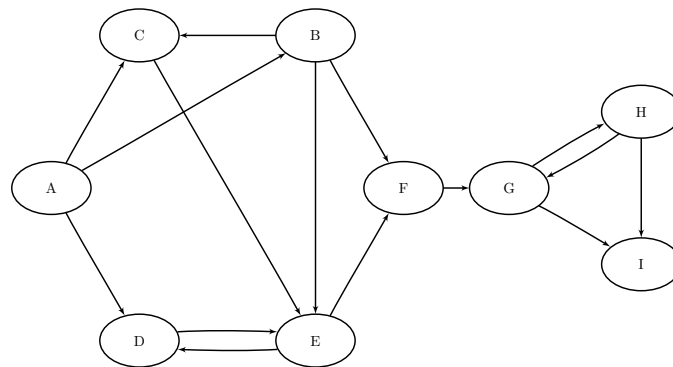
If node weights are present, the `OUTCOUNTS=` data table contains the following additional columns:

- `count_in_wt`: the sum of the weights of the nodes counted in `count_in`
- `count_out_wt`: the sum of the weights of the nodes counted in `count_out`
- `count_in_or_out_wt`: the sum of the weights of the nodes counted in `count_in_or_out`
- `count_in_and_out_wt`: the sum of the weights of the nodes counted in `count_in_and_out`

Reach Network of a Directed Graph

This section illustrates the use of the reach networks algorithm on the directed graph G shown in Figure 2.169.

Figure 2.169 Directed Graph G



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

```

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

data mylib.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          = mylib.LinkSetIn
  nodesSubset    = mylib.NodeSubSetIn1;
  reach
    outReachNodes = mylib.ReachNodes1
    outReachLinks = mylib.ReachLinks1
    outCounts     = mylib.ReachCounts1
    maxReach      = 1;
run;

```

The output data tables mylib.ReachNodes1, mylib.ReachLinks1, and mylib.ReachCounts1 now contain the nodes, links, and counts of the reach network, respectively, that come from S_1 . They are shown in Figure 2.170.

Figure 2.170 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

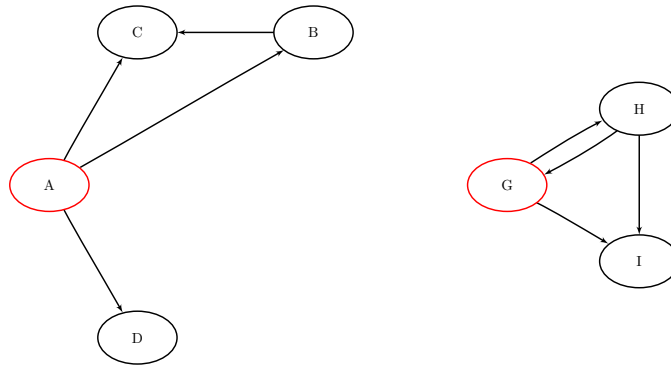
Figure 2.170 *continued***ReachLinks1**

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

ReachCounts1

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

The results are displayed graphically in [Figure 2.171](#).

Figure 2.171 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          = mylib.LinkSetIn
  nodesSubset    = mylib.NodeSubSetIn2;
  reach
    outReachNodes = mylib.ReachNodes2
    outReachLinks = mylib.ReachLinks2
    outCounts     = mylib.ReachCounts2
    maxReach      = 2;
run;
```

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

Figure 2.172.

Figure 2.172 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

ReachLinks2

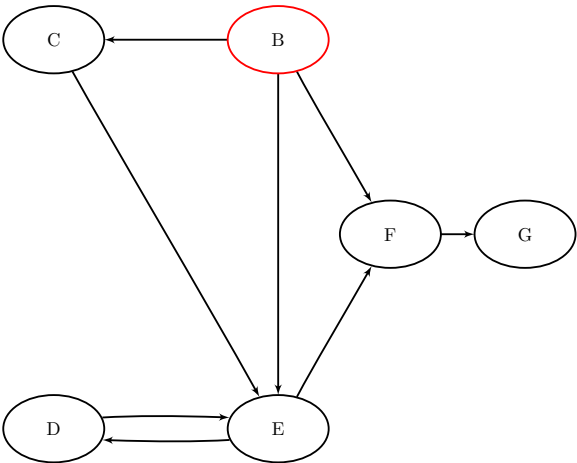
reach	from	to
1	B	C
1	B	E
1	B	F
1	C	E
1	D	E
1	E	D
1	E	F
1	F	G

ReachCounts2

reach	node	count	count_not
1	B	6	3

The results are displayed graphically in Figure 2.173.

Figure 2.173 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 2.169, now with source node sets $S_1 = \{C\}$ and $S_2 = \{A, H\}$. These source node sets can be defined together as follows:

```
data mylib.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           = mylib.LinkSetIn
  nodesSubset     = mylib.NodeSubSetIn;
  reach
    outReachNodes = mylib.ReachNodes
    outReachLinks = mylib.ReachLinks
    outCounts      = mylib.ReachCounts
    maxReach       = 1;
run;
```

The output data tables `mylib.ReachNodes`, `mylib.ReachLinks`, and `mylib.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 2.174.

Figure 2.174 Reach Networks for $S_1 = \{C\}$ and $S_2 = \{A, H\}$ with Hop Limit of 1

ReachNodes

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

Figure 2.174 continued

ReachLinks

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

ReachCounts

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

Shortest Path

A *shortest path* between two nodes, i and j , in a graph is a path that starts at i and ends at j 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 57.

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 67. The following sections show examples of how to use these options.

Which algorithm PROC NETWORK uses to find shortest paths depends on the data. The algorithm and run-time complexity for each link type are shown in [Table 2.14](#).

Table 2.14 Algorithm Complexity for Shortest Paths

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

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

By default, `MAXPATHSPERPAIR=1` and the algorithm returns one shortest path for each source-sink pair. If you set the `MAXPATHSPERPAIR=` option to a value K greater than 1, `PROC NETWORK` finds, for each source-sink pair, up to K paths arranged in ascending order in the sum of link weights for each path. In this case, the algorithm that is used is a variant of Yen's algorithm (Yen 1971). The run-time complexity of Yen's algorithm for each link type is shown in Table 2.15.

Table 2.15 Algorithm Complexity for Ranked Paths

Link Type	Complexity (per Source-Sink Pair)
Weighted (nonnegative)	$O(K N (N \log N + E))$
Weighted (positive and negative allowed)	$O(K N ^2 E)$

For weighted graphs, the algorithms use 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 four output data tables. The output data table that you specify in the `OUTPATHSLINKS=` option contains the links of the shortest paths for each source-sink pair. The output data table that you specify in the `OUTPATHSNODES=` option contains the nodes of the shortest paths for each source-sink pair. The output data table that you specify in the `OUTWEIGHTS=` option contains the total weight for the shortest paths for each source-sink pair. The output data table that you specify in the `OUTSUMMARY=` option contains descriptive statistics of the finite shortest paths for each source.

OUTPATHSLINKS= Data Table

The `OUTPATHSLINKS=` 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: Fundamentals*.

An example of finding the all-pairs shortest path for this road network is shown in “Example 2.13: Shortest Paths of the New York Road Network” on page 307.

The `OUTPATHSLINKS=` 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 the shortest path

- *from*: the *from* node label of this link in the shortest path
- *to*: the *to* node label of this link in the shortest path
- *weight*: the weight of this link in the shortest path

If you use the `AUXWEIGHT=` option in the `LINKSVAR` statement, the following column also appears in the output table:

- *column*: the auxiliary weight of this link

If the value of the `MAXPATHSPERPAIR=` option is greater than 1, the following column also appears in the output table:

- *rank*: for this source-sink pair, the rank of this shortest path

OUTPATHSNODES= Data Table

The `OUTPATHSNODES=` data table contains the nodes present in 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
- *order*: for this source-sink pair, the order of this link in the shortest path
- *node*: the node label of this node in the shortest path

If the value of the `MAXPATHSPERPAIR=` option is greater than 1, the following column also appears in the output table:

- *rank*: for this source-sink pair, the rank of this shortest path

OUTSUMMARY= Data Table

The `OUTSUMMARY=` data table contains descriptive statistics for the finite shortest paths for each source. This data table contains the following columns:

- *source*: the source node label
- *paths*: the number of finite shortest paths from source to requested sinks
- *path_weight_min*: the minimum weight of shortest paths from source to requested sinks
- *path_weight_max*: the maximum weight of shortest paths from source to requested sinks
- *path_weight_avg*: the average weight of shortest paths from source to requested sinks
- *path_weight_std*: the standard deviation of weight of shortest paths from source to requested sinks

- `path_weight_var`: the variance of weight of shortest paths from source to requested sinks

If you use the `AUXWEIGHT=` option in the `LINKSVAR` statement, the following columns also appear in the output table:

- `path_auxweight_min`: the minimum auxiliary weight of shortest paths from source to requested sinks
- `path_auxweight_max`: the maximum auxiliary weight of shortest paths from source to requested sinks
- `path_auxweight_avg`: the average auxiliary weight of shortest paths from source to requested sinks
- `path_auxweight_std`: the standard deviation of auxiliary weight of shortest paths from source to requested sinks
- `path_auxweight_var`: the variance of auxiliary weight of shortest paths from source to requested sinks

OUTWEIGHTS= Data Table

The `OUTWEIGHTS=` data table contains the weight (and 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 weight of the shortest path for this source-sink pair

If you use the `AUXWEIGHT=` option in the `LINKSVAR` statement, the following column also appears in the output table:

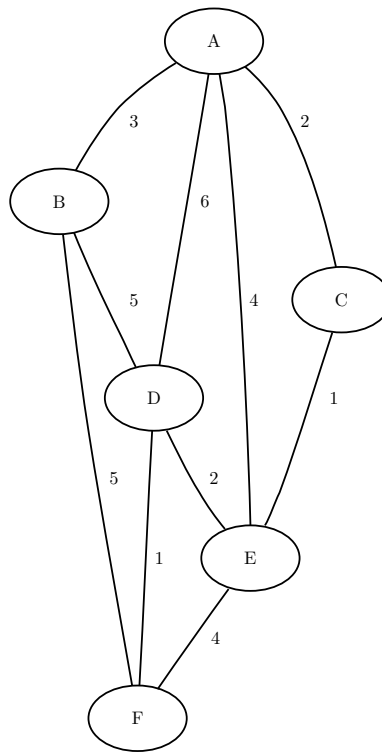
- `path_auxweight`: the auxiliary weight of the shortest path for this source-sink pair (if you specify the `AUXWEIGHT=` option in the `LINKSVAR` statement)

If the value of the `MAXPATHSPERPAIR=` option is greater than 1, the following column also appears in the output table:

- *rank*: for this source-sink pair, the rank of this shortest path

Shortest Paths for All Pairs

This example illustrates the use of the shortest path algorithm for all source-sink pairs on the undirected graph *G* shown in [Figure 2.175](#).

Figure 2.175 Undirected Graph *G*

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

```
data mylib.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          = mylib.LinkSetIn;
  shortestPath
    outSummary   = mylib.ShortPathS
    outWeights   = mylib.ShortPathW
    outPathsLinks = mylib.ShortPathP;
run;
```

The output data table `mylib.ShortPathP` contains the shortest paths, as shown in [Figure 2.176](#).

Figure 2.176 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	D	E	2
A	E	1	A	C	2
A	E	2	C	E	1
A	F	1	A	C	2
A	F	2	C	E	1
A	F	3	D	E	2
A	F	4	D	F	1
B	A	1	A	B	3
B	C	1	A	B	3
B	C	2	A	C	2
B	D	1	B	D	5
B	E	1	A	B	3
B	E	2	A	C	2
B	E	3	C	E	1
B	F	1	B	F	5
C	A	1	A	C	2
C	B	1	A	C	2
C	B	2	A	B	3
C	D	1	C	E	1
C	D	2	D	E	2
C	E	1	C	E	1
C	F	1	C	E	1
C	F	2	D	E	2
C	F	3	D	F	1

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

The output data table `mylib.ShortPathW` contains the path weights of the shortest paths of each source-sink pair, as shown in [Figure 2.177](#).

Figure 2.177 All-Pairs Shortest Paths Weights

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

The output data table `mylib.ShortPathS` contains descriptive statistics of the finite shortest paths for each source, as shown in [Figure 2.178](#).

Figure 2.178 All-Pairs Shortest Paths Summary

source	paths	path_weight_min	path_weight_max	path_weight_avg	path_weight_std	path_weight_var
C	5	1	5	3.0	1.58114	2.5
D	5	1	5	3.2	1.78885	3.2
F	5	1	6	3.8	1.92354	3.7
A	5	2	6	3.8	1.64317	2.7
E	5	1	6	3.0	1.87083	3.5
B	5	3	6	4.8	1.09545	1.2

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 mylib.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      = mylib.NodeSubSetIn
  links            = mylib.LinkSetIn;
  shortestPath
    outPathsLinks = mylib.ShortPathP;
run;
```

The output data table mylib.ShortPathP contains the shortest paths, as shown in [Figure 2.179](#).

Figure 2.179 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	D	E	2
A	F	4	D	F	1
C	B	1	A	C	2
C	B	2	A	B	3
C	F	1	C	E	1
C	F	2	D	E	2
C	F	3	D	F	1

Shortest Paths for a Subset of Source or Sink Pairs

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

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

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

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

You can use the same PROC NETWORK call that is used in the section “[Shortest Paths for a Subset of Source-Sink Pairs](#)” on page 232 to find all the shortest paths from nodes *B* and *E*. The output data table mylib.ShortPathP contains the shortest paths, as shown in [Figure 2.180](#).

Figure 2.180 Shortest Paths for a Subset of Source Pairs

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

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

```
data mylib.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 mylib.ShortPathP contains the shortest paths, as shown in [Figure 2.181](#).

Figure 2.181 Shortest Paths for a Subset of Sink Pairs

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

Shortest Paths for One Source-Sink Pair

This section illustrates the use of the shortest path algorithm to 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          = mylib.LinkSetIn;
  shortestPath
    source       = C
    sink         = F
    outPathsLinks = mylib.ShortPathP
    outPathsNodes = mylib.ShortPathN;
run;
```

The output data table mylib.ShortPathP contains the links of this shortest path, as shown in Figure 2.182.

Figure 2.182 Shortest Path Links for One Source-Sink Pair

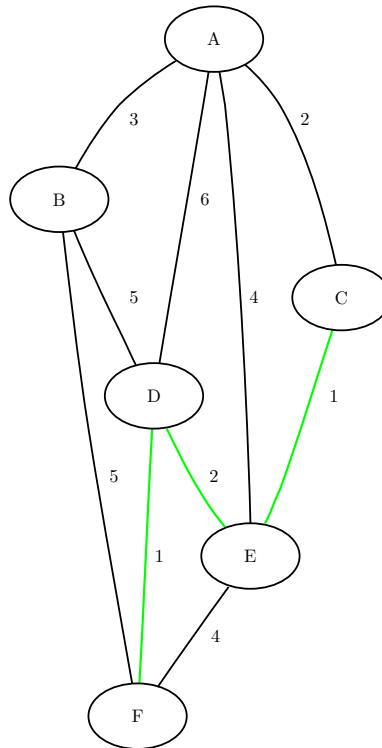
source	sink	order	from	to	weight
C	F	1	C	E	1
C	F	2	D	E	2
C	F	3	D	F	1

The output data table mylib.ShortPathN contains the nodes of this shortest path, as shown in Figure 2.183.

Figure 2.183 Shortest Path Nodes for One Source-Sink Pair

source	sink	order	node
C	F	1	C
C	F	2	E
C	F	3	D
C	F	4	F

The shortest path is shown graphically in [Figure 2.184](#).

Figure 2.184 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 mylib.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          = mylib.LinkSetIn;
  linksVar
    auxWeight    = count;
  shortestPath
    outWeights   = mylib.ShortPathW;
run;

```

The output data table `mylib.ShortPathW` contains the total path weight of shortest paths in each source-sink pair, as shown in Figure 2.185. Because the variable `count` in `mylib.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 2.185 Shortest Paths Including Auxiliary Weights in Calculation

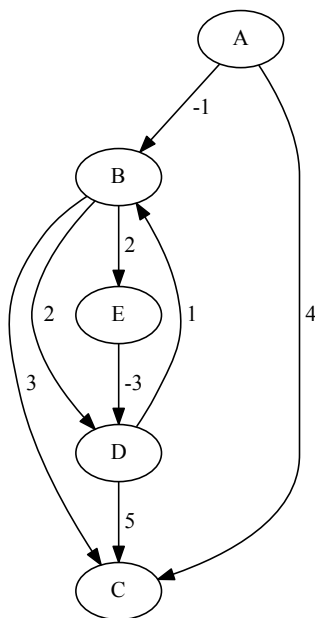
source	sink	path_weight	path_auxweight	source	sink	path_weight	path_auxweight
A	B	3	1	D	A	5	3
A	C	2	1	D	B	5	1
A	D	5	3	D	C	3	2
A	E	3	2	D	E	2	1
A	F	6	4	D	F	1	1
B	A	3	1	E	A	3	2
B	C	5	2	E	B	6	3
B	D	5	1	E	C	1	1
B	E	6	3	E	D	2	1
B	F	5	1	E	F	3	2
C	A	2	1	F	A	6	4
C	B	5	2	F	B	5	1
C	D	3	2	F	C	4	3
C	E	1	1	F	D	1	1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

The section “[Road Network Shortest Path](#)” on page 11 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 directed graph G with negative link weights, shown in Figure 2.186.

Figure 2.186 Directed Graph *G* with Negative Link Weights



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

```
data mylib.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          = mylib.LinkSetIn;
  shortestPath
    source       = E
    sink         = B
    outPathsLinks = mylib.ShortPathP;
run;
```

The output data table mylib.ShortPathP contains a shortest path from node *E* to node *B*, as shown in Figure 2.187.

Figure 2.187 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 mylib.LinkSetIn;
  set mylib.LinkSetIn;
  if (from="B" and to="E") then
    weight=1;
run;
```

In this case, there is a negative weight cycle $(E \rightarrow D \rightarrow B \rightarrow E)$. The Bellman-Ford algorithm catches the cycle and produces an error message, as shown in [Figure 2.188](#).

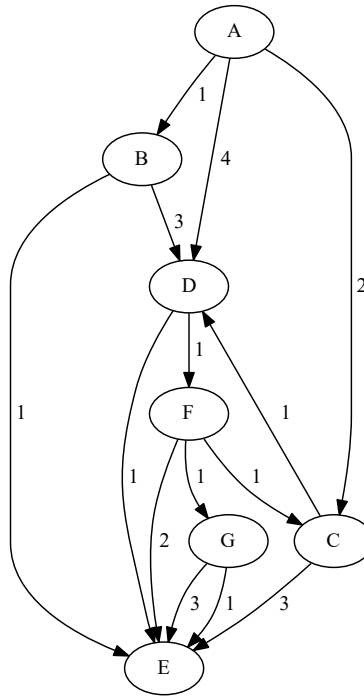
Figure 2.188 PROC NETWORK Log: Negative Weight Cycle

```
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 shortest paths problem using 1200 threads across 15 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 0.01 (cpu: 0.00) seconds.
ERROR: The action stopped due to errors.
NOTE: The Cloud Analytic Services server processed the request in 0.562228 seconds.
NOTE: The SAS System stopped processing this step because of errors.
STATUS=ERROR  PROBLEM_TYPE=SHORTESTPATH  CPU_TIME=0.82  REAL_TIME=0.56
```

Ranked Paths

This section illustrates the use of the shortest path algorithm to find ranked paths that are arranged in ascending order in the sum of link weights for each path. The `MAXPATHSPERPAIR=` option defines the maximum number of ranked paths per source-sink pair to return.

Consider the directed graph G shown in [Figure 2.189](#).

Figure 2.189 Directed Graph *G*

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

```

data mylib.LinkSetIn;
  input from $ to $ weight @@;
  datalines;
A B 1  A D 4  A C 2  B E 1  B D 3
D E 1  C D 1  C E 3  D F 1  F C 1
F E 2  F G 1  G E 1  G E 3
;

```

The following statements find the four shortest paths between the source node *A* and the sink node *E*:

```

proc network
  direction          = directed
  links              = mylib.LinkSetIn;
  shortestPath
    source           = A
    sink             = E
    maxPathsPerPair = 4
    outWeights       = mylib.ShortPathW
    outPathsLinks    = mylib.ShortPathP;
run;

```

The output data table mylib.ShortPathW contains the path weights of the ranked paths, as shown in [Figure 2.190](#).

Figure 2.190 Path Weights of Ranked Paths for One Source-Sink Pair

source	sink	rank	path_weight
A	E	1	2
A	E	2	4
A	E	3	5
A	E	4	5

The output data table `mylib.ShortPathP` contains the ranked paths (displayed by rank), as shown in [Figure 2.191](#).

Figure 2.191 Ranked Paths for One Source-Sink Pair by Rank

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

The ranked paths are shown graphically in [Figure 2.192](#) and [Figure 2.193](#).

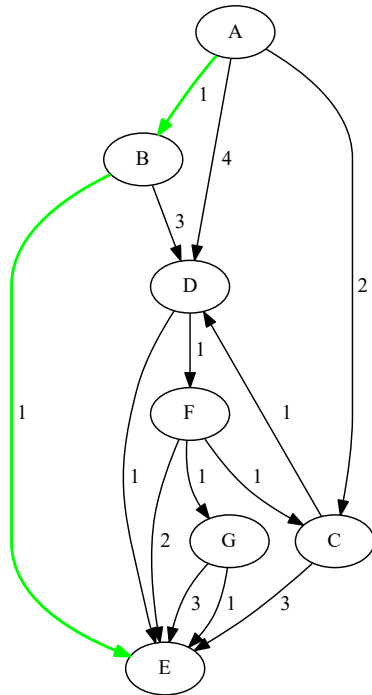
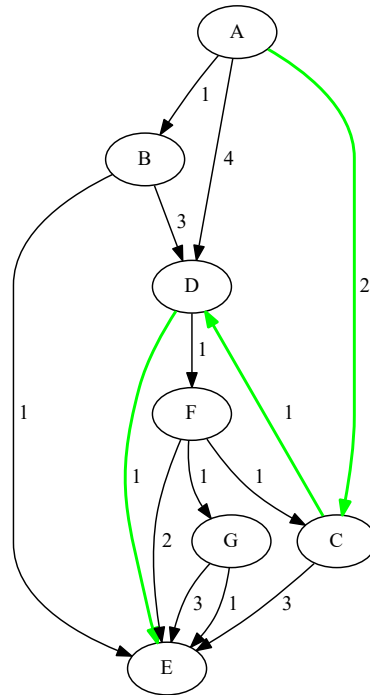
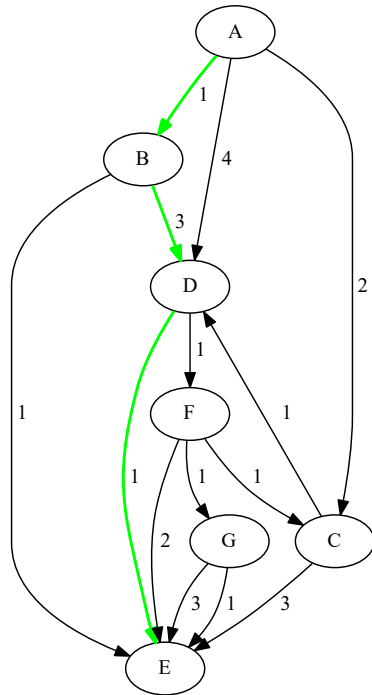
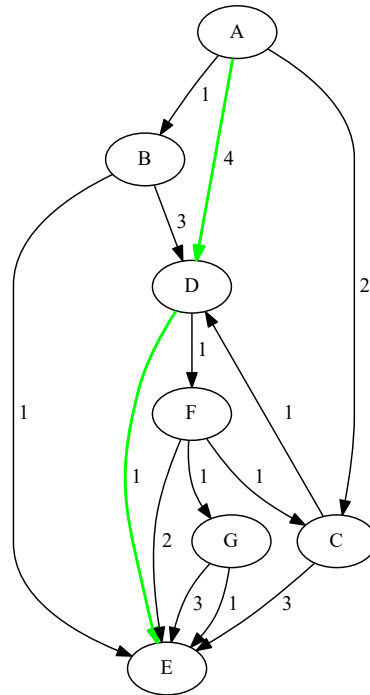
Figure 2.192 Rank 1 and 2 Paths from A to E $A \rightarrow B \rightarrow E$
path_weight=2 $A \rightarrow C \rightarrow D \rightarrow E$
path_weight=4

Figure 2.193 Rank 3 and 4 Paths from *A* to *E*

$A \rightarrow B \rightarrow D \rightarrow E$
path_weight=5



$A \rightarrow D \rightarrow E$
path_weight=5



The **MAXRELOBJGAP**= option specifies the maximum acceptable relative difference between the sum of link weights in the shortest path and the sum of link weights in each alternative path. In the previous example, you found the four shortest paths between nodes *A* and *E* with respect to the sum of link weights. For the shortest path, path_weight=2. Specifying **MAXRELOBJGAP**=1.0 removes any paths that are more than 100% worse than the shortest path.

The following statements find the (at most) four shortest paths between the source node *A* and the sink node *E* such that each path is within 100% of the shortest path:

```
proc network
  direction      = directed
  links          = mylib.LinkSetIn;
  shortestPath
    source       = A
    sink         = E
    maxPathsPerPair = 4
    maxRelObjGap  = 1.0
    outWeights   = mylib.ShortPathW;
run;
```

The output data table mylib.ShortPathW contains the path weights of the acceptable ranked paths, as shown in Figure 2.194.

Figure 2.194 Path Weights of Acceptable Ranked Paths for One Source-Sink Pair

source	sink	rank	path_weight
A	E	1	2
A	E	2	4

Shortest Paths through a Sequence of Nodes

This section illustrates how to use the shortest path algorithm to find the shortest path through a sequence of nodes.

Consider again the directed graph G shown in [Figure 2.189](#). In the previous example, you calculate the shortest path from A to E . Consider the case where you start at A and need to visit F and then D before ending at E .

You can represent the sequence of nodes to visit by using the following nodes subset data table, `mylib.NodeSubSetIn`:

```
data mylib.NodeSubSetIn;
    input node $ sequence;
    datalines;
A 1
F 2
D 3
E 4
;
```

The following statements find the shortest path through the sequence of nodes A , F , D , E :

```
proc network
    direction      = directed
    links           = mylib.LinkSetIn
    nodesSubset     = mylib.NodeSubSetIn;
    shortestPath
        sequence    = sequence
        outPathsLinks = mylib.ShortPathP
        outPathsNodes = mylib.ShortPathN;
run;
```

The output data table `mylib.ShortPathP` contains the links of this shortest path, as shown in [Figure 2.195](#).

Figure 2.195 Shortest Path Links through a Sequence of Nodes

source	sink	order	from	to	weight
A	E	1	A	C	2
A	E	2	C	D	1
A	E	3	D	F	1
A	E	4	F	C	1
A	E	5	C	D	1
A	E	6	D	E	1

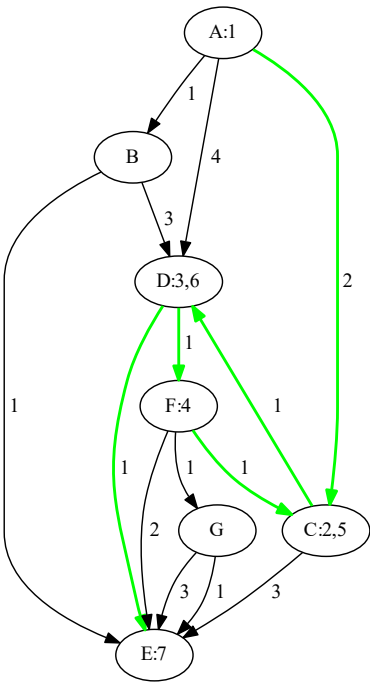
The output data table `mylib.ShortPathN` contains the nodes of this shortest path, as shown in [Figure 2.196](#).

Figure 2.196 Shortest Path Nodes through a Sequence of Nodes

source	sink	order	node
A	E	1	A
A	E	2	C
A	E	3	D
A	E	4	F
A	E	5	C
A	E	6	D
A	E	7	E

The shortest path is shown graphically in [Figure 2.197](#). The number next to each node label represents the order in which the node is visited in the path.

Figure 2.197 Shortest Path through the Sequence of Nodes *A, F, D, E*



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 59.

Output Data Tables

The summary statistics that PROC NETWORK produces are divided into two categories: statistics on the entire graph and statistics on the nodes and links of the graph. The latter statistics are appended to the output nodes and links data tables that you specify in the OUTNODES= and OUTLINKS= option in the PROC NETWORK statement. The former statistics are contained in the data table that you specify in the OUT= option in the SUMMARY statement.

In an undirected graph, $\hat{N}_i$ represents the set of neighbors, excluding node i itself (that is, unique nodes that are connected by links, excluding self-links, incident with node i). In a directed graph, $\hat{N}_i^{\text{out}}$ represents the set of out-neighbors, excluding node i itself (that is, unique nodes that are connected by outgoing links, excluding self-links, from node i), and $\hat{N}_i^{\text{in}}$ represents the set of in-neighbors, excluding node i itself (that is, unique nodes that are connected by incoming links, excluding self-links, to node i).

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 ($|E|$)
- 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 $\left(\frac{|E|}{K(N)}\right)$
- self_links_ignored: the number of self-links that are ignored
- dup_links_ignored: the number of links removed in multilink aggregation
- leaf_nodes: the number of leaf nodes
 - for an undirected graph: a node i is a leaf node if $|\hat{N}_i| = 1$
 - for a directed graph: a node i is a leaf node if $|\hat{N}_i^{\text{out}}| = 0$ and $|\hat{N}_i^{\text{in}}| > 0$
- singleton_nodes: the number of singleton nodes
 - for an undirected graph: a node i is a singleton node if $|\hat{N}_i| = 0$
 - a directed graph: a node i is a singleton node if $|\hat{N}_i^{\text{out}}| + |\hat{N}_i^{\text{in}}| = 0$

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 137 and “[Biconnected Components and Articulation Points](#)” on page 94, 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 in which one node $i \in C$ has $|\hat{N}_i| = |C| - 1$ and all other nodes $i \in C \setminus \{i\}$ have $|\hat{N}_i| = 1$)

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 in which one node $i \in C$ has $|\hat{N}_i^{\text{out}}| = |C| - 1$ and all other nodes $i \in C \setminus \{i\}$ have $|\hat{N}_i^{\text{in}}| = 1$)
- **isolated_stars_in**: the number of isolated inward stars (a weakly connected component, C , of size greater than 2 in which one node $i \in C$ has $|\hat{N}_i^{\text{in}}| = |C| - 1$ and all other nodes $i \in C \setminus \{i\}$ have $|\hat{N}_i^{\text{out}}| = 1$)

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 the graph can contain. For more information about shortest paths, see the section “[Shortest Path](#)” on page 226. If you use the SHORTESTPATH= option, the following columns also appear in the summary output data table:

- **diameter_wt**: the longest weighted shortest path distance in the graph
- **diameter_unwt**: the longest unweighted shortest path distance in the graph
- **avg_shortpath_wt**: the average weighted shortest path distance in the graph
- **avg_shortpath_unwt**: the average unweighted shortest path distance 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 $O(|N| \times (|N| \log |N| + |E|))$; the approximate method runs in time $O(|E| \sqrt{|N|})$ 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**: the approximate longest weighted shortest path distance in the graph

- `diameter_approx_unwt`: the approximate longest unweighted shortest path distance in the graph

You can produce statistics about clustering coefficients by using the `CLUSTERINGCOEFFICIENT` option. The *triangle count* of an undirected graph is the number of distinct three-node sets in which each node is a neighbor of the other two. Triangle counting is also used in the formula for computing clustering coefficients. For more information about clustering coefficients, see the section “[Clustering Coefficient](#)” on page 105. If you use the `CLUSTERINGCOEFFICIENT` option, the following column also appears in the summary output data table:

- `triangles`: the total triangle count of the graph

The assortativity of the graph measures the tendency of similar nodes to be connected in the graph. In a graph that has positive assortativity (or homophily), links tend to connect similar nodes. In a graph that has negative assortativity (or heterophily), links tend to connect dissimilar nodes. The assortativity coefficient is between -1 and 1 , where 1 indicates a strong positive relationship and -1 indicates a strong negative relationship. An assortativity of 0 shows no bias toward homophily or heterophily. This would be expected in a graph that has random connections. You can examine the similarity of nodes either in terms of their degrees or in terms of any numeric or nominal attributes. You can find details about each algorithm in Newman (2002, 2003).

The degree assortativity and the numeric assortativity are computed by using the Pearson correlation coefficient formula:

$$r = \frac{\sum_{e \in E} (x_e - \bar{x})(y_e - \bar{y})}{\sqrt{\sum_{e \in E} (x_e - \bar{x})^2 \sum_{e \in E} (y_e - \bar{y})^2}}$$

where x and y are vectors of size $|E|$ of node attributes defined at the terminals of each link in E . Note that if x or y is constant over all links, the denominator is 0 . In this case, the assortativity is assigned a missing value.

The most common way to compute assortativity is based on node degrees. You can produce statistics about the degree assortativity of the graph by using the `ASSORTATIVITYDEGREE=` option. You can compute the degree assortativity by using the degrees of nodes at either end of each link.

For an undirected graph, each edge is replaced by two directed edges that point in opposite directions. Then, x and y are constructed as follows:

$$\begin{aligned} x_e &= C_d(\text{from}(e)) = \sum_{e' \in \delta_{\text{from}(e)}} w_{e'}^E \\ y_e &= C_d(\text{to}(e)) = \sum_{e' \in \delta_{\text{to}(e)}} w_{e'}^E \end{aligned}$$

where $C_d(i)$ is the degree centrality of node i , $\text{from}(e)$ represents the *from* node of link e , $\text{to}(e)$ represents the *to* node of link e , and w_e^E is the weight of link e . For unweighted graphs, the default weight of each link is 1 . If the graph is undirected and you specify `ASSORTATIVITYDEGREE=WEIGHT` or `ASSORTATIVITYDEGREE=BOTH`, the following column also appears in the summary output data table:

- `deg_assort_wt`: the degree assortativity computed by using the weighted degree

If the graph is undirected and you specify `ASSORTATIVITYDEGREE=UNWEIGHT` or `ASSORTATIVITYDEGREE=BOTH`, the following column also appears in the summary output data table:

- `deg_assort_unwt`: the degree assortativity computed by using the unweighted degree

For a directed graph, x and y have two options, the nodes' in-degree and out-degree, which are computed as follows:

$$\begin{aligned} x_e = C_d^{\text{in}}(\text{from}(e)) &= \sum_{e' \in \delta_{\text{from}(e)}^{\text{in}}} w_{e'}^E & \text{or} & & x_e = C_d^{\text{out}}(\text{from}(e)) &= \sum_{e' \in \delta_{\text{from}(e)}^{\text{out}}} w_{e'}^E \\ y_e = C_d^{\text{in}}(\text{to}(e)) &= \sum_{e' \in \delta_{\text{to}(e)}^{\text{in}}} w_{e'}^E & \text{or} & & y_e = C_d^{\text{out}}(\text{to}(e)) &= \sum_{e' \in \delta_{\text{to}(e)}^{\text{out}}} w_{e'}^E \end{aligned}$$

where $C_d^{\text{in}}(i)$ is the in-degree centrality of node i , $C_d^{\text{out}}(i)$ is the out-degree centrality of node i , $\text{from}(e)$ represents the *from* node of link e , $\text{to}(e)$ represents the *to* node of link e , and w_e^E is the weight of link e . For unweighted graphs, the default weight of each link is 1.

If the graph is directed and you specify `ASSORTATIVITYDEGREE=WEIGHT` or `ASSORTATIVITYDEGREE=BOTH`, the following columns also appear in the summary output data table:

- `deg_assort_wt_in_in`: the degree assortativity computed by using the weighted in-degree for the source and the weighted in-degree for the target
- `deg_assort_wt_in_out`: the degree assortativity computed by using the weighted in-degree for the source and the weighted out-degree for the target
- `deg_assort_wt_out_in`: the degree assortativity computed by using the weighted out-degree for the source and the weighted in-degree for the target
- `deg_assort_wt_out_out`: the degree assortativity computed by using the weighted out-degree for the source and the weighted out-degree for the target

If the graph is directed and you specify `ASSORTATIVITYDEGREE=UNWEIGHT` or `ASSORTATIVITYDEGREE=BOTH`, the following columns also appear in the summary output data table:

- `deg_assort_unwt_in_in`: the degree assortativity computed by using the unweighted in-degree for the source and the unweighted in-degree for the target
- `deg_assort_unwt_in_out`: the degree assortativity computed by using the unweighted in-degree for the source and the unweighted out-degree for the target
- `deg_assort_unwt_out_in`: the degree assortativity computed by using the unweighted out-degree for the source and the unweighted in-degree for the target
- `deg_assort_unwt_out_out`: the degree assortativity computed by using the unweighted out-degree for the source and the unweighted out-degree for the target

You can produce statistics about the numeric assortativity of the graph by using the `ASSORTATIVITYNUMERIC=` option. The Pearson correlation coefficient formula is used to compute the numeric assortativity of

the graph by using a numeric node attribute. In the case of undirected graphs, each link is replaced by two directed links that point in opposite directions. Then, x and y are constructed as follows:

$$\begin{aligned}x_e &= \beta_{\text{from}(e)} \\ y_e &= \beta_{\text{to}(e)}\end{aligned}$$

where $\text{from}(e)$ represents the *from* node of link e , $\text{to}(e)$ represents the *to* node of link e , and β_i is the numeric attribute of node i specified by the ASSORTATIVITYNUMERIC= option. If you use the ASSORTATIVITYNUMERIC= option for a graph, the following column also appears in the summary output data table:

- num_assort: the numeric assortativity computed by using the specified numeric column

You can produce statistics about the nominal assortativity of the graph by using the ASSORTATIVITYNOMINAL= option. To compute the nominal assortativity, the attribute mixing matrix M is constructed, where each element M_{ij} represents the fraction of links that connect nodes whose attribute value is i to nodes whose attribute value is j . Note that in the case of undirected graphs, each link, including self-links, is replaced by two directed links that point in opposite directions.

Subsequently, a_i and b_j are defined as the fractions of all the links that have source nodes whose attribute value is i or target nodes whose attribute value is j , respectively. They are computed as follows:

$$\begin{aligned}a_i &= \sum_j M_{ij} \\ b_j &= \sum_i M_{ij}\end{aligned}$$

Then, the following formula computes the nominal assortativity of the graph:

$$r = \frac{\sum_i M_{ii} - \sum_i a_i b_i}{1 - \sum_i a_i b_i}$$

If you use the ASSORTATIVITYNOMINAL= option for a graph, the following column also appears in the summary output data table:

- nom_assort: the nominal assortativity computed by using the specified nominal column

OUTNODES= Data Table

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

- sum_in_and_out_wt: the sum of the link weights from and to the node
- leaf_node: 1, if the node is a leaf node; otherwise, 0
- singleton_node: 1, if the node is a singleton node; otherwise, 0

- `isolated_pair`: the identifier, if the node is in an isolated pair; otherwise, missing (.)
- `neighbor_leaf_nodes`: the number of leaf nodes connected to the node

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

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

The following columns also appear for directed graphs:

- `isolated_star_out`: the identifier, if the node is in an isolated outward star; otherwise, missing (.)
- `isolated_star_in`: the identifier, if the node is in an isolated inward star; otherwise, missing (.)

You can produce statistics about the shortest path distances to and from nodes in the graph by using the `SHORTESTPATH=` option. The *eccentricity* of a node i is the longest of all possible shortest path distances between i and any other node. If you use the `SHORTESTPATH=` option, the following columns also appear in the nodes output data table for undirected graphs:

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

The following columns also appear for directed graphs:

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

OUTLINKS= Data Table

In addition, you can produce summary statistics about the connectedness of the links of the graph. If you use the `CONNECTEDCOMPONENTS` or `BICONNECTEDCOMPONENTS` option, the following columns are appended to the data table that you specify in the `OUTLINKS=` option in the `PROC NETWORK` statement for undirected graphs:

- `isolated_pair`: the identifier, if the link is in an isolated pair; otherwise, missing (.)
- `isolated_star`: the identifier, if the link is in an isolated star; otherwise, missing (.)

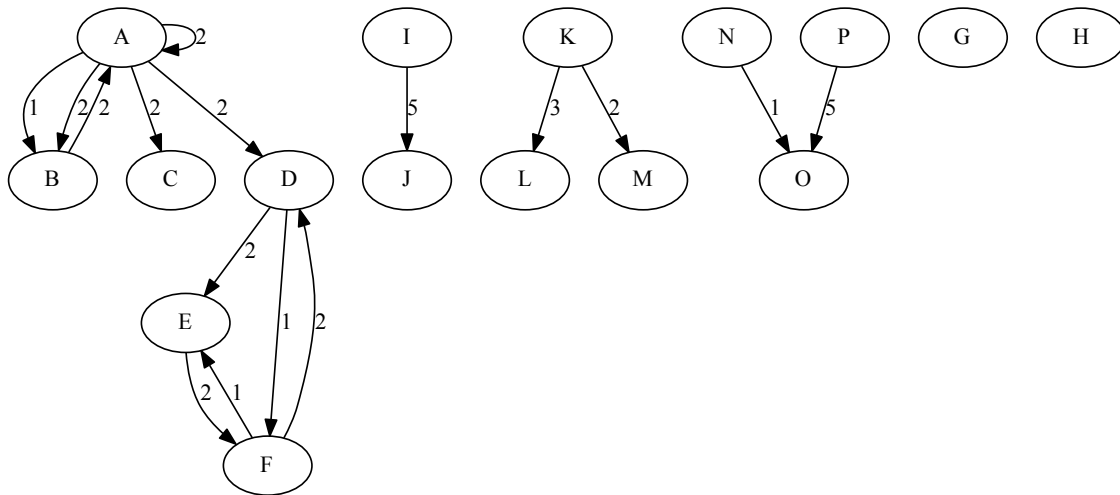
The following columns are appended for directed graphs:

- `isolated_star_out`: the identifier, if the link is in an isolated outward star; otherwise, missing (.)
- `isolated_star_in`: the identifier, if the link is in an isolated inward star; otherwise, missing (.)

Summary Statistics of a Directed Graph

This section illustrates the calculation of summary statistics on the directed graph G shown in Figure 2.198.

Figure 2.198 Directed Graph G



You can represent the directed graph G by using the following nodes data table, mylib.NodeSetIn, and links data table, mylib.LinkSetIn:

```
data mylib.NodeSetIn;
  input node $ @@;
  datalines;
A B C D E F G H I J K L M N O P
;
data mylib.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
;
```

Basic Summary Statistics

The following statements calculate the default summary statistics and output the results in the data table mylib.Summary:

```
proc network
  direction = directed
  nodes      = mylib.NodeSetIn
  links      = mylib.LinkSetIn;
  summary
    out      = mylib.Summary;
```

```
run;
```

The output data table mylib.Summary contains the default summary statistics of the input graph, as shown in Figure 2.199.

Figure 2.199 Graph Summary Statistics of a Directed Graph

nodes	links	avg_links_per_node	density	self_links_ignored	dup_links_ignored	leaf_nodes	singleton_nodes
16	16	1	0.066667	0	0	5	2

Basic Summary Statistics and Connected Components

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 mylib.Summary.

```
proc network
  direction = directed
  nodes     = mylib.NodeSetIn
  links     = mylib.LinkSetIn;
  summary
    connectedComponents
  out      = mylib.Summary;
run;
```

The output data table mylib.Summary contains the summary statistics of the input graph, as shown in Figure 2.200.

Figure 2.200 Graph Summary and Connectedness Statistics of a Directed Graph

nodes	links	avg_links_per_node	density	self_links_ignored	dup_links_ignored	leaf_nodes	singleton_nodes
16	16	1	0.066667	0	0	5	2

concomp	isolated_pairs	isolated_stars_out	isolated_stars_in
13	1	1	1

Basic Summary Statistics and Shortest Paths

The following statements calculate the default summary statistics and produce information about the shortest path distances of the graph. They output the results in the data table mylib.Summary. In addition, node statistics are produced and output in the data table mylib.NodeSetOut. Because the graph is disconnected, use the FINITEPATH option in the SUMMARY statement so that the shortest path descriptive statistics only consider finite paths.

```
proc network
  direction      = directed
  nodes          = mylib.NodeSetIn
  links          = mylib.LinkSetIn
  outNodes       = mylib.NodeSetOut;
  summary
    finitePath
    out          = mylib.Summary
    shortestPath = weight;
run;
```

The output data table `mylib.Summary` contains the summary and shortest path statistics of the input graph, as shown in Figure 2.201.

Figure 2.201 Graph Summary and Shortest Path Statistics of an Directed Graph

nodes	links	avg_links_per_node	density	self_links_ignored	dup_links_ignored	leaf_nodes
16	16	1	0.066667	0	0	5

singleton_nodes	diameter_wt	avg_shortpath_wt
2	6	2.90476

The output data table `mylib.NodeSetOut` contains per-node summary and shortest path statistics of the input graph, as shown in Figure 2.202.

Figure 2.202 Per-Node Summary and Shortest Path Statistics of an Directed Graph

node	leaf_node	singleton_node	neighbor_leaf_nodes	sum_in_and_out_wt	eccentr_wt_out	eccentr_wt_in
A	0	0	1	13	4	2
B	0	0	0	5	6	1
C	1	0	0	2	0	4
D	0	0	0	7	2	4
E	0	0	0	5	4	6
F	0	0	0	6	2	5
G	0	1	0	0	0	0
H	0	1	0	0	0	0
I	0	0	1	5	5	0
J	1	0	0	5	0	5
K	0	0	2	5	3	0
L	1	0	0	3	0	3
M	1	0	0	2	0	2
N	0	0	1	1	1	0
O	1	0	0	6	0	5
P	0	0	1	5	5	0

Topological Sort

A *topological ordering* of a directed graph $G = (N, E)$ is a linear ordering of its nodes such that, for every directed link $(i, j) \in E$, node i comes before node j in the ordering. A topological ordering is commonly used to handle workflow dependencies. For example, consider scheduling a set of tasks that have dependencies. You can build a directed graph in which each node represents a task and each link (i, j) represents the requirement that task i be completed before task j . A topological ordering of this graph would give you a valid sequence of tasks.

A topological ordering exists if and only if the graph contains no directed cycles.

In PROC NETWORK, you can calculate a topological ordering of the nodes of a directed acyclic input graph by using the **TOPOLOGICALSORT** statement.

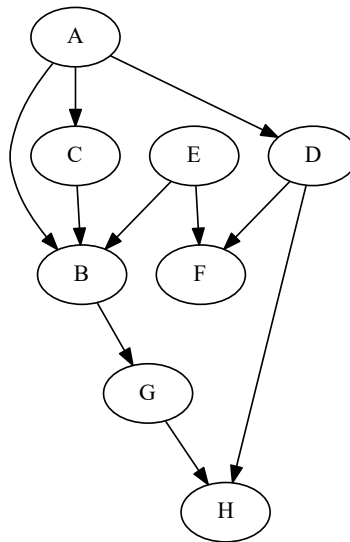
The results of the topological sort algorithm are written to the output nodes data table that you specify in the OUTNODES= option in the PROC NETWORK statement. For each node in the nodes data table, the variable top_order identifies the order in the topological ordering.

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

Topological Ordering of a Directed Graph

This section illustrates the use of the topological sort on the directed graph G shown in Figure 2.203.

Figure 2.203 Directed Graph G



The directed graph G can be represented by the following links data table, mylib.LinkSetInTopSort:

```

data mylib.LinkSetInTopSort;
  input from $ to $ @@;
  datalines;
A B  A C  A D  E B  D F
E F  C B  B G  G H  D H
;

```

The following statements calculate a topological ordering for G and output the results in the data table mylib.NodeSetOut:

```

proc network
  direction = directed
  links      = mylib.LinkSetInTopSort
  outNodes   = mylib.NodeSetOut;
  topologicalSort;
run;

```

The output data table `mylib.NodeSetOut` contains a topological ordering of the input graph, as shown in Figure 2.204.

Figure 2.204 Topological Ordering of a Directed Graph

node	top_order
E	1
A	2
D	3
F	4
C	5
B	6
G	7
H	8

Transitive Closure

The *transitive closure* of a graph G is a graph $G^T = (N, E^T)$ such that for all $i, j \in N$ there is a link $(i, j) \in E^T$ 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.

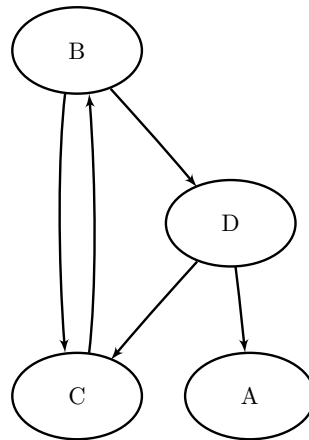
In PROC NETWORK, you can invoke the transitive closure algorithm by using the TRANSITIVECLOSURE statement. The options for this statement are described in the section “[TRANSITIVECLOSURE Statement](#)” on page 61.

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

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

Transitive Closure of a Directed Graph

This example illustrates the use of the transitive closure algorithm on the directed graph G shown in Figure 2.205.

Figure 2.205 Directed Graph *G*

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

```

data mylib.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 mylib.TransClosure:

```

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

```

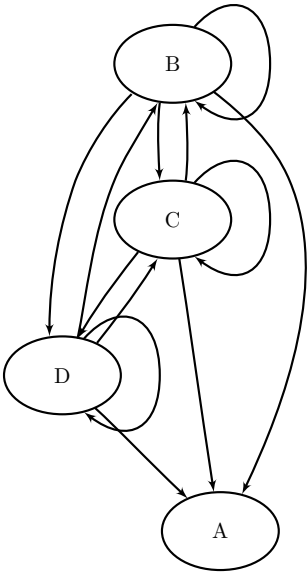
The output data table mylib.TransClosure contains the transitive closure of *G*, as shown in [Figure 2.206](#).

Figure 2.206 Transitive Closure of a Directed Graph

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

The transitive closure of G is shown graphically in Figure 2.207.

Figure 2.207 Transitive Closure of G



For a more detailed example, see “[Example 2.10: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System](#)” on page 300.

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, listed here in the order in which they appear in the macro, are interpreted as follows:

STATUS

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

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

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
LOADGRAPH	Loading a graph (no algorithm)
NODESIMILARITY	Node similarity
PATH	Path enumeration
PATTERNMATCH	Pattern matching
PROJECTION	Network projection
REACH	Reach (ego) networks
READGRAPH	Reading a graph (no algorithm)
SHORTESTPATH	Shortest path
SUMMARY	Graph summary
TOPOLOGICALSORT	Topological sorting
TRANSITIVECLOSURE	Transitive closure
UNLOADGRAPH	Unloading a graph (no algorithm)

SOLUTION_STATUS

indicates the solution status of the selected problem type (algorithm class). The `SOLUTION_STATUS` term can take one of the following values:

ERROR	The algorithm encountered an error.
INTERRUPTED	The algorithm was interrupted by the user.
OK	The algorithm terminated normally.
OUTMEM_NOSOL	The algorithm ran out of memory and either did not find a solution or failed to output the solution due to insufficient memory.
OUTMEM_SOL	The algorithm ran out of memory but still found a solution.
SOLUTION_LIM	The algorithm reached its limit on the number of solutions found.
TIMELIMIT	The algorithm reached its execution time limit.

CPU_TIME

indicates the total CPU time (in seconds) that `PROC NETWORK` used.

REAL_TIME

indicates the elapsed time (in seconds) that `PROC NETWORK` used.

In addition, each algorithm might report some additional details. The following section provides more information about these details.

Macro Variable **_NETWORK_** Details

The BICONNECTEDCOMPONENTS algorithm provides the following additional information:

NUM_COMPONENTS

indicates the number of biconnected components that the algorithm found.

NUM_ARTICULATION_POINTS

indicates the number of articulation points that the algorithm found.

The CENTRALITY algorithm provides the following additional information:

NUM_TRIANGLES

indicates the number of triangles in the graph (if you specify CLUSTERINGCOEFFICIENT and DIRECTION=UNDIRECTED).

The CLIQUE algorithm provides the following additional information:

NUM_CLIQUES

indicates the number of cliques that the algorithm found.

CLIQUENUMBER

indicates the maximum size of the cliques that the algorithm found (if you specify CLIQUENUMBER=TRUE).

The CONNECTEDCOMPONENTS algorithm provides the following additional information:

NUM_COMPONENTS

indicates the number of connected components that the algorithm found.

The CYCLE algorithm provides the following additional information:

NUM_CYCLES

indicates the number of cycles that the algorithm found.

The LOADGRAPH statement provides the following additional information:

CREATETIME

indicates the creation time of the in-memory graph.

GRAPH

indicates the in-memory graph identifier.

The PATH algorithm provides the following additional information:

NUM_PATHS

indicates the number of paths that the algorithm found.

The PATTERNMATCH algorithm provides the following additional information:

NUM_MATCHES

indicates the number of pattern matches that the algorithm found.

The SHORTESTPATH algorithm provides the following additional information:

NUM_PATHS

indicates the number of shortest paths that the algorithm found.

ODS Table Names

For general information about ODS tables, see *SAS Output Delivery System: Procedures Guide*. Each ODS table that the NETWORK procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. These names are listed in Table 2.16.

Table 2.16 ODS Tables Produced by PROC NETWORK

Table Name	Description
ProblemSummary	Summary of the graph
SolutionSummary	Summary of the solution status, timing, and results
OutputCasTables	See the section “ OutputCasTables Table ” on page 262

The following statements use the example in the section “[Shortest Paths for All Pairs](#)” on page 229 and find all-pairs shortest paths for a small undirected graph. By default, this code produces the two ODS output tables listed in Table 2.16.

```
data mylib.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          = mylib.LinkSetIn;
    shortestPath
        outWeights  = mylib.ShortPathW
        outPathsLinks = mylib.ShortPathP;
run;
```

The problem summary table in Figure 2.208 provides a basic summary of the graph input.

Figure 2.208 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 2.209](#) provides a basic solution summary for the algorithm that is processed. The information in this table is similar to the information that is provided in the macro variable `_NETWORK_`, described in the section “[Macro Variable _NETWORK_](#)” on page 258. The timing information in this table (and in the log) represents the time spent running the algorithm, excluding the time spent in input, graph building, and output. In the case of reading, loading, or unloading a graph (with no algorithm), the time in the solution summary represents the input and graph building time. In the case of a distributed algorithm, which uses multiple machines, the real time represents the maximum amount of time that an individual machine used to run the algorithm, and the CPU time represents the total amount of time across all active machines in your configured session.

Figure 2.209 Solution Summary Table

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

OutputCasTables Table

The OutputCasTables table is a special table that has information about each CAS table that is created during a CAS action execution. The information for each CAS table consists of the CAS table name, the caslib in which the table resides, and the number of columns and rows in the CAS table. Because this table is not a typical ODS table that contains analytical results, you cannot include it in the *table-spec-list* in the `DISPLAYOUT` statement.

The %PlotGraph Visualization Macro

The `%PlotGraph` macro uses the *sasviya.network* Python package to visualize a graph. For instructions on configuring Python for use with SAS, see [SAS Viya Platform: Integration with External Languages](#). The macro constructs a graph object, applies a chosen layout, and draws nodes and links, with flexible styling options for you to choose from.

The following sections describe all available options, which are divided into six categories.

Data Table Options

The following options enable you to specify the data tables for the `%PlotGraph` macro to read when it constructs the graph. You must specify either the `LINKS=` option or the `NODES=` option. You can also specify both options.

LINKS=*libref.data-table*

EDGES=*libref.data-table*

specifies the input data table that contains the graph link (edge) information. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the section “[Using CAS Sessions and CAS Engine Librefs](#)” on page 2 in Chapter 1, “[Introduction](#).”

NODES=libref.data-table

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

Column Mapping Options

The following options enable you to explicitly specify the data variable names for the %PlotGraph macro to use when it reads the data tables that you specify in the **LINKS=** and **NODES=** options.

FROM=column**FROMVAR=column**

specifies the name of the column in the links data table that contains the source of each link. By default, FROM=from.

TO=column**TOVAR=column**

specifies the name of the column in the links data table that contains the target of each link. By default, TO=to.

NODE=column

specifies the name of the column in the nodes data table that contains the node identifier. By default, NODE=node.

Graph Semantics and Preprocessing Options

You can specify the following options to modify the graph semantics and preprocessing:

DIRECTION=DIRECTED | UNDIRECTED

specifies whether the input graph should be considered to be directed or undirected. By default, DIRECTION=UNDIRECTED. For more information, see the section “Graph Input and Output Data” on page 61.

MULTILINKS=TRUE | FALSE

specifies whether to include (TRUE) or to aggregate (FALSE) multilinks when the links data table is read. By default, MULTILINKS=TRUE. For more information about this option, see the section “Multigraphs” on page 68.

SELFLINKS=TRUE | FALSE

specifies whether to include (TRUE) or to remove (FALSE) self-links when the links data table is read. By default, SELFLINKS=TRUE. For more information about this option, see the section “Self-Links” on page 71.

Link Styling Options

You can specify the following options to customize the appearance of the links in the graph:

ARROWSIZE=*number* | *column*

specifies a single size or the name of the column in the links data table that contains the size of each arrow. By default, ARROWSIZE=20. For undirected graphs, this parameter is ignored.

LINKSCOLOR=*string* | *column*

EDGESCOLOR=*string* | *column*

specifies a single color or the name of the column in the links data table that contains the color of each link. By default, LINKSCOLOR=*black*. The supported colors are *blue*, *green*, *red*, and any color that is listed here: https://matplotlib.org/3.3.1/gallery/color/named_colors.html. Hexadecimal color codes, such as #FF0000 for *red*, are also supported.

LINKSCOLORBYCATEGORY=*column*

EDGESCOLORBYCATEGORY=*column*

specifies the name of the column in the links data table to be used to color links according to the column's distinct categories.

LINKSCOLORBYVALUE=*column*

EDGESCOLORBYVALUE=*column*

specifies the name of the column in the links data table to be used to color links according to the column's numeric values.

LINKSCOLORLEGEND=**TRUE** | **FALSE**

EDGESCOLORLEGEND=**TRUE** | **FALSE**

specifies whether to display (TRUE) or not to display (FALSE) a color legend when links are colored by either category or value — that is, when you specify the LINKSCOLORBYCATEGORY= or LINKSCOLORBYVALUE= option, respectively. By default, LINKSCOLORLEGEND=FALSE.

LINKSLABEL=*column*

EDGESLABEL=*column*

specifies the name of the column in the links data table that contains the label of each link.

LINKSSTYLE=*string* | *column*

EDGESSTYLE=*string* | *column*

specifies a single style or the name of the column in the links data table that contains the style of each link. The following styles are supported: *dash*, *dashdot*, *dot*, and *solid*. By default, LINKSSTYLE=*solid*.

LINKSWIDTH=*number* | *column*

EDGESWIDTH=*number* | *column*

specifies a single width or the name of the column in the links data table that contains the width of each link. By default, LINKSWIDTH=1.

Node Styling Options

You can specify the following options to customize the appearance of the nodes in the graph:

NODESBORDERCOLOR=*string* | *column*

specifies a single color or the name of the column in the nodes data table that contains the color of each node's border. By default, NODESBORDERCOLOR=*black*. For supported colors, see the [LINKSCOLOR=](#) option.

NODESCOLOR=*string* | *column*

specifies a single color or the name of the column in the nodes data table that contains the color of each node. By default, NODESCOLOR=*white*. For supported colors, see the [LINKSCOLOR=](#) option.

NODESCOLORBYCATEGORY=*column*

specifies the name of the column in the nodes data table to be used to color nodes according to the column's distinct categories.

NODESCOLORBYVALUE=*column*

specifies the name of the column in the nodes data table to be used to color nodes according to the column's numeric values.

NODESCOLORLEGEND=TRUE | FALSE

specifies whether to display (TRUE) or not to display (FALSE) the color legend when nodes are colored by either category or value — that is, when you specify the NODESCOLORBYCATEGORY= or NODESCOLORBYVALUE= option, respectively. By default, NODESCOLORLEGEND=FALSE.

NODESLABEL=*column*

specifies the name of the column in the nodes data table that contains the label to display for each node. By default, the value of *column* is the same as the value that you specify in the [NODE=](#) option.

NODESSHAPE=*string* | *column*

specifies a single shape or the name of the column in the nodes data table that contains the shape of each node. The following shapes are supported: *e* or *ellipse*, *h* or *hexagon*, *o* or *circle*, *p* or *pentagon*, *s* or *square*, *^* or *triangleup*, *v* or *triangledown*, *<* or *triangleleft*, and *>* or *triangleright*. By default, NODESSHAPE=*ellipse*.

NODESSIZE=*number* | *column*

specifies a single size or the name of the column in the nodes data table that contains the size of each node. By default, NODESSIZE=500.

NODESX=*column*

specifies the name of the column in the nodes data table that contains the X coordinate of each node.

NODESY=*column*

specifies the name of the column in the nodes data table that contains the Y coordinate of each node.

Display Options

You can specify the following options to customize the figure display:

DISPLAYLABELS=TRUE | FALSE

specifies whether to display (TRUE) or not to display (FALSE) labels. By default, DISPLAYLABELS=TRUE.

DPI=number

specifies the resolution of the figure in dots per inch. The value of *number* can be any nonnegative number. By default, DPI=60.

HEIGHT=number

specifies the height of the figure in inches. The value of *number* can be any nonnegative number. By default, HEIGHT=8.

LAYOUT=string

specifies the plot layout. The following layouts are supported: *bipartite*, *circo*, *circular*, *dot*, *fdp*, *fruchterman_reingold*, *kamada_kawai*, *multipartite*, *neato*, *planar*, *random*, *sfdp*, *shell*, *spectral*, *spiral*, *spring*, and *twopi*. By default, LAYOUT=*kamada_kawai*.

The *circo*, *dot*, *fdp*, *neato*, *sfdp*, and *twopi* layouts require the pydot and Graphviz Python packages, as well as the Graphviz executables. These executables must be installed and accessible via the system path.

OUTFILE=string

specifies the full output path, file name, and file format to use for saving the plot; for example, OUTFILE='/path/to/your/graph.png'. By default, the plot is displayed but not saved. The following file formats are supported: .png and .jpg.

SCALEFACTOR=number

specifies the factor to linearly scale the position of nodes. By default, SCALEFACTOR=1. A value greater than 1 enlarges the graph, and a value between 0 and 1 shrinks the graph. This parameter is ignored when you specify both the **NODESX=** and **NODESY=** options.

SEED=number

specifies the random seed for layout algorithms that use randomness, where *number* can be any integer value.

TITLE=string

specifies the text to use for the title of the figure.

WIDTH=number

specifies the width of the figure in inches. The value of *number* can be any nonnegative number. By default, WIDTH=12.

Example 1: Simple Directed Graph

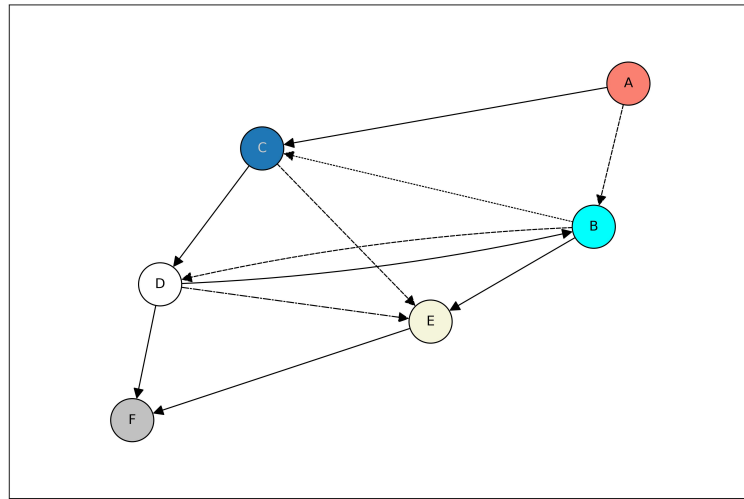
This example illustrates how to use the %PlotGraph macro to visualize a graph. You can represent the directed graph G by using the following links and nodes data tables:

```
data mylib.LinkSetIn;
  input from to style $;
  datalines;
1 2 dash
1 3 solid
2 3 dot
2 4 dash
4 2 solid
2 5 solid
3 4 solid
3 5 dash
4 5 dashdot
4 6 solid
5 6 solid
;

data mylib.NodeSetIn;
  input node label $ color $;
  datalines;
1 A salmon
2 B cyan
3 C tab:blue
4 D white
5 E beige
6 F silver
;
```

Then, you can use the %PlotGraph macro as follows to visualize the graph. The resulting graph is shown in [Output 2.1](#).

```
%PlotGraph(links      = mylib.LinkSetIn,
            nodes      = mylib.NodeSetIn,
            direction  = directed,
            nodeslabel = label,
            nodescolor = color,
            nodesshape = circle,
            nodessize  = 1500,
            linksstyle = style,
            dpi        = 500);
```

Output 2.1 Simple Directed Graph G 

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

Figure 2.210 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 mylib.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      = mylib.LinkSetInTerror911
  outNodes = mylib.NodeSetOut;
  biconnectedComponents;
run;

data mylib.ArtPoints;
  set mylib.NodeSetOut;
  where artpoint=1;
run;

```

The output data table `mylib.ArtPoints` contains members of the network who are articulation points. By focusing on cutting off these particular members, investigators could have significantly disrupted the terrorists' ability to communicate when planning the attack.

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

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

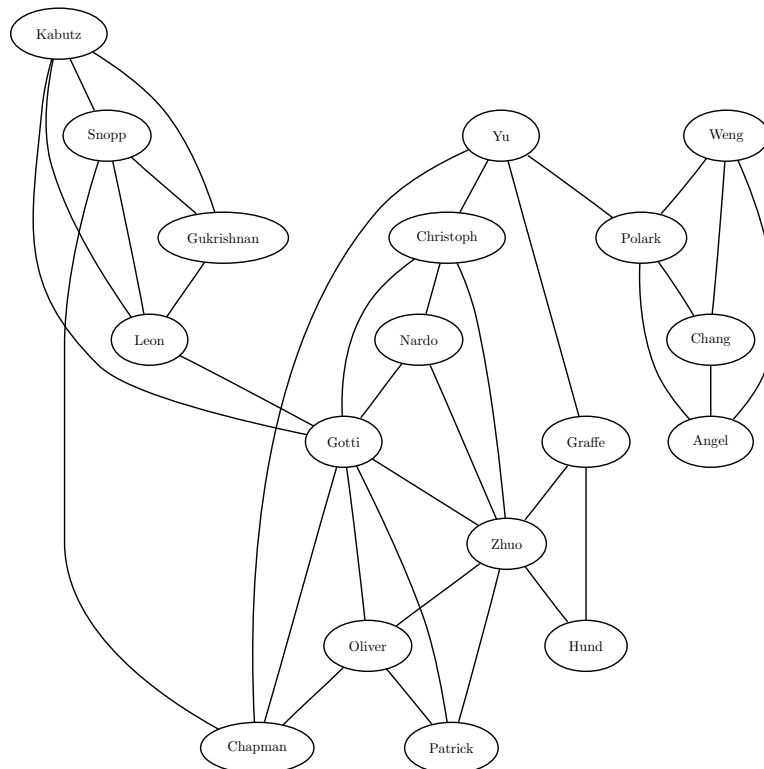
Example 2.2: Influence Centrality for Project Groups in a Research Department

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 2.211.

Figure 2.211 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 mylib.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 mylib.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 weighted influence centrality (in addition to unweighted degree centrality):

```

proc network
  logLevel      = moderate
  links         = mylib.LinkSetInDept
  nodes         = mylib.NodeSetInDept
  outNodes      = mylib.NodeSetOut;
  centrality
    degree      = unweight
    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 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 using 1 threads across 1 machines.
NOTE: Processing centrality metrics used 0.1 MBs of memory.
NOTE: Processing degree centrality metrics used 0.00 (cpu: 0.00) seconds.
NOTE: Processing influence centrality metrics using 1 threads across 1 machines.
NOTE: Processing centrality metrics used 0.1 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 0.074047 seconds.
NOTE: The data set MYLIB.NODESETOUT has 18 observations and 5 variables.
STATUS=OK  PROBLEM_TYPE=CENTRALITY  SOLUTION_STATUS=OK  CPU_TIME=0.71  REAL_TIME=0.07

```

The nodes data table mylib.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

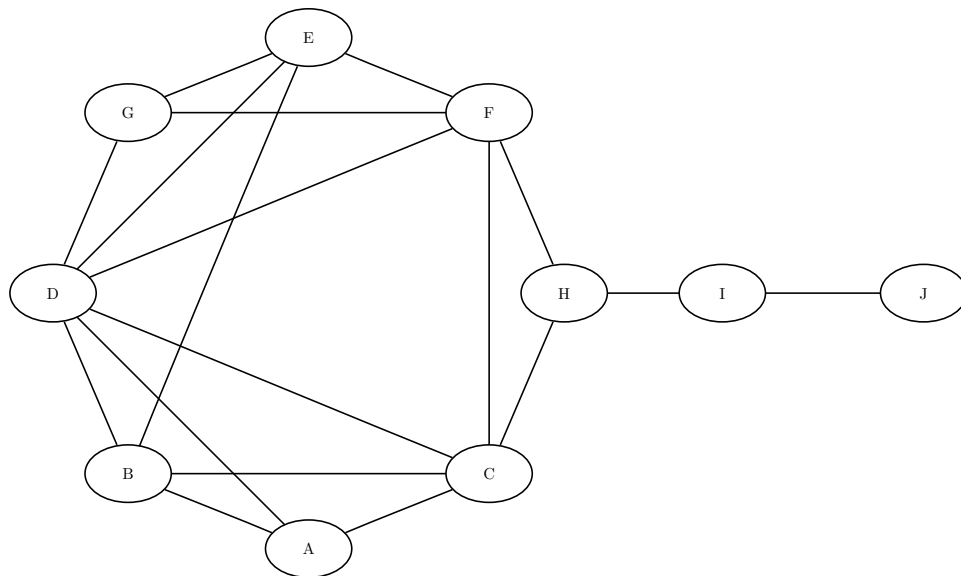
node	weight	centr_degree_out	centr_influence1_wt	centr_influence2_wt
Gotti	2	8	0.35714	1.57143
Zhuo	1	7	0.25000	1.17857
Oliver	2	4	0.21429	1.14286
Chapman	4	4	0.42857	1.10714
Christoph	2	4	0.17857	1.03571
Yu	3	4	0.32143	0.92857
Patrick	1	3	0.10714	0.82143
Kabutz	1	4	0.14286	0.82143
Snopp	2	4	0.21429	0.82143
Leon	1	4	0.14286	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
Weng	1	3	0.10714	0.39286
Angel	1	3	0.10714	0.39286
Chang	1	3	0.10714	0.39286
Hund	1	2	0.07143	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 2.212. This is an example of the well-known *kite network*, which was popularized by Krackhardt (1990) for better understanding of social networks in the workplace.

Figure 2.212 Office Computer Network



Define the links data table as follows:

```

data mylib.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      = mylib.LinkSetInCompNet
  outLinks    = mylib.LinkSetOut
  outNodes    = mylib.NodeSetOutCentr;
  centrality
    degree    = unweight
    close      = unweight
    between    = unweight;
run;
proc network
  links      = mylib.LinkSetInCompNet
  outNodes    = mylib.NodeSetOutBiCC;
  biconnectedComponents;
run;
data mylib.NodeSetOut;
  merge mylib.NodeSetOutCentr mylib.NodeSetOutBiCC;
  by node;
run;
%put &_NETWORK_;

```

Output 2.3.1 shows the resulting nodes data table mylib.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
G	3	0.50000	0.00000	0
A	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 (mylib.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
G	3	0.50000	0.00000	0
A	3	0.50000	0.00000	0
J	1	0.31034	0.00000	0

Output 2.3.3 shows the resulting links data table (mylib.LinkSetOut) sorted by link betweenness.

Output 2.3.3 Link Betweenness Centrality, Sorted by Betweenness

from	to	centr_between_unwt
H	I	0.35556
C	H	0.23333
F	H	0.23333
I	J	0.20000
B	C	0.10370
E	F	0.10370
A	C	0.10000
F	G	0.10000
C	D	0.07407
D	F	0.07407
A	D	0.06667
D	G	0.06667
B	E	0.05926
C	F	0.05926
B	D	0.04074
D	E	0.04074
A	B	0.03333
E	G	0.03333

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 271, 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 108, 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         = mylib.LinkSetInDept
  outLinks      = mylib.LinkSetOut
  outNodes      = mylib.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: Reading the links data.
NOTE: Data input used 0.00 (cpu: 0.00) seconds.
NOTE: Building the input graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 18.
NOTE: The number of links in the input graph is 35.
NOTE: Link weight values for betweenness and closeness centrality are automatically inverted.
      You can use the AUXWEIGHT= option in the LINKSVAR statement to specify the link weight
      values used by betweenness and closeness centrality.
NOTE: Processing centrality metrics.
NOTE: Processing between/close centrality metrics using 1200 threads across 15 machines.
```

			Real
Algorithm	Nodes	Complete	Time
centrality	18	100%	0.03

```
NOTE: Processing between/close centrality metrics used 0.03 seconds.
NOTE: Processing centrality metrics used 0.03 (cpu: 0.17) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.128162 seconds.
NOTE: The data set MYLIB.LINKSETOUT has 35 observations and 5 variables.
NOTE: The data set MYLIB.NODESETOUT has 18 observations and 5 variables.
STATUS=OK  PROBLEM_TYPE=CENTRALITY  SOLUTION_STATUS=OK  CPU_TIME=1.63  REAL_TIME=0.13
```

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

The links data table `mylib.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	0.00654	0.00654
Christoph	Gotti	1	0.02288	0.08551
Christoph	Nardo	1	0.04248	0.03922
Christoph	Yu	2	0.12092	0.14107
Christoph	Zhuo	1	0.02941	0.04575
Gotti	Chapman	3	0.17974	0.08682
Gotti	Oliver	1	0.00000	0.03050
Gotti	Patrick	1	0.05229	0.05392
Graffe	Hund	1	0.06209	0.04270
Graffe	Yu	2	0.16013	0.11024
Graffe	Zhuo	1	0.03268	0.07625
Gukrishnan	Leon	1	0.00654	0.03050
Kabutz	Gotti	1	0.06536	0.11187
Kabutz	Gukrishnan	1	0.00654	0.03050
Kabutz	Leon	1	0.00654	0.00654
Kabutz	Snopp	1	0.03268	0.03126
Leon	Gotti	1	0.06536	0.11187
Nardo	Gotti	1	0.04902	0.04575
Nardo	Zhuo	1	0.01961	0.02614
Oliver	Chapman	3	0.12745	0.06776
Oliver	Patrick	1	0.03268	0.01797
Polark	Angel	1	0.09804	0.09804
Polark	Chang	1	0.09804	0.09804
Polark	Yu	2	0.36601	0.36601
Snopp	Chapman	3	0.23529	0.14227
Snopp	Gukrishnan	1	0.09804	0.05011
Snopp	Leon	1	0.03268	0.03126
Weng	Angel	1	0.00654	0.00654
Weng	Chang	1	0.00654	0.00654
Weng	Polark	1	0.09804	0.09804
Yu	Chapman	3	0.35294	0.22734
Zhuo	Gotti	1	0.04902	0.09052
Zhuo	Hund	1	0.04902	0.06841
Zhuo	Oliver	1	0.02288	0.03453
Zhuo	Patrick	1	0.02614	0.03922

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.

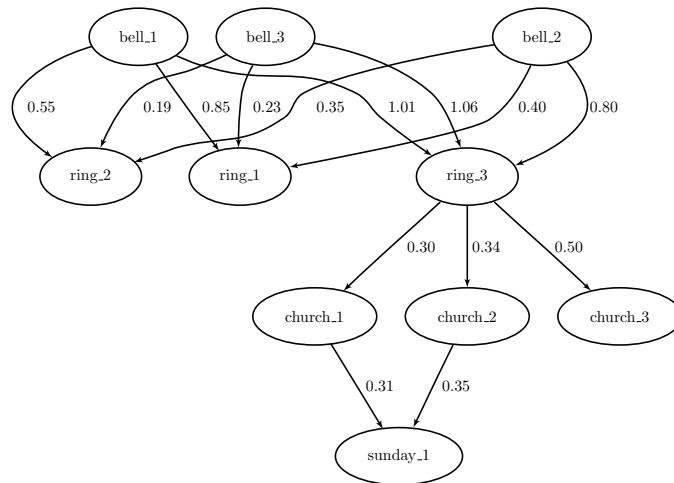
Example 2.5: Eigenvector Centrality for Word Sense Disambiguation

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 2.213](#).

Figure 2.213 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 mylib.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      = mylib.LinkSetIn
    outNodes    = mylib.NodeSetOut;
    centrality
        eigen   = weight;
run;

data mylib.NodeSetOut;
    length word $8 sense $1;
    set mylib.NodeSetOut;
    word = scan(node,1,'_');
    sense = scan(node,2,'_');
run;

```

```

proc sort
  data = mylib.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](#).

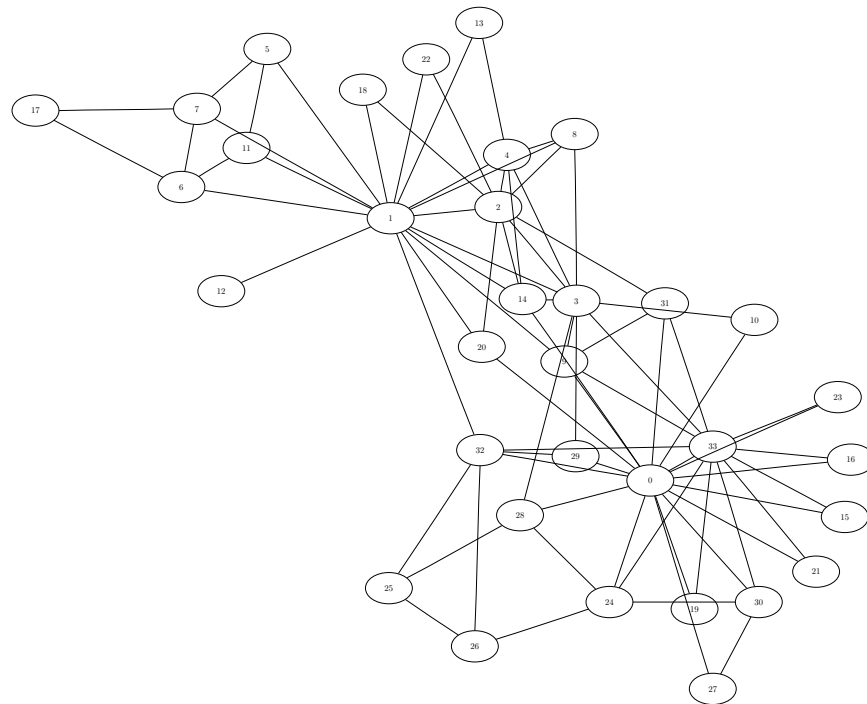
Output 2.5.1 Eigenvector Centrality for Word Sense Disambiguation

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 2.214](#).

Figure 2.214 Zachary's Karate Club Graph

You can represent the graph by using the following links data table, mylib.LinkSetIn:

```
data mylib.LinkSetIn;
  input from to @@;
  datalines;
    0 9 0 10 0 14 0 15 0 16 0 19 0 20 0 21
    0 23 0 24 0 27 0 28 0 29 0 30 0 31 0 32
    0 33 2 1 3 1 3 2 4 1 4 2 4 3 5 1
    6 1 7 1 7 5 7 6 8 1 8 2 8 3 8 4
    9 1 9 3 10 3 11 1 11 5 11 6 12 1 13 1
    13 4 14 1 14 2 14 3 14 4 17 6 17 7 18 1
    18 2 20 1 20 2 22 1 22 2 26 24 26 25 28 3
    28 24 28 25 29 3 30 24 30 27 31 2 31 9 32 1
    32 25 32 26 32 29 33 3 33 9 33 15 33 16 33 19
    33 21 33 23 33 24 33 30 33 31 33 32
  ;
```

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 130.

```
proc network
  links          = mylib.LinkSetIn
  outNodes       = mylib.NodeSetOut;
  community
    resolutionList = 1.0 0.5
    outLevel       = mylib.CommLevelOut
    outCommunity   = mylib.CommOut
    outOverlap     = mylib.CommOverlapOut
```

```

outCommLinks = mylib.CommLinksOut;
run;

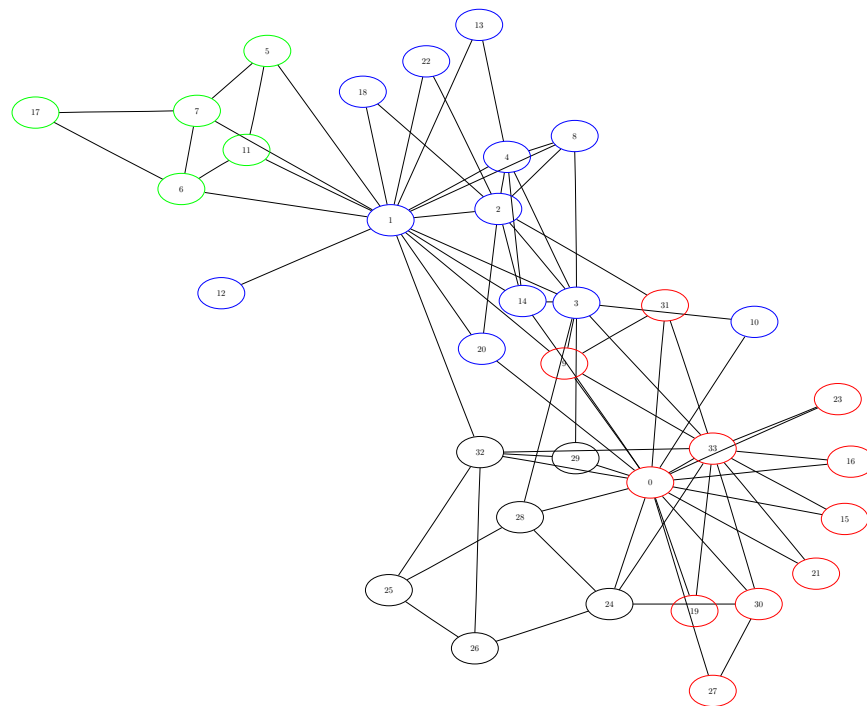
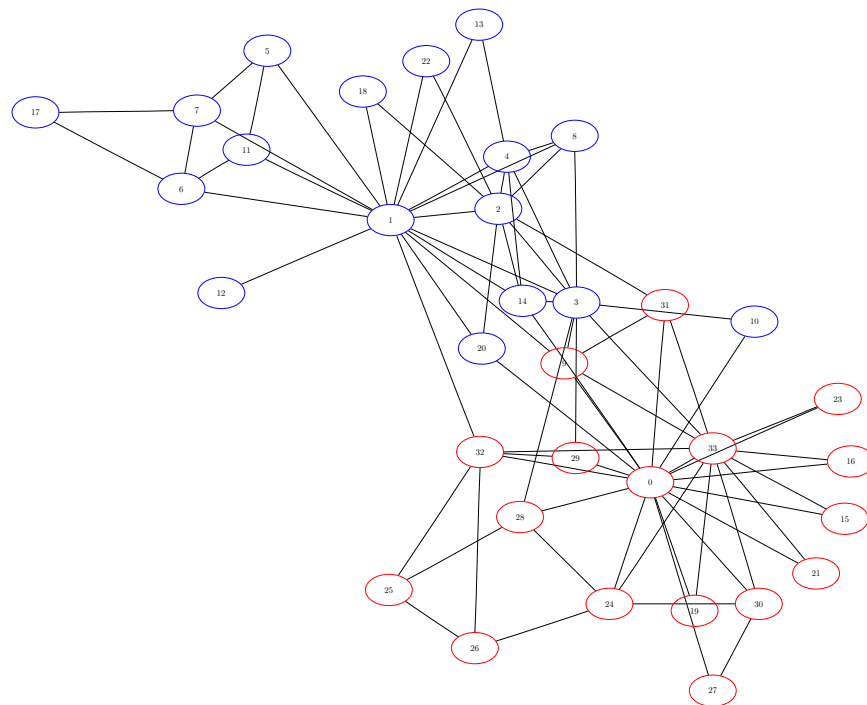
```

The output data table `mylib.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	node	community_1	community_2
0	1	1	17	3	2
1	2	2	18	2	2
2	2	2	19	1	1
3	2	2	20	2	2
4	2	2	21	1	1
5	3	2	22	2	2
6	3	2	23	1	1
7	3	2	24	4	1
8	2	2	25	4	1
9	1	1	26	4	1
10	2	2	27	1	1
11	3	2	28	4	1
12	2	2	29	4	1
13	2	2	30	1	1
14	2	2	31	1	1
15	1	1	32	4	1
16	1	1	33	1	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 2.215](#) and [Figure 2.216](#). As you can see from the figures, four communities at resolution 1.0 are merged into two communities at resolution 0.5.

Figure 2.215 Karate Club Communities (Resolution = 1.0)**Figure 2.216** Karate Club Communities (Resolution = 0.5)

The output data table `mylib.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 `mylib.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	intra_links	inter_links	density	cut_ratio	conductance
1	1.0	1	11	20	14	0.36364	0.055336	0.25926
1	1.0	2	12	24	14	0.36364	0.053030	0.22581
1	1.0	3	5	6	4	0.60000	0.027586	0.25000
1	1.0	4	6	7	10	0.46667	0.059524	0.41667
2	0.5	1	17	34	10	0.25000	0.034602	0.12821
2	0.5	2	17	34	10	0.25000	0.034602	0.12821

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

The output data table `mylib.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 4.

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	4	7
1	1.0	2	3	4
1	1.0	2	4	3
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 283. 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          = mylib.LinkSetIn
  outNodes       = mylib.NodeSetOut;
  community
    resolutionList = 1.0
    recursive (maxCommSize = 5 maxDiameter = 2 relation = AND)
    outCommunity  = mylib.CommOut;
run;
```

The output data table mylib.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	node	community_1
0	4	17	1
1	7	18	8
2	8	19	2
3	6	20	8
4	5	21	4
5	1	22	7
6	1	23	2
7	1	24	9
8	6	25	9
9	3	26	9
10	6	27	4
11	1	28	9
12	7	29	10
13	5	30	4
14	5	31	3
15	2	32	10
16	2	33	2

The output data table mylib.CommOut contains the number of nodes in each community, as shown in [Output 2.7.2](#).

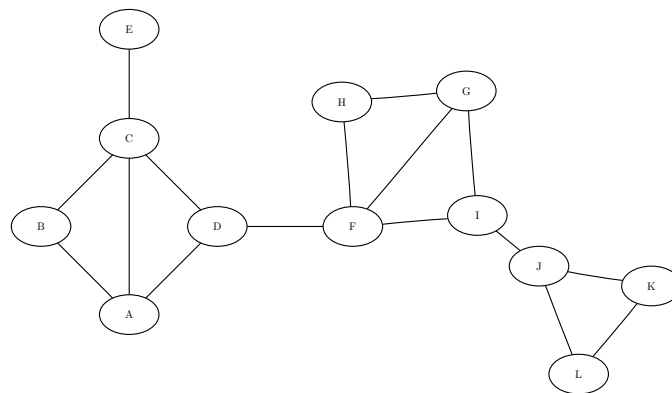
Output 2.7.2 Community Number of Nodes Output

level	resolution	community	nodes	intra_links	inter_links	density	cut_ratio	conductance
1	1	1	5	6	4	0.60000	0.02759	0.25000
1	1	2	5	4	12	0.40000	0.08276	0.60000
1	1	3	2	1	7	1.00000	0.10938	0.77778
1	1	4	4	4	17	0.66667	0.14167	0.68000
1	1	5	3	2	9	0.66667	0.09677	0.69231
1	1	6	3	2	12	0.66667	0.12903	0.75000
1	1	7	3	2	15	0.66667	0.16129	0.78947
1	1	8	3	2	10	0.66667	0.10753	0.71429
1	1	9	4	4	7	0.66667	0.05833	0.46667
1	1	10	2	1	7	1.00000	0.10938	0.77778

Example 2.8: Centrality Metrics for an 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 2.217.

Figure 2.217 Undirected Graph

The following statements create the data table mylib.LinkSetIn:

```

data mylib.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      = mylib.LinkSetIn
  outNodes   = mylib.OutNodesComms
  outLinks   = mylib.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, mylib.OutNodesComms, is shown in [Output 2.8.1](#).

Output 2.8.1 Nodes for the Communities of an 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, mylib.OutLinksComms, is shown in [Output 2.8.2](#).

Output 2.8.2 Links for the Communities of an 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 2.218 through Figure 2.220.

Figure 2.218 Subgraph $C^1 = \{A, B, C, D, E\}$

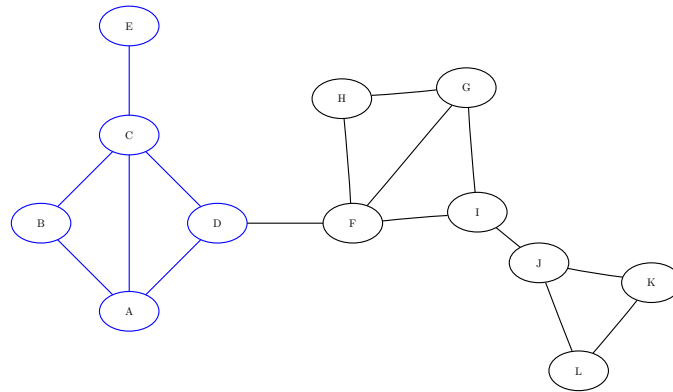


Figure 2.219 Subgraph $C^2 = \{F, G, H, I\}$

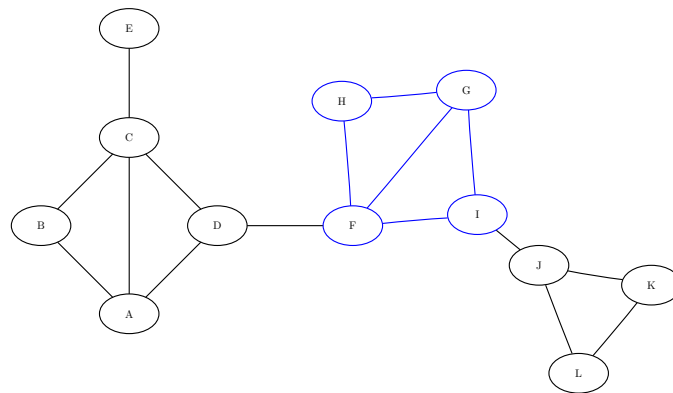
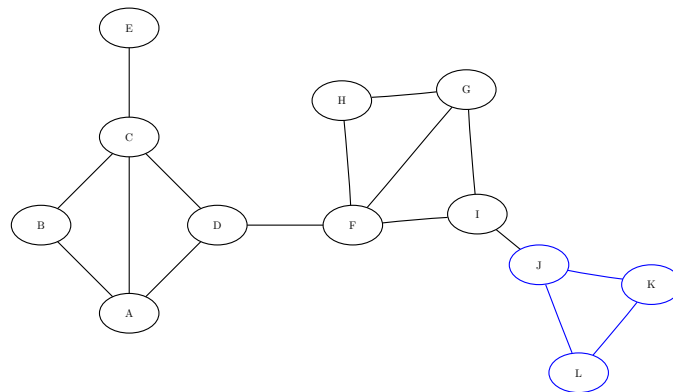


Figure 2.220 Subgraph $C^3 = \{J, K, L\}$



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 is done in parallel across

machines and threads (depending on your server configuration).

```
proc network
  links          = mylib.OutLinksComms(where=(community_1 ne .))
  outNodes       = mylib.NodeSetOut;
  centrality
    degree       = unweight
    influence     = unweight
    close        = unweight
    between      = unweight
    eigen        = unweight;
  display
    ProblemSummary = ProblemSummary
    SolutionSummary = SolutionSummary;
  by community_1;
run;
%put &_NETWORK_;
```

Assuming that your grid has a total of at least three cores, all three subgraphs are processed simultaneously with one call to PROC NETWORK. The progress of the procedure is shown in [Output 2.8.3](#).

Output 2.8.3 PROC NETWORK Log: Centrality by Cluster for an 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 Cloud Analytic Services server processed the request in 0.157482 seconds.
NOTE: The data set MYLIB.NODESETOUT has 12 observations and 8 variables.
STATUS=OK  PROBLEM_TYPE=CENTRALITY  CPU_TIME=2.26  REAL_TIME=0.16
```

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 mylib.ProblemSummary contains a summary of each induced subgraph that is processed by

PROC NETWORK.

Output 2.8.4 Problem Summary by Community

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

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

Output 2.8.5 Solution Summary by Community

community_1	problemType	status	cpuTime	realTime
3	Centrality	OK	0.00	0.00
1	Centrality	OK	0.00	0.00
2	Centrality	OK	0.00	0.00

The centrality results (by community) are shown in [Output 2.8.6](#).

Output 2.8.6 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

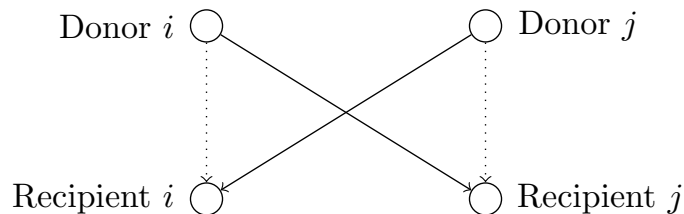
community_1=3

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: Cycle Enumeration for Kidney Donor Exchange

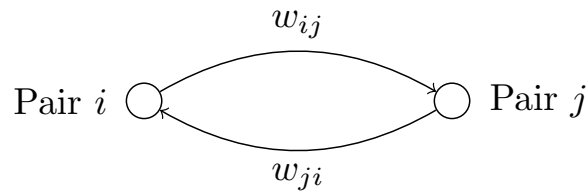
This example looks at an application of cycle enumeration to help create a kidney donor exchange. Suppose someone needs a kidney transplant and a family member is willing to be a donor. If the donor and recipient are incompatible (because of blood type, tissue mismatch, and so on), the transplant cannot happen. Now suppose two donor-recipient pairs, i and j , are in this situation, but donor i is compatible with recipient j and donor j is compatible with recipient i . Then two transplants can take place in a two-way swap, shown in Figure 2.221. More generally, an n -way swap can be performed involving n donors and n recipients (CNN 2012).

Figure 2.221 Kidney Donor Exchange Two-Way Swap



To model this problem, define a directed graph as follows: Each node is an incompatible donor-recipient pair. Link (i, j) exists if the donor from node i is compatible with the recipient from node j , as shown in Figure 2.222.

Figure 2.222 Kidney Donor Exchange Network



The link weight is a measure of the quality of the match. By introducing dummy links whose weight is 0, you can also include recipients who have no donors and altruistic donors who have no recipients. The idea is to find a maximum-weight node-disjoint union of directed cycles. You want the union to be node-disjoint so that no kidney is donated more than once, and you want cycles so that the donor from node i donates a kidney if and only if the recipient from node i receives a kidney.

Without any other constraints, the problem could be solved as a linear assignment problem, as described in *SAS Optimization: The OPTNETWORK Procedure*. But doing so would allow arbitrarily long cycles in the solution. For practical considerations (such as travel) and to mitigate risk, each cycle must have no more than L links. The kidney exchange problem is to find a maximum-weight node-disjoint union of short directed cycles.

One way to solve this problem is to explicitly generate all cycles whose length is at most L and then solve a set-packing problem. You can use PROC NETWORK to generate the cycles and then use PROC OPTMODEL to read the PROC NETWORK output, formulate the set-packing problem, call the mixed integer linear programming solver, and output the optimal solution. See Chapter 9, “The OPTMODEL Procedure” (*SAS Optimization: Mathematical Optimization Procedures*).

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

```
/* create random graph on n nodes with link probability p
   and uniform(0,1) weight */
%let n = 100;
%let p = 0.02;
data mylib.LinkSetIn;
  call streaminit(1);
  do from = 0 to &n - 1;
    do to = 0 to &n - 1;
      if from eq to then continue;
      else if rand('UNIFORM') < &p then do;
        weight = rand('UNIFORM');
        output;
      end;
    end;
  end;
run;
```

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

```
/* generate all cycles with 2 <= length <= max_length */
%let max_length = 10;
proc network
  logLevel      = moderate
  direction     = directed
  links         = mylib.LinkSetIn;
  cycle
    minLength   = 2
    maxLength   = &max_length
    maxCycles    = all
    outCyclesLinks = mylib.CyclesLinks;
run;
%put &_NETWORK_;
```

PROC NETWORK finds 395 cycles of the appropriate length, as shown in [Output 2.9.1](#).

Output 2.9.1 PROC NETWORK Log: Cycles for Kidney Donor Exchange

```

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 graph storage used 0.00 (cpu: 0.00) seconds.
NOTE: The number of nodes in the input graph is 98.
NOTE: The number of links in the input graph is 208.
NOTE: Processing cycle enumeration using 80 threads across 1 machines.
NOTE: Processing cycle enumeration using the build algorithm.
NOTE: The algorithm found 395 cycles.
NOTE: Processing cycle enumeration used 0.01 (cpu: 0.02) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.071178 seconds.
NOTE: The data set MYLIB.CYCLESLINKS has 3431 observations and 5 variables.
STATUS=OK  PROBLEM_TYPE=CYCLE  SOLUTION_STATUS=OK  NUM_CYCLES=395  CPU_TIME=0.67  REAL_TIME=0.07

```

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

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

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

```

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

```

```

num cycle_weight {c in CYCLES} = sum {<i,j> in LINKS_c[c]} weight[i,j];

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

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

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

/* call solver */
solve;

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

```

PROC OPTMODEL solves the problem by using the mixed integer linear programming solver.

Output 2.9.2 PROC OPTMODEL Log: Cycles for Kidney Donor Exchange

```

NOTE: There were 208 observations read from the data set MYLIB.LINKSETIN.
NOTE: There were 3431 observations read from the data set MYLIB.CYCLESLINKS.
NOTE: Problem generation will use 4 threads.
NOTE: The problem has 395 variables (0 free, 0 fixed).
NOTE: The problem has 395 binary and 0 integer variables.
NOTE: The problem has 64 linear constraints (64 LE, 0 EQ, 0 GE, 0 range).
NOTE: The problem has 3431 linear constraint coefficients.
NOTE: The problem has 0 nonlinear constraints (0 LE, 0 EQ, 0 GE, 0 range).
NOTE: The OPTMODEL presolver is disabled for linear problems.
NOTE: The initial MILP heuristics are applied.
NOTE: The MILP presolver value AUTOMATIC is applied.
NOTE: The MILP presolver removed 110 variables and 31 constraints.
NOTE: The MILP presolver removed 1685 constraint coefficients.
NOTE: The MILP presolver added 1 constraint coefficients.
NOTE: The MILP presolver modified 11 constraint coefficients.
NOTE: The presolved problem has 285 variables, 33 constraints, and 1746 constraint coefficients.
NOTE: The MILP solver is called.
NOTE: The parallel Branch and Cut algorithm is used.
NOTE: The Branch and Cut algorithm is using up to 4 threads.

```

Node	Active	Sols	BestInteger	BestBound	Gap	Time
0	1	3	20.7728712	1199.7787581	98.27%	0
0	1	3	20.7728712	25.4194215	18.28%	0
0	1	3	20.7728712	24.9816665	16.85%	0
0	1	4	21.1953805	24.9816665	15.16%	0
0	1	5	24.8508554	24.8508554	0.00%	0
0	0	5	24.8508554	24.8508554	0.00%	0

```

NOTE: The MILP solver added 36 cuts with 4129 cut coefficients at the root.
NOTE: Optimal.
NOTE: Objective = 24.850855395.
NOTE: The data set WORK.SOLUTION has 7 observations and 2 variables.
STATUS=OK ALGORITHM=BAC SOLUTION_STATUS=OPTIMAL OBJECTIVE=24.850855395 RELATIVE_GAP=0
ABSOLUTE_GAP=0 PRIMAL_INFEASIBILITY=3.330669E-15 BOUND_INFEASIBILITY=1.404247E-13
INTEGER_INFEASIBILITY=6.76903E-13 BEST_BOUND=24.850855395 NODES=1 SOLUTIONS_FOUND=5
ITERATIONS=99 PRESOLVE_TIME=0.01 SOLUTION_TIME=0.07

```

The output data table mylib.Solution, shown in [Output 2.9.3](#), now contains the cycles that define the best exchange and their associated weight (quality).

Output 2.9.3 Maximum-Quality Solution for Kidney Donor Exchange

c cycle_weight	
26	4.3542
81	4.3403
134	4.9748
169	5.0843
362	1.9424
385	2.4295
391	1.7253
	24.8509

Example 2.10: Transitive Closure for Identification of Circular Dependencies in a Bug Tracking System

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

The bug tracking system needs to check for two situations when users declare a bug to be a DUP. The first situation is called a *circular dependency*. Consider bugs A, B, C, and D in the tracking system. The first user declares that A is a DUP of B and that C is a DUP of D. A second user declares that B is a DUP of C, and a third user declares that D is a DUP of A. You now have a circular dependency, and no primary bug is defined for the development team to focus on. You can easily see this circular dependency in the graph representation, because $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$. You can find such circular dependencies by using cycle enumeration, which is described in the section “[Cycle Enumeration](#)” on page 146. The second situation that needs to be checked is more general. If one user declares that A is a DUP of B and another user declares that B is a DUP of C, this chain of duplicates is already an issue. The bug tracking system needs to provide one primary bug to which the rest of the bugs are duplicated. You can identify the existence of these chains by calculating the transitive closure of the directed graph that is defined by the DUP links.

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

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

```
data mylib.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 `mylib.DefectLinks`. The output data table `mylib.Cycles` contains any circular dependencies, and the data table `mylib.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              = mylib.DefectLinks;
  linksVar
    from            = defectId
    to              = linkedDefect;
  cycle
    outCyclesNodes  = mylib.Cycles
    maxCycles       = all;
run;
%put &_NETWORK_;

proc network
  logLevel          = moderate
  direction          = directed
  links              = mylib.DefectLinks;
  linksVar
    from            = defectId
    to              = linkedDefect;
  transitiveClosure
    out             = mylib.TransClosure;
run;
%put &_NETWORK_;

proc sql;
  create table Chains as
  select defectId, linkedDefect
    from mylib.TransClosure (where=(defectId ne linkedDefect)) except
  select defectId, linkedDefect
    from mylib.DefectLinks;
quit;

```

The progress of the procedure is shown in [Output 2.10.1](#).

Output 2.10.1 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 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 enumeration using 1 threads across 1 machines.
NOTE: Processing cycle enumeration using the backtrack algorithm.
NOTE: The algorithm found 1 cycles.
NOTE: Processing cycle enumeration used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.079934 seconds.
NOTE: The data set MYLIB.CYCLES has 4 observations and 3 variables.
STATUS=OK  PROBLEM_TYPE=CYCLE  SOLUTION_STATUS=OK  NUM_CYCLES=1  CPU_TIME=2.16  REAL_TIME=0.08
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 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 using 80 threads across 1 machines.
NOTE: Processing the transitive closure used 0.01 (cpu: 0.01) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.099669 seconds.
NOTE: The data set MYLIB.TRANSCLASURE has 20 observations and 2 variables.
STATUS=OK  PROBLEM_TYPE=TRANSITIVECLOSURE  SOLUTION_STATUS=OK  CPU_TIME=2.15  REAL_TIME=0.10
NOTE: Table WORK.CHAINS created, with 5 rows and 2 columns.

```

Output 2.10.2 displays the output data table mylib.Cycles, which contains one case of a circular dependency in which the DUPs start and end at S0152674.

Output 2.10.2 Cycle in Bug Tracking System

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

Output 2.10.3 displays the local data set Chains, which contains the chains in the bug tracking system that come from the links in G^T that are not in G .

Output 2.10.3 Chains in Bug Tracking System

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

Example 2.11: Reach Networks for Computing the Market Coverage of a Terrorist Network

The problem of finding an efficient method for *covering* a market (a set of entities) is important in numerous industries. For example, consider that you are an advertising company with access to data that are collected from your customers' social networks. To keep costs at a minimum in a new promotion, you want to find a minimal set of customers to whom you need to advertise in order to reach the entire market. To do this, you could first generate all the reach networks for each customer by using PROC NETWORK. These networks could then be used in a *set-covering problem*, which you can solve as an integer linear program by using PROC OPTMODEL. Let N be the set of customers that you want to reach, and let the links E define the social network of those customers. If you use a one-hop reach network, you assume that if an advertisement is sent to customer i , then customer i will promote the advertisement to all his friends (those he is connected to in E). If you use two-hop reach networks, you assume that customer i 's friends will also promote the advertisement to their friends. So the question is, To which subset of customers should you advertise to reach all customers through the promotion mechanism?

This problem can be generalized as follows:

Given a graph $G = (N, E)$, choose a node set N^* of minimal size such that there is a path of length less than or equal to L to every node in N from a node in N^* .

To illustrate an application of this problem, consider again the terrorist communications network from “Example 2.1: Articulation Points in a Terrorist Network” on page 268. In this case, customers are alleged terrorists. Solving the covering problem here can give you a subset of people to focus on in an investigation in order to cover all members of the network.

The following macro, %GenerateReach, runs PROC NETWORK to generate the reach network for each person in the terrorist network for a variable hop limit:

```
%macro GenerateReach(limit=);
proc network
  outNodes      = mylib.NodeSetOut
  links         = mylib.LinkSetInTerror911;
  reach
    eachSource
      outReachNodes = mylib.ReachNode
      maxReach      = &limit;
run;
%mend GenerateReach;
```

The following macro, %SolverCover, runs PROC OPTMODEL to solve the set-covering problem:

```

%macro SolverCover();
proc optmodel;
    string      tmpLabel;
    set<num>     NODE_ID;
    set<string>  NODE_LABEL init {};
    string      nodeIdToLabel{NODE_ID};
    num         nodeLabelToId{NODE_LABEL};

    set<num> REACH_SET{NODE_ID} init {};
    set<string,num> PAIRS;

    /* read data */
    read data mylib.NodeSetOut into NODE_ID=[_n_] nodeIdToLabel=node;
    read data mylib.ReachNode into PAIRS=[node reach];
    for{i in NODE_ID} do;
        tmpLabel = nodeIdToLabel[i];
        NODE_LABEL = NODE_LABEL union {tmpLabel};
        nodeLabelToId[tmpLabel] = i;
    end;
    for{<label,i> in PAIRS} do;
        REACH_SET[i] = REACH_SET[i] union {nodeLabelToId[label]};
    end;

    /* declare decision variables */
    var x {NODE_ID} binary;

    /* declare objective */
    minimize numNodes = sum{j in NODE_ID} x[j];

    /* cover constraint */
    con cover {i in NODE_ID}:
        sum{j in REACH_SET[i]} x[j] >= 1;

    /* solve */
    solve;

    create data Solution from [label]=
        (setof{j in NODE_ID : round(x[j].sol)=1}nodeIdToLabel[j]);
quit;
%mend SolverCover;

```

The following statements calculate the minimal cover for the one-hop limit:

```

%GenerateReach(limit=1);
%SolverCover();

```

To cover the network, assuming a one-hop limit, the investigators would need to investigate the people listed in the data table `mylib.Solution`, shown in [Output 2.11.1](#).

Output 2.11.1 Minimal One-Hop Cover for Terrorist Communications Network

label
Djamal Beghal
Essid Sami Ben Khemais
Fayez Ahmed
Hani Hanjour
Mamoun Darkazanli
Mohamed Atta
Nabil al-Marabh
Nawaf Alhazmi
Ramzi Bin al-Shibh
Zacarias Moussaoui

The following statements calculate the minimal cover for the two-hop limit:

```
%GenerateReach(limit=2);
%SolverCover();
```

If investigators assume a two-hop limit, they could focus their attention on the two people shown in [Output 2.11.2](#). Then, by following their links (and their links' links), they could cover the entire network.

Output 2.11.2 Minimal Two-Hop Cover for Terrorist Communications Network

label
Mohamed Atta
Zacarias Moussaoui

Example 2.12: Connected Components for US Patent Citations

This example looks at the structural relationship of US patent citations by using a large data set that is maintained by the Stanford Network Analysis Project (SNAP) (Leskovec 2014). The citation graph includes over 16 million citations made to patents between 1975 and 1999.

The following statements construct the links data table `mylib.Patents` from a local copy of the raw patent citation data:

```
filename in 'cit-Patents.txt';
data mylib.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          = mylib.Patents
  outNodes       = mylib.NodeSetOut
  distributed    = true;
```

```

connectedComponents
    out      = mylib.ConCompOut;
run;
%put &_NETWORK_;

```

The progress of the procedure is shown in [Output 2.12.1](#).

Output 2.12.1 PROC NETWORK Log: Connected Components for US Patent Citations

```

NOTE: -----
NOTE: Running NETWORK.
NOTE: -----
NOTE: The number of nodes in the input graph is 3774768.
NOTE: The number of links in the input graph is 16518948.
NOTE: Processing connected components using 1200 threads across 15 machines.
NOTE: The graph has 3627 connected components.
NOTE: Processing connected components used 1.09 (cpu: 9.21) seconds.
NOTE: The Cloud Analytic Services server processed the request in 1.986867 seconds.
NOTE: The data set MYLIB.NODESETOUT has 3774768 observations and 2 variables.
NOTE: The data set MYLIB.CONCOMPOUT has 3627 observations and 2 variables.
STATUS=OK  PROBLEM_TYPE=CONNECTEDCOMPONENTS  SOLUTION_STATUS=OK  NUM_COMPONENTS=3627
CPU_TIME=57.11  REAL_TIME=1.99

```

The 10 biggest components are shown in [Output 2.12.2](#). It is interesting to note that the vast majority of patents (over 99%) are all contained in the same component. This is not too surprising, because many of the seminal patent claims are required in order to understand subsequent inventions.

Output 2.12.2 Ten Largest Components for US Patent Citations

Obs	concomp	nodes
1	1	3764117
2	69	19
3	295	16
4	301	15
5	43	14
6	2680	14
7	3222	14
8	163	14
9	1283	13
10	194	13

Example 2.13: Shortest Paths of the New York Road Network

This example looks at the road networks in the state of New York (NY). The distance graph raw data are maintained at the DIMACS challenge website (Demetrescu 2010). The NY road network includes 264,346 intersections (nodes) and 733,846 roads (links). Although the input data table is not large, the computing power that you need to find all-pairs shortest paths is enormous. In addition, the storage space that you need to handle the results data can easily overwhelm the capacity of a single machine. In this example, a session of 16 machines (each with 80 cores) was configured to process this graph.

The following statements construct the links data table `mylib.RoadNY` from a local copy of the raw distance graph data:

```
filename in 'USA-road-d.NY.gr';
data mylib.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
  direction        = directed
  links            = mylib.RoadNY;
  shortestPath
    maxLinkWeight = 20000
    outWeights    = mylib.shortPathSummary
    outPathsLinks = mylib.shortPathPaths;
run;
%put &_NETWORK_;
```

The progress of the procedure is shown in [Output 2.13.1](#).

Output 2.13.1 PROC NETWORK Log: Shortest Paths of the NY Road Network

```

NOTE: -----
NOTE: -----
NOTE: Running NETWORK.
NOTE: -----
NOTE: -----
NOTE: Reading the links data.
NOTE: Data input used 0.06 (cpu: 4.78) seconds.
NOTE: Building the input graph storage used 0.14 (cpu: 10.29) seconds.
NOTE: The number of nodes in the input graph is 264346.
NOTE: The number of links in the input graph is 733846.
NOTE: Processing shortest paths problem using 1200 threads across 15 machines.
NOTE: Processing the shortest paths problem between 264346 source nodes and 264346 sink nodes.

      Algorithm      Sources      Complete      Real
      Time
shortestPath      18000           6%          9.27
shortestPath      43600          16%         19.15
shortestPath      64400          24%         29.06
shortestPath      84800          32%         39.62
shortestPath     107200          40%         49.37
shortestPath     131200          49%         59.10
shortestPath     154000          58%         69.09
shortestPath     177600          67%         79.11
shortestPath     200000          75%         89.09
shortestPath     222800          84%         99.42
shortestPath     243600          92%        109.30
shortestPath     262400          99%        119.35
shortestPath     264346         100%       120.38

NOTE: Processing the shortest paths problem used 120.38 (cpu: 120578.53) seconds.
NOTE: The Cloud Analytic Services server processed the request in 121.312426 seconds.
NOTE: The data set MYLIB.SHORTPATHSUMMARY has 104263396 observations and 3 variables.
NOTE: The data set MYLIB.SHORTPATHPATHS has 1419295876 observations and 6 variables.
STATUS=OK  PROBLEM_TYPE=SHORTESTPATH  SOLUTION_STATUS=OK  NUM_PATHS=104263396
CPU_TIME=121735.16  REAL_TIME=121.31

```

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

Example 2.14: Shortest Path in a Road Network by Date and Time

This example reconsiders the road network between a SAS employee's home in Raleigh, North Carolina, and SAS headquarters nearby in Cary introduced in the section “[Road Network Shortest Path](#)” on page 11. The following data provide a snapshot of the road network and travel times observed at three different times:

```
data mylib.LinkSetInRoadNC;
  input start_inter $1-20 end_inter $21-40 miles miles_per_hour
        date date11. time time10.;
  format date date11. time time10.;
  time_to_travel = miles * 1/miles_per_hour * 60;
  datalines;
614CapitalBlvd      Capital/WadeAve      0.6  25  15-APR-2013  10:30 am
614CapitalBlvd      Capital/US70W      0.6  25  15-APR-2013  10:30 am
614CapitalBlvd      Capital/US440W     3.0  45  15-APR-2013  10:30 am
Capital/WadeAve      WadeAve/RaleighExpy  3.0  40  15-APR-2013  10:30 am
Capital/US70W        US70W/US440W      3.2  60  15-APR-2013  10:30 am
US70W/US440W        US440W/RaleighExpy  2.7  60  15-APR-2013  10:30 am
Capital/US440W       US440W/RaleighExpy  6.7  60  15-APR-2013  10:30 am
US440W/RaleighExpy  RaleighExpy/US40W   3.0  60  15-APR-2013  10:30 am
WadeAve/RaleighExpy RaleighExpy/US40W   3.0  60  15-APR-2013  10:30 am
RaleighExpy/US40W   US40W/HarrisonAve   1.3  55  15-APR-2013  10:30 am
US40W/HarrisonAve   SASCampusDrive      0.5  25  15-APR-2013  10:30 am
614CapitalBlvd      Capital/WadeAve      0.6  25  16-APR-2013   9:30 am
614CapitalBlvd      Capital/US70W      0.6  25  16-APR-2013   9:30 am
614CapitalBlvd      Capital/US440W     3.0  45  16-APR-2013   9:30 am
Capital/WadeAve      WadeAve/RaleighExpy  3.0  25  16-APR-2013   9:30 am
Capital/US70W        US70W/US440W      3.2  60  16-APR-2013   9:30 am
US70W/US440W        US440W/RaleighExpy  2.7  60  16-APR-2013   9:30 am
Capital/US440W       US440W/RaleighExpy  6.7  60  16-APR-2013   9:30 am
US440W/RaleighExpy  RaleighExpy/US40W   3.0  60  16-APR-2013   9:30 am
WadeAve/RaleighExpy RaleighExpy/US40W   3.0  60  16-APR-2013   9:30 am
RaleighExpy/US40W   US40W/HarrisonAve   1.3  55  16-APR-2013   9:30 am
US40W/HarrisonAve   SASCampusDrive      0.5  25  16-APR-2013   9:30 am
614CapitalBlvd      Capital/WadeAve      0.6  25  18-APR-2013   8:30 am
614CapitalBlvd      Capital/US440W     3.0  45  18-APR-2013   8:30 am
Capital/WadeAve      WadeAve/RaleighExpy  3.0  25  18-APR-2013   8:30 am
Capital/US440W       US440W/RaleighExpy  6.7  60  18-APR-2013   8:30 am
US440W/RaleighExpy  RaleighExpy/US40W   3.0  60  18-APR-2013   8:30 am
WadeAve/RaleighExpy RaleighExpy/US40W   3.0  60  18-APR-2013   8:30 am
RaleighExpy/US40W   US40W/HarrisonAve   1.3  55  18-APR-2013   8:30 am
US40W/HarrisonAve   SASCampusDrive      0.5  25  18-APR-2013   8:30 am
;
```

The first snapshot (15-APR-2013 10:30 am) is a typical traffic pattern on a workday. The second snapshot (16-APR-2013 9:30 am) represents morning rush-hour traffic, and the third (18-APR-2013 8:30 am) represents rush-hour traffic where a major highway (US70W) has been closed for repairs.

The following statements find the route that yields the shortest path between home (614 Capital Boulevard) and SAS headquarters (SAS Campus Drive) for all three scenarios simultaneously by using the **BY** statement:

```

proc network
  links          = mylib.LinkSetInRoadNC;
  linksVar
    from         = start_inter
    to           = end_inter
    weight       = time_to_travel;
  shortestPath
    outPathsLinks = mylib.ShortPathP
    outWeights    = mylib.ShortPathW
    source        = "614CapitalBlvd"
    sink          = "SASCampusDrive";
  display
    ProblemSummary = ProblemSummary
    SolutionSummary = SolutionSummary;
  by date time;
run;
%put &_NETWORK_;

```

Assuming that your grid has a total of at least three cores, all three graphs are processed simultaneously through one call to PROC NETWORK. The progress of the procedure is shown in [Output 2.14.1](#).

Output 2.14.1 PROC NETWORK Log: Shortest Path in a Road Network by Date and Time

```

NOTE: -----
NOTE: Running NETWORK.
NOTE: -----
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 11.
NOTE: Processing the shortest paths problem using 1 threads across 1 machines.
NOTE: Processing the shortest paths problem between 1 source nodes and 1 sink nodes.
NOTE: Processing the shortest paths problem used 0.00 (cpu: 0.00) seconds.
NOTE: The above message was for the following BY group:
      date=15-APR-2013 time=10:30:00
NOTE: The number of nodes in the input graph is 10.
NOTE: The number of links in the input graph is 11.
NOTE: Processing the shortest paths problem using 5 threads across 1 machines.
NOTE: Processing the shortest paths problem between 1 source nodes and 1 sink nodes.
NOTE: Processing the shortest paths problem used 0.00 (cpu: 0.00) seconds.
NOTE: The above message was for the following BY group:
      date=16-APR-2013 time=9:30:00
NOTE: The number of nodes in the input graph is 8.
NOTE: The number of links in the input graph is 8.
NOTE: Processing the shortest paths problem using 2 threads across 1 machines.
NOTE: Processing the shortest paths problem between 1 source nodes and 1 sink nodes.
NOTE: Processing the shortest paths problem used 0.00 (cpu: 0.00) seconds.
NOTE: The above message was for the following BY group:
      date=18-APR-2013 time=8:30:00
NOTE: The Cloud Analytic Services server processed the request in 0.24477 seconds.
NOTE: The data set MYLIB.SHORTPATHP has 16 observations and 8 variables.
NOTE: The data set MYLIB.SHORTPATHW has 3 observations and 5 variables.
STATUS=OK  PROBLEM_TYPE=SHORTESTPATH  CPU_TIME=2.66  REAL_TIME=0.24

```

Output 2.14.2 displays the output table mylib.ProblemSummary, which contains a summary of each graph that is processed by PROC NETWORK.

Output 2.14.2 Problem Summary by Date and Time

date	time	numNodes	numLinks	graphDirection
15-APR-2013	10:30:00	10	11	Undirected
16-APR-2013	9:30:00	10	11	Undirected
18-APR-2013	8:30:00	8	8	Undirected

Output 2.14.3 displays the output table mylib.SolutionSummary, which contains a solution summary for the processing on each graph.

Output 2.14.3 Solution Summary by Date and Time

date	time	problemType	status	numPaths	cpuTime	realTime
15-APR-2013	10:30:00	Shortest Path	OK	1	0.00	0.00
16-APR-2013	9:30:00	Shortest Path	OK	1	0.00	0.00
18-APR-2013	8:30:00	Shortest Path	OK	1	0.00	0.00

Output 2.14.4 displays the output data table mylib.ShortPathW, which shows the total time to travel on the best route for each time snapshot.

Output 2.14.4 Shortest Path Summary for Road Network at Each Date and Time

date	time	source	sink	path_weight
15-APR-2013	10:30:00	614CapitalBlvd	SASCampusDrive	11.5582
16-APR-2013	9:30:00	614CapitalBlvd	SASCampusDrive	12.9582
18-APR-2013	8:30:00	614CapitalBlvd	SASCampusDrive	14.2582

Output 2.14.5 displays the output data table mylib.ShortPathP, which shows (by date and time) the best route for each time snapshot.

Output 2.14.5 Shortest Path for Road Network by Date and Time

date=15-APR-2013 time=10:30:00			
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
time			11.5582
date			11.5582

Output 2.14.5 *continued*

date=16-APR-2013 time=9:30:00

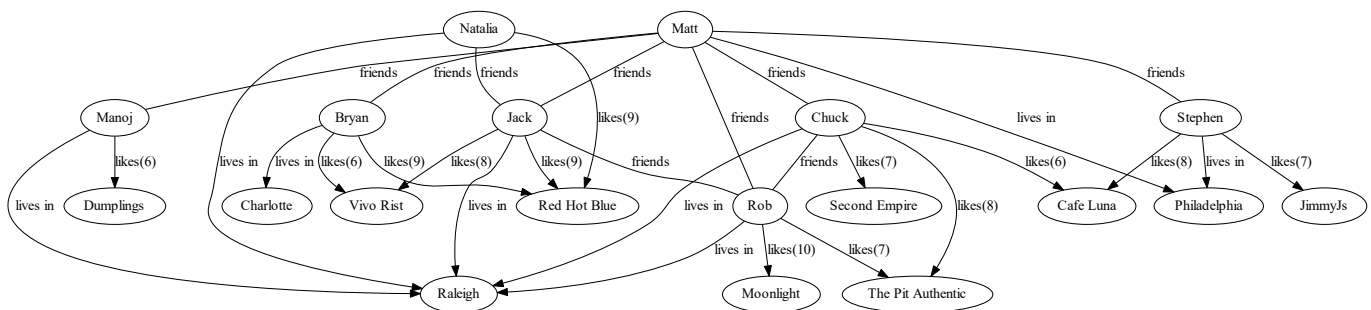
order	start_inter	end_inter	time_to_travel
1	614CapitalBlvd	Capital/US70W	1.4400
2	Capital/US70W	US70W/US440W	3.2000
3	US70W/US440W	US440W/RaleighExpy	2.7000
4	US440W/RaleighExpy	RaleighExpy/US40W	3.0000
5	RaleighExpy/US40W	US40W/HarrisonAve	1.4182
6	US40W/HarrisonAve	SASCampusDrive	1.2000
time			12.9582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

date=18-APR-2013 time=8:30:00

order	start_inter	end_inter	time_to_travel
1	614CapitalBlvd	Capital/WadeAve	1.4400
2	Capital/WadeAve	WadeAve/RaleighExpy	7.2000
3	WadeAve/RaleighExpy	RaleighExpy/US40W	3.0000
4	RaleighExpy/US40W	US40W/HarrisonAve	1.4182
5	US40W/HarrisonAve	SASCampusDrive	1.2000
time			14.2582
date			14.2582
			38.7745

Example 2.15: Pattern Matching in a Social Network

This example considers a portion of a social network that conveys relationships between people (*friends*), residences (*lives in*), and preferences for particular restaurants (*likes*). The network, directed graph G , is shown in Figure 2.223.

Figure 2.223 Social Network G 

The following data provide a snapshot of the social connections between Matt and a few of his friends:


```

data mylib.NodesSocial;
  infile datalines dsd;
  length node $40. type $40. subtype $20.;
  input node $ type $ subtype $;
  label=node;
  datalines;
Matt,          Person,
Rob,           Person,
Chuck,         Person,
Stephen,       Person,
Manoj,         Person,
Bryan,         Person,
Jack,          Person,
Natalia,       Person,
Raleigh,       City,
Philadelphia,  City,
Charlotte,     City,
The Pit Authentic, Restaurant, BBQ
Red Hot Blue,  Restaurant, BBQ
JimmyJs,       Restaurant, BBQ
Second Empire, Restaurant, American
Cafe Luna,     Restaurant, Italian
Vivo Rist,     Restaurant, Italian
Moonlight,     Restaurant, Italian
Dumplings,     Restaurant, Chinese
;
data mylib.LinksSocial;
  infile datalines dsd;
  length from $40. to $40. connection $20.;
  input from $ to $ connection $ rating;
  datalines;
Matt,  Rob,          friends,  .
Rob,   Matt,         friends,  .
Matt,  Chuck,        friends,  .
Chuck, Matt,         friends,  .
Chuck, Rob,          friends,  .
Rob,   Chuck,        friends,  .
Jack,  Rob,          friends,  .
Rob,   Jack,         friends,  .
Matt,  Stephen,      friends,  .
Stephen, Matt,       friends,  .
Matt,  Manoj,        friends,  .
Manoj, Matt,         friends,  .
Matt,  Bryan,        friends,  .
Bryan, Matt,         friends,  .
Matt,  Jack,         friends,  .
Jack,  Matt,         friends,  .
Natalia, Jack,       friends,  .
Jack,  Natalia,      friends,  .
Matt,  Philadelphia, lives in,  .
Stephen, Philadelphia, lives in,  .
Stephen, JimmyJs,    likes,    7
Stephen, Cafe Luna,  likes,    8

```

```

Rob,      Raleigh,      lives in, .
Chuck,    Raleigh,      lives in, .
Manoj,    Raleigh,      lives in, .
Jack,     Raleigh,      lives in, .
Natalia,  Raleigh,      lives in, .
Bryan,    Charlotte,     lives in, .
Rob,      The Pit Authentic, likes, 7
Jack,     Red Hot Blue,  likes, 9
Chuck,    The Pit Authentic, likes, 8
Chuck,    Cafe Luna,     likes, 6
Chuck,    Second Empire, likes, 7
Jack,     Vivo Rist,     likes, 8
Manoj,    Dumplings,     likes, 6
Natalia,  Red Hot Blue,  likes, 9
Bryan,    Red Hot Blue,  likes, 9
Bryan,    Vivo Rist,     likes, 6
Rob,      Moonlight,     likes, 10
;

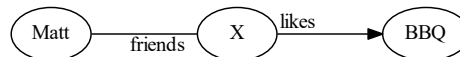
```

The nodes in the nodes data table `mylib.NodesSocial` represent people, cities, and restaurants. The node attribute `type` defines the node type. In the case of a restaurant, the node attribute `subtype` defines the type of restaurant.

The links in the links data table `mylib.LinksSocial` represent connections between the nodes. The type of connection is defined by the link attribute `connection`, and in the case of people connected to restaurants, the link attribute `rating` specifies a rating on a scale of 1 to 10.

For these data, a typical social network pattern search might be to find “friends of Matt who like barbecue restaurants.” This pattern is shown in [Figure 2.224](#).

Figure 2.224 Query Graph Q



In order to construct this pattern, the query graph can be represented using the data that are created by the following DATA steps:

```

data mylib.NodesSocialQuery;
  infile datalines dsd;
  length node $40. label $40. type $40. subtype $20.;
  input node $ label $ type $ subtype $;
  datalines;
Matt, Matt, Person,
X,, Person,
BBQ,, Restaurant, BBQ
;
data mylib.LinksSocialQuery;
  infile datalines dsd;
  length from $40. to $40. connection $20.;
  input from $ to $ connection $;

```

```

    datalines;
Matt, X,      friends
X,      Matt, friends
X,      BBQ,  likes
;

```

The query graph nodes data table implies that:

- The query node Matt must be a person with the node attribute label=Matt.
- The query node X can be any person.
- The query node BBQ must be a barbecue restaurant (that is, type=Restaurant and subtype=BBQ).

The query graph links data table implies that:

- Person Matt and person X must be friends.
- Person X must like the restaurant that is assigned to node BBQ.

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

```

proc network
    direction      = directed
    nodes          = mylib.NodesSocial
    links          = mylib.LinksSocial
    nodesQuery     = mylib.NodesSocialQuery
    linksQuery     = mylib.LinksSocialQuery;
    nodesVar
        vars      = (label type subtype);
    linksVar
        vars      = (connection);
    nodesQueryVar
        vars      = (label type subtype);
    linksQueryVar
        vars      = (connection);
    patternMatch
        outMatchNodes = mylib.OutMatchNodes
        outMatchLinks = mylib.OutMatchLinks;
run;
%put &_NETWORK_;

```

The progress of the procedure is shown in [Output 2.15.1](#).

Output 2.15.1 PROC NETWORK Log: Pattern Matching in a Social Network

```

NOTE: -----
NOTE: Running NETWORK.
NOTE: -----
NOTE: The number of nodes in the input graph is 19.
NOTE: The number of links in the input graph is 39.
NOTE: The number of nodes in the query graph is 3.
NOTE: The number of links in the query graph is 3.
NOTE: Processing the pattern matching query using 80 threads across 1 machines.
NOTE: The algorithm found 5 matches.
NOTE: Processing the pattern matching query used 0.02 (cpu: 0.03) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.205574 seconds.
NOTE: The data set MYLIB.OUTMATCHNODES has 15 observations and 6 variables.
NOTE: The data set MYLIB.OUTMATCHLINKS has 15 observations and 4 variables.
STATUS=OK  PROBLEM_TYPE=PATTERNMATCH  SOLUTION_STATUS=OK  NUM_MATCHES=5  CPU_TIME=2.80
REAL_TIME=0.21

```

Output 2.15.2 displays the output data table `mylib.OutMatchNodes`, which shows the mappings from nodes in the query graph to nodes in the input graph for each match. For this query, five friends (X) match the specified criteria: Bryan, Chuck, Jack, Rob, and Stephen.

Output 2.15.2 Node Mappings for Friends Who Like Barbecue

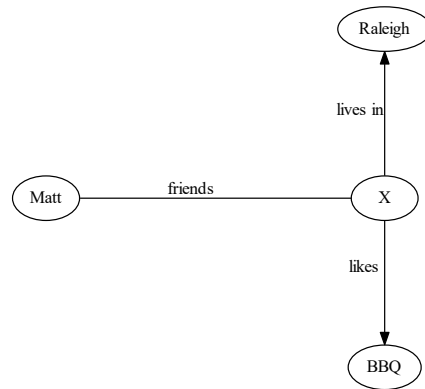
match	nodeQ	node	label	type	subtype
1	BBQ	Red Hot Blue	Red Hot Blue	Restaurant	BBQ
1	Matt	Matt	Matt	Person	
1	X	Bryan	Bryan	Person	
2	BBQ	The Pit Authentic	The Pit Authentic	Restaurant	BBQ
2	Matt	Matt	Matt	Person	
2	X	Chuck	Chuck	Person	
3	BBQ	Red Hot Blue	Red Hot Blue	Restaurant	BBQ
3	Matt	Matt	Matt	Person	
3	X	Jack	Jack	Person	
4	BBQ	The Pit Authentic	The Pit Authentic	Restaurant	BBQ
4	Matt	Matt	Matt	Person	
4	X	Rob	Rob	Person	
5	BBQ	JimmyJs	JimmyJs	Restaurant	BBQ
5	Matt	Matt	Matt	Person	
5	X	Stephen	Stephen	Person	

Output 2.15.3 displays the output data table `mylib.OutMatchLinks`, which shows the subgraphs for each match.

Output 2.15.3 Subgraphs for Friends Who Like Barbecue

match	from	to	connection
1	Bryan	Matt	friends
1	Bryan	Red Hot Blue	likes
1	Matt	Bryan	friends
2	Chuck	Matt	friends
2	Chuck	The Pit Authentic	likes
2	Matt	Chuck	friends
3	Jack	Matt	friends
3	Jack	Red Hot Blue	likes
3	Matt	Jack	friends
4	Matt	Rob	friends
4	Rob	Matt	friends
4	Rob	The Pit Authentic	likes
5	Matt	Stephen	friends
5	Stephen	JimmyJs	likes
5	Stephen	Matt	friends

Another example search pattern that you might want to find is “friends of Matt who like barbecue restaurants and live in Raleigh.” This pattern is shown in Figure 2.225.

Figure 2.225 Query Graph Q 

In order to construct this pattern, the query graph can be represented using the data that are created by the following DATA steps and the same call to PROC NETWORK as before:

```

data mylib.NodesSocialQuery;
  infile datalines dsd;
  length node $40. label $40. type $40. subtype $20.;
  input node $ label $ type $ subtype $;
  datalines;
Matt,    Matt,    Person,
X,,      Person,
Raleigh, Raleigh, City,

```

```

BBQ,,          Restaurant, BBQ
;
data mylib.LinksSocialQuery;
  infile datalines dsd;
  length from $40. to $40. connection $20.;
  input from $ to $ connection $;
  datalines;
Matt, X,      friends
X,    Matt,   friends
X,    Raleigh, lives in
X,    BBQ,    likes
;

```

Output 2.15.4 displays the output data table mylib.OutMatchNodes. For this query, three friends (X) match the specified criteria: Rob, Chuck, and Jack.

Output 2.15.4 Node Mappings for Friends Who Like Barbecue and Live in Raleigh

match	nodeQ	node	label	type	subtype
1	BBQ	The Pit Authentic	The Pit Authentic	Restaurant	BBQ
1	Matt	Matt	Matt	Person	
1	Raleigh	Raleigh	Raleigh	City	
1	X	Chuck	Chuck	Person	
2	BBQ	Red Hot Blue	Red Hot Blue	Restaurant	BBQ
2	Matt	Matt	Matt	Person	
2	Raleigh	Raleigh	Raleigh	City	
2	X	Jack	Jack	Person	
3	BBQ	The Pit Authentic	The Pit Authentic	Restaurant	BBQ
3	Matt	Matt	Matt	Person	
3	Raleigh	Raleigh	Raleigh	City	
3	X	Rob	Rob	Person	

Output 2.15.5 displays the output data table mylib.OutMatchLinks.

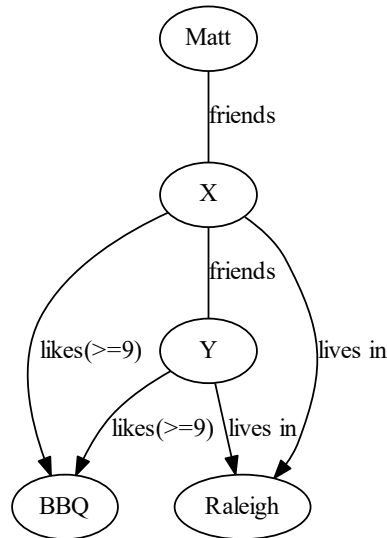
Output 2.15.5 Subgraphs for Friends Who Like Barbecue and Live in Raleigh

match	from	to	connection
1	Chuck	Matt	friends
1	Chuck	Raleigh	lives in
1	Chuck	The Pit Authentic	likes
1	Matt	Chuck	friends
2	Jack	Matt	friends
2	Jack	Raleigh	lives in
2	Jack	Red Hot Blue	likes
2	Matt	Jack	friends
3	Matt	Rob	friends
3	Rob	Matt	friends
3	Rob	Raleigh	lives in
3	Rob	The Pit Authentic	likes

Finally, you might want to find “a pair of people, at least one of whom is a friend of Matt, who like the same

barbecue restaurant (with rating of 9 or higher), live in Raleigh, and are friends of each other.” This pattern is shown in Figure 2.226.

Figure 2.226 Query Graph Q



In order to construct this pattern, the query graph can be represented using the data that are created by the following DATA steps:

```

data mylib.NodesSocialQuery;
  infile datalines dsd;
  length node $40. label $40. type $40. subtype $20.;
  input node $ label $ type $ subtype $;
  datalines;
Matt,    Matt,    Person,
X,,      Person,
Y,,      Person,
Raleigh, Raleigh, City,
BBQ,,    Restaurant, BBQ
;
data mylib.LinksSocialQuery;
  infile datalines dsd;
  length from $40. to $40. connection $20.;
  input from $ to $ connection $;
  datalines;
Matt, X,      friends
X,    Matt,   friends
X,    Raleigh, lives in
Y,    Raleigh, lives in
X,    BBQ,    likes
Y,    BBQ,    likes
X,    Y,      friends
  
```

```

Y,      X,      friends
;

```

The query node Matt must be a person with the node attribute label=Matt. The query nodes, X and Y, can be any pair of people, at least one of whom is a friend of Matt and who both live in Raleigh. The query node BBQ must be a barbecue restaurant (that is, type=Restaurant and subtype=BBQ) that is liked by both persons X and Y with a rating of at least 9. Person X and person Y must be friends with Matt.

In order to enforce that the restaurant was rated with a value of at least limitRating, you can use the following FCMP link filter function:

```

%macro linkPairFilterCode();
  function myLinkFilter(connectionQ $, rating, limitRating);
    if (connectionQ='likes') then return (rating >= limitRating);
    else return (1);
  endsub;
%mend linkPairFilterCode;

```

The following statements find all subgraphs that have the specified pattern:

```

proc network
  direction      = directed
  nodes          = mylib.NodesSocial
  links          = mylib.LinksSocial
  nodesQuery     = mylib.NodesSocialQuery
  linksQuery     = mylib.LinksSocialQuery;
  nodesVar
    vars         = (label type subtype);
  linksVar
    vars         = (connection rating);
  nodesQueryVar
    vars         = (label type subtype);
  linksQueryVar
    vars         = (connection);
  patternMatch
    code         = "%linkPairFilterCode()"
    linkFilter    = myLinkFilter(linksQuery.connection, links.rating, 9)
    outMatchNodes = mylib.OutMatchNodes
    outMatchLinks = mylib.OutMatchLinks;
run;

```

Output 2.15.6 displays the output data table mylib.OutMatchNodes. For this query, only one pair of friends (Jack and Natalia) matches the specified criteria.

Output 2.15.6 Node Mapping for a Pair of Friends

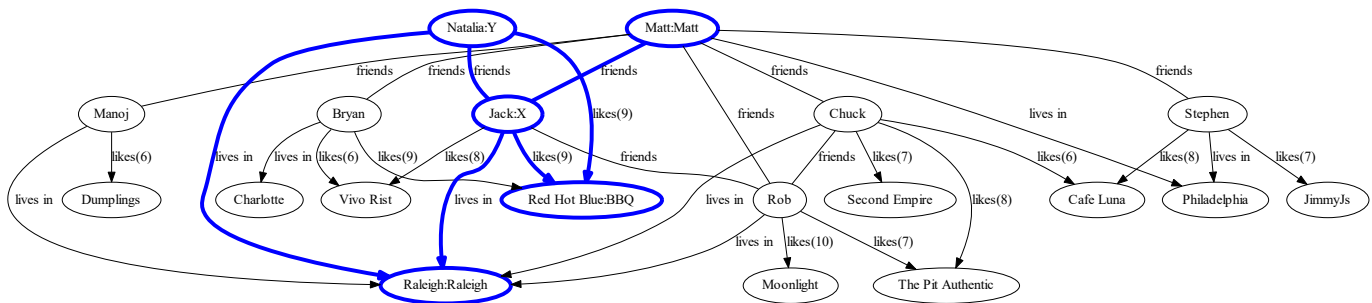
match	nodeQ	node	label	type	subtype
1	BBQ	Red Hot Blue	Red Hot Blue	Restaurant	BBQ
1	Matt	Matt	Matt	Person	
1	Raleigh	Raleigh	Raleigh	City	
1	X	Jack	Jack	Person	
1	Y	Natalia	Natalia	Person	

Output 2.15.7 displays the output data table mylib.OutMatchLinks.

Output 2.15.7 Subgraph for a Pair of Friends

match	from	to	connection	rating
1	Jack	Matt	friends	.
1	Jack	Natalia	friends	.
1	Jack	Raleigh	lives in	.
1	Jack	Red Hot Blue	likes	9
1	Matt	Jack	friends	.
1	Natalia	Jack	friends	.
1	Natalia	Raleigh	lives in	.
1	Natalia	Red Hot Blue	likes	9

The result is shown graphically in Figure 2.227.

Figure 2.227 Subgraph for a Pair of Friends**Example 2.16: Detection of Value-Added Tax Carousel Fraud**

Value-added tax (VAT) carousel fraud can be accomplished by a series of sequential transactions through a subset of real (or fictitious) corporations that starts and ends at the same entity. The fraud scheme starts when one corporation (1) in a European Union (EU) country—for example, Germany—sells its goods to another corporation (2) in a different EU country—for example, the United Kingdom (UK). Because this is a cross-border sale within the EU, corporation 1 does not charge corporation 2 any VAT. Next, corporation 2 sells these same goods to corporation 3 (also in the UK). Corporation 2 charges corporation 3 for VAT—for example, 20% of the price of the goods. Corporation 2 now owes the VAT authority its collected tax. Next, corporation 3 sells its goods back to corporation 1 (cross-border) with no VAT charge. Then, corporation 3 claims a VAT refund from the authority for the tax payment made to corporation 2. The payment for the refund by the tax authority occurs promptly because of laws that serve to protect the interests of smaller companies. At this point, all three corporations disappear before corporation 2 has paid the VAT authority for the taxes it has collected. By colluding, the three corporations illegally net the refunded money from the VAT authority. For more information about VAT carousel fraud, see Lamensch and Ceci (2018).

The following data provide a small subset of transactions between corporations in EU countries:

```

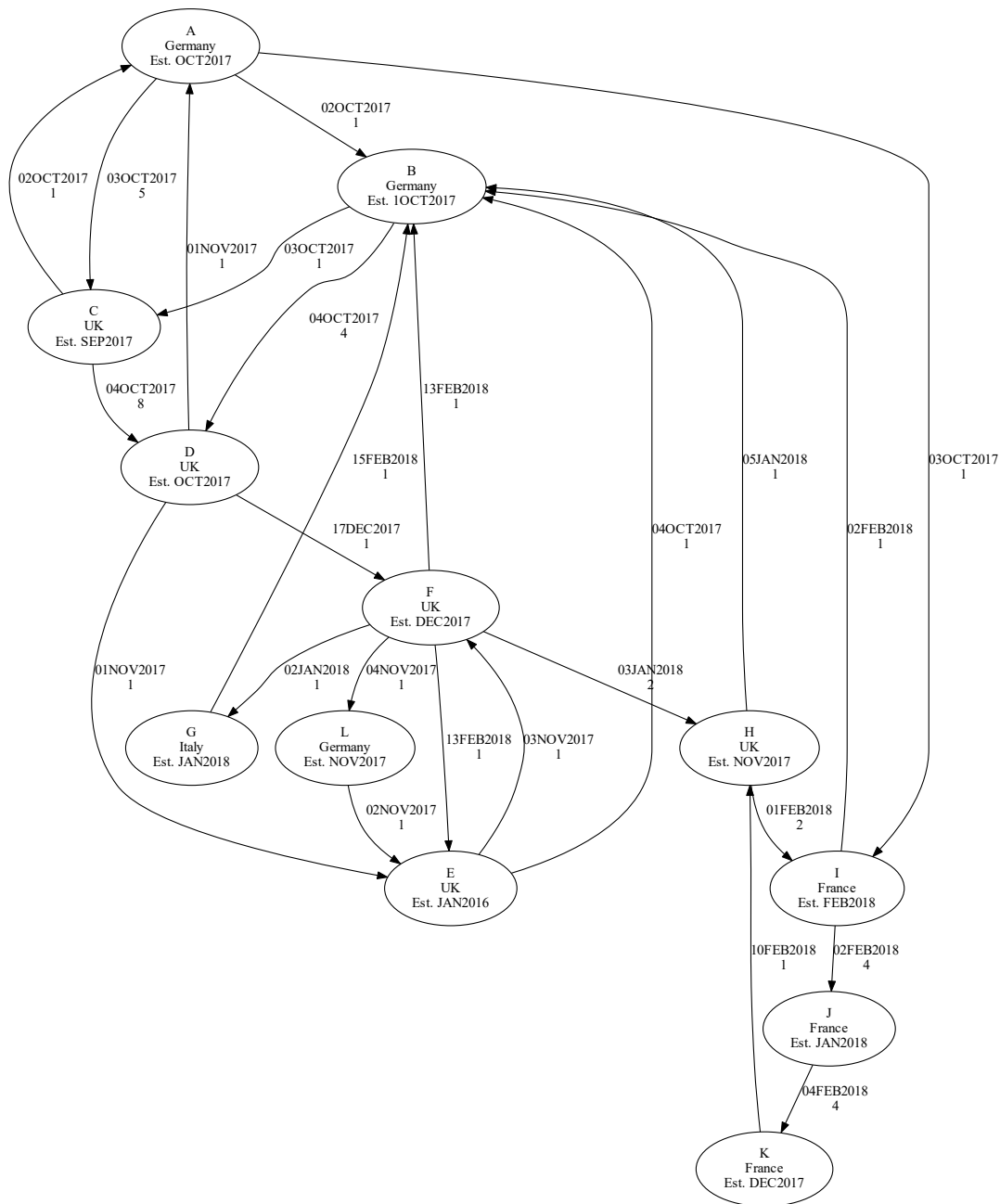
data mylib.NodesVAT;
    format est DATE9.;
    input node $ country $ est :DATE9.;
    datalines;
A Germany 01OCT2017
B Germany 01OCT2017
C UK      01SEP2017
D UK      01OCT2017
E UK      01JAN2016
F UK      01DEC2017
G Italy   01JAN2018
H UK      01NOV2017
I France  01FEB2018
J France  01JAN2018
K France  01DEC2017
L Germany 01NOV2017
;
data mylib.LinksVAT;
    format time DATE9.;
    input from $ to $ time :DATE9. score;
    datalines;
A B 02OCT2017 1
A C 03OCT2017 5
B C 03OCT2017 1
B D 04OCT2017 4
C A 02OCT2017 1
C D 04OCT2017 8
D A 01NOV2017 1
D E 01NOV2017 1
D F 17DEC2017 1
E B 04OCT2017 1
F B 13FEB2018 1
F E 13FEB2018 1
F G 02JAN2018 1
G B 15FEB2018 1
F H 03JAN2018 2
H B 05JAN2018 1
H I 01FEB2018 2
I B 02FEB2018 1
A I 03OCT2017 1
I J 02FEB2018 4
J K 04FEB2018 4
K H 10FEB2018 1
L E 02NOV2017 1
E F 03NOV2017 1
F L 04NOV2017 1
;

```

The nodes in the nodes data table `mylib.NodesVAT` represent corporations; the node attributes are host country (country) and the date the corporation was established (est). The links in the links data table `mylib.LinksVAT` represent transactions between the corporations; the link attributes are the date of the transaction (time) and a fraud score (score). The fraud score is some predetermined risk factor that is associated with certain transactions between corporations of interest.

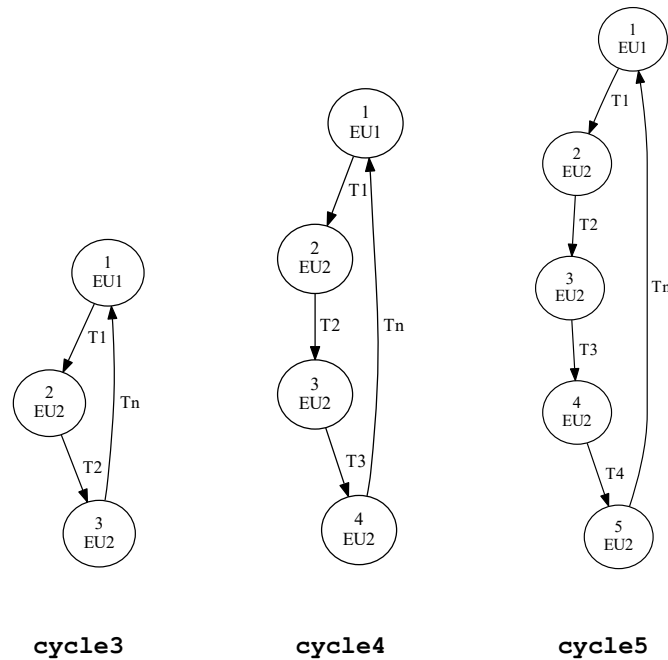
The data are shown graphically in Figure 2.228.

Figure 2.228 Transactions between Corporations in EU Countries



The carousel fraud scheme involves a cycle of sequential transactions that starts and ends in some EU country and have any number of transactions within the borders of some other EU country. In this example, consider cycles of length between three and five links. The query graphs that define the patterns to search for are shown in Figure 2.229. In addition, the pattern requires that $T1 < T2 < \dots < Tn$.

Figure 2.229 Query Graphs



In order to construct this set of patterns (**cycle3** through **cycle5**), the query graphs can be represented using the data that are created by the following DATA steps:

```
data mylib.NodesQuery;
  input node key $ @@;
  datalines;
1 cycle3 2 cycle3 3 cycle3
1 cycle4 2 cycle4 3 cycle4 4 cycle4
1 cycle5 2 cycle5 3 cycle5 4 cycle5 5 cycle5
;
data mylib.LinksQuery;
  input from to key $ @@;
  datalines;
1 2 cycle3 2 3 cycle3 3 1 cycle3
1 2 cycle4 2 3 cycle4 3 4 cycle4 4 1 cycle4
1 2 cycle5 2 3 cycle5 3 4 cycle5 4 5 cycle5 5 1 cycle5
;
```

In addition to searching for cycles of transactions, investigators need to identify several other characteristic patterns in order to consider a sequence of transactions that are at risk for VAT carousel fraud.

First, each corporation in the cycle must have been established in the past six months. Most corporations involved in VAT carousel fraud are fake corporations that are established for the sake of carrying forward the fraudulent transactions. Because these schemes are difficult to hide for long, corporations that are involved are usually recently established.

The following statements specify a node filter function (**myNodeFilter**) that enforces that each node in the pattern was established within **limitMonths** months of **startDate**:

```
%macro nodeFilterCode();
    function myNodeFilter(est, limitMonths, startDate);
        /* All of the nodes in the cycle were established in the past
           limitMonths months (from startDate). */
        return (intck('MONTH',est,startDate) < limitMonths);
    endsub;
%mend nodeFilterCode;
```

The fraud scheme involves a cycle of transactions that start and end in some EU country and have any number of transactions within the borders of some other EU country. The following statements specify a node-pair filter function (**myNodePairFilter**) that enforces this pattern of transactions:

```
%macro nodePairFilterCode();
    function myNodePairFilter(nodeQ[*], country[*] $);
        /* The first node must be from a different country than the other nodes,
           and each subsequent node must be from the same country. */
        if (nodeQ[1] = 1) then
            return (country[1] ne country[2]);
        else if (nodeQ[2] ne 1) then
            return (country[1] = country[2]);
        else return (1);
        endsub;
%mend nodePairFilterCode;
```

Because the suspicious pattern involves tracking a sale of goods, the transactions must be sequential in time. In addition, the entire series of transactions should occur in a relatively short period of time. The following statements specify a link-pair filter function (**myLinkPairFilter**) that enforces that the transactions occur sequentially and that the time between the first and last transaction must be within limitDays days:

```
%macro linkPairFilterCode();
    function myLinkPairFilter(fromQ[*], toQ[*], time[*], limitDays);
        /* All transactions must be sequential in time.
           The time between the first and last transactions must be
           less than limitDays days. */
        if (toQ[1] = 1) then
            return (1);
        else if (toQ[1] = fromQ[2]) then
            return (time[1] < time[2]);
        else if (fromQ[1] = 1 and toQ[2] = 1) then
            return (intck('DAY',time[1],time[2]) < limitDays);
        else return (1);
        endsub;
%mend linkPairFilterCode;
```

The final requirement that signals a suspicious series of transactions is the overall fraud score. Calculating this value involves looking at the entire candidate subgraph and therefore requires a match filter function. The following statements specify a match filter function (**myMatchFilter**) that enforces that the total fraud score across all links in the cycle of transactions must be greater than or equal to limitScore:

```
%macro matchFilterCode();
    function myMatchFilter(score[*], limitScore);
        /* The total fraud score across all links in the cycle of transactions
           must be greater than or equal to limitScore.

```

```

        This function assumes all score values are nonnegative. */
total = 0;
nLinks = dim(score);
do i=1 to nLinks;
    total = total + score[i];
    if (total >= limitScore) then return (1);
end;
return (0);
endsub;
%mend matchFilterCode;

```

The following statements find all subgraphs that have the pattern that is specified by the query input data tables and the FCMP filter functions:

```

%let startDate=%sysvalf('01FEB2018'd);
options noQuoteLenMax;
proc network
    direction          = directed
    nodes               = mylib.NodesVAT
    links               = mylib.LinksVAT
    nodesQuery          = mylib.NodesQuery
    linksQuery          = mylib.LinksQuery;
    nodesVar
        vars            = (country est);
    linksVar
        vars            = (time score);
    patternMatch
        code            =
        "
            %nodeFilterCode()
            %nodePairFilterCode()
            %linkPairFilterCode()
            %matchFilterCode()
        "
    queryKey            = key
    nodeFilter           = myNodeFilter(nodes.est,6,&startDate)
    nodePairFilter       = myNodePairFilter(nodesQuery.node,nodes.country)
    linkPairFilter       = myLinkPairFilter(linksQuery.from,linksQuery.to,links.time,100)
    matchFilter          = myMatchFilter(links.score,10)
    outMatchNodes        = mylib.OutMatchNodes
    outMatchLinks        = mylib.OutMatchLinks
    outSummary           = mylib.OutMatchSummary;
run;
%put &_NETWORK_;

```

The progress of the procedure is shown in [Output 2.16.1](#).

Output 2.16.1 PROC NETWORK Log: Pattern Matching for VAT Carousel Fraud

```

NOTE: -----
NOTE: Running NETWORK.
NOTE: -----
NOTE: The number of nodes in the input graph is 12.
NOTE: The number of links in the input graph is 25.
NOTE: Processing pattern matching using 1200 threads across 15 machines.
NOTE: The algorithm found 3 matches.
NOTE: Processing the pattern matching query used 0.03 (cpu: 0.10) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.26916 seconds.
NOTE: The data set MYLIB.OUTMATCHNODES has 12 observations and 6 variables.
NOTE: The data set MYLIB.OUTMATCHLINKS has 12 observations and 6 variables.
NOTE: The data set MYLIB.OUTMATCHSUMMARY has 3 observations and 5 variables.
STATUS=OK  PROBLEM_TYPE=PATTERNMATCH  SOLUTION_STATUS=OK  NUM_MATCHES=3  CPU_TIME=2.91
REAL_TIME=0.27

```

For these data, three sequences of transactions match the specified pattern.

Output 2.16.2 displays the output data table mylib.OutMatchSummary, which shows the summary information about the executed queries.

Output 2.16.2 Summary Information for Executed Queries

key	nodes	links	matches	realTime
cycle3	3	3	1	0.028705
cycle4	4	4	1	0.019045
cycle5	5	5	1	0.022385

Output 2.16.3 displays the output data table mylib.OutMatchNodes, which shows the mappings from nodes in the query graph to nodes in the input graph for each matching sequence of transactions.

Output 2.16.3 Node Mappings for Suspicious Transactions

key	match	nodeQ	node	country	est
cycle3	1	1	A	Germany	01OCT2017
cycle3	1	2	C	UK	01SEP2017
cycle3	1	3	D	UK	01OCT2017
cycle4	1	1	H	UK	01NOV2017
cycle4	1	2	I	France	01FEB2018
cycle4	1	3	J	France	01JAN2018
cycle4	1	4	K	France	01DEC2017
cycle5	1	1	B	Germany	01OCT2017
cycle5	1	2	C	UK	01SEP2017
cycle5	1	3	D	UK	01OCT2017
cycle5	1	4	F	UK	01DEC2017
cycle5	1	5	H	UK	01NOV2017

Output 2.15.3 displays the output data table mylib.OutMatchLinks, which shows the subgraphs for each matching sequence of transactions.

Output 2.16.4 Subgraphs for Suspicious Transactions

key	match	from	to	time	score
cycle3	1	A	C	03OCT2017	5
cycle3	1	C	D	04OCT2017	8
cycle3	1	D	A	01NOV2017	1
cycle4	1	H	I	01FEB2018	2
cycle4	1	I	J	02FEB2018	4
cycle4	1	J	K	04FEB2018	4
cycle4	1	K	H	10FEB2018	1
cycle5	1	B	C	03OCT2017	1
cycle5	1	C	D	04OCT2017	8
cycle5	1	D	F	17DEC2017	1
cycle5	1	F	H	03JAN2018	2
cycle5	1	H	B	05JAN2018	1

The result is shown graphically in [Figure 2.230](#) through [Figure 2.232](#).

Figure 2.230 Subgraph for Suspicious Transaction (cyc1e3)

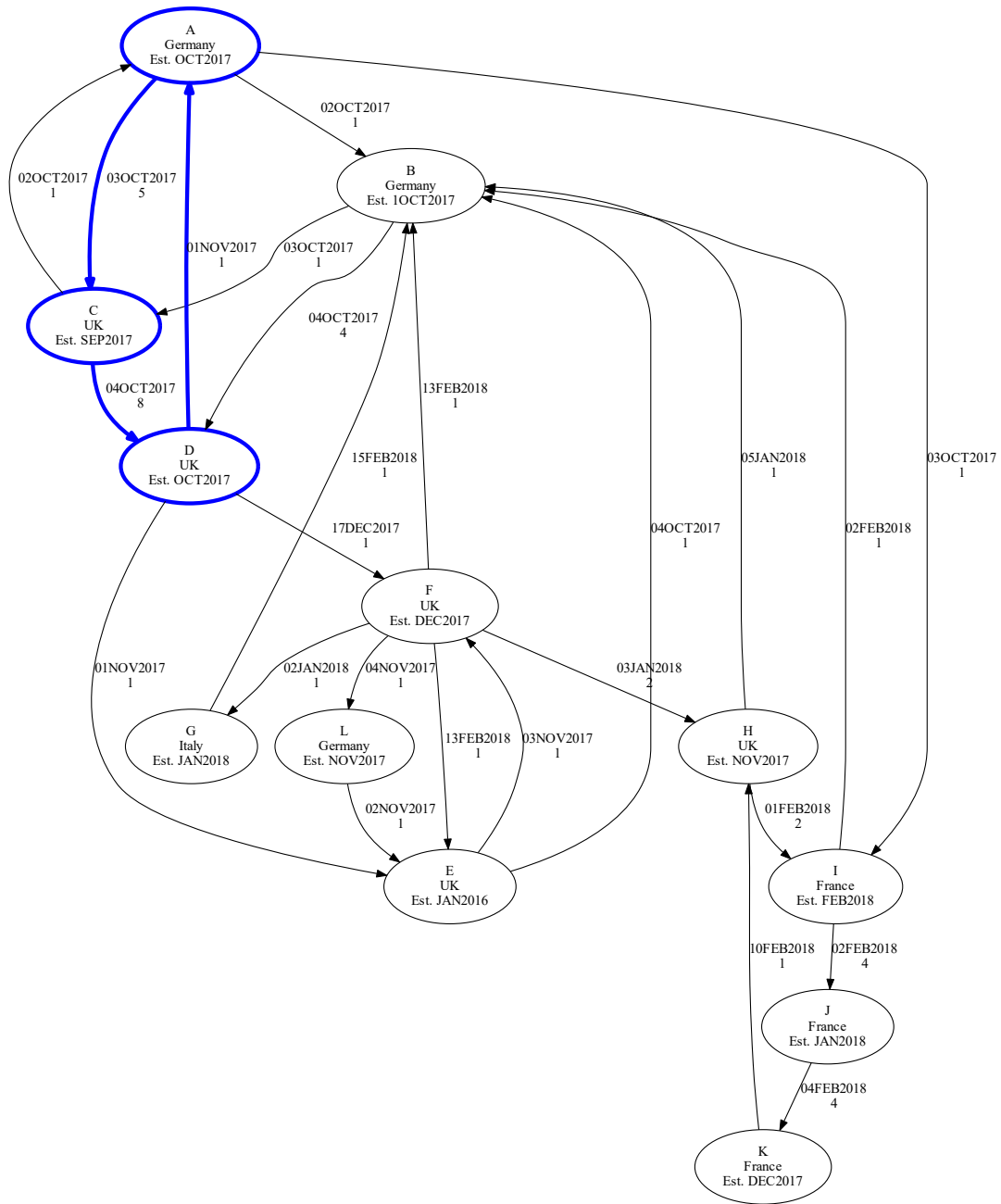


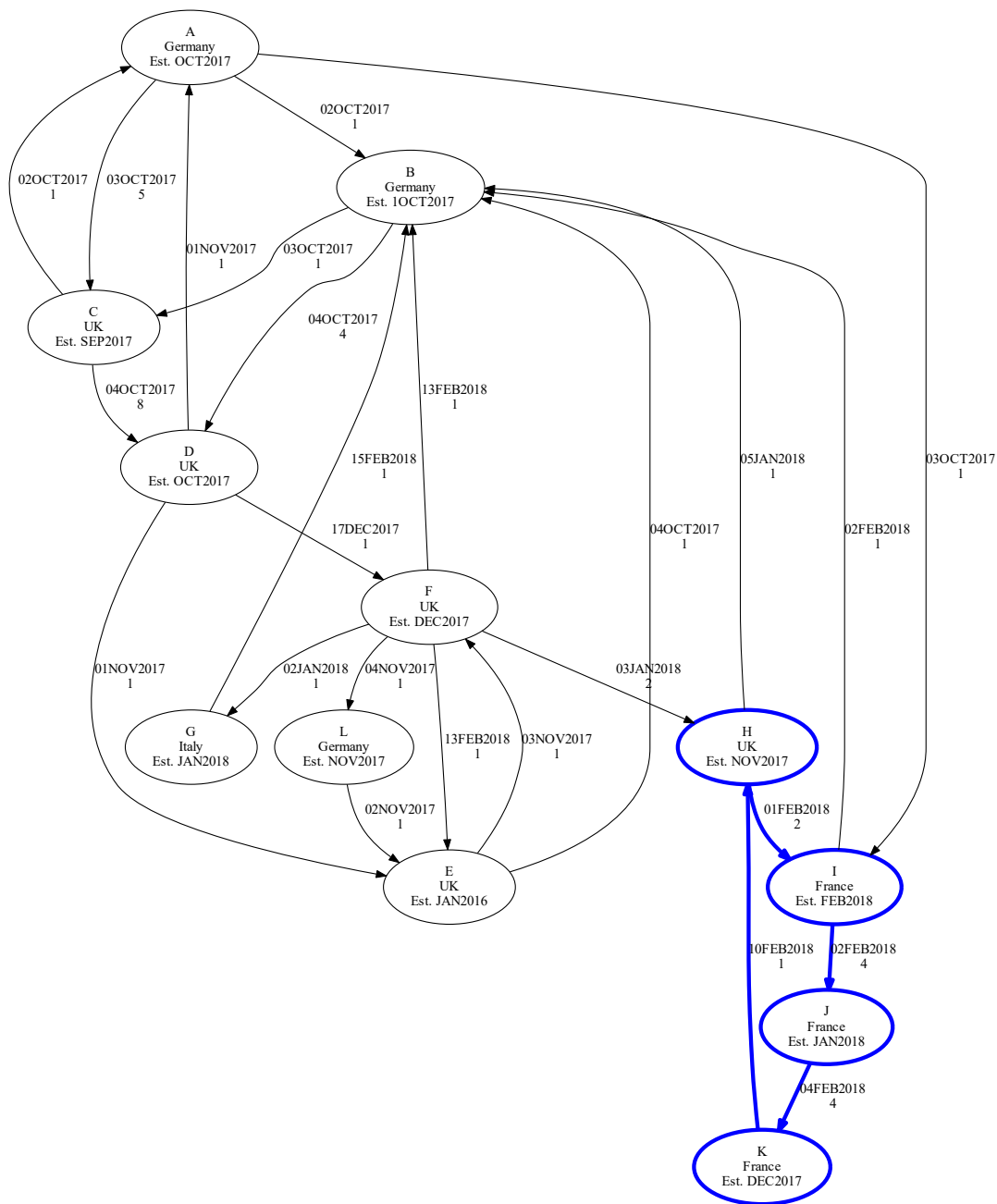
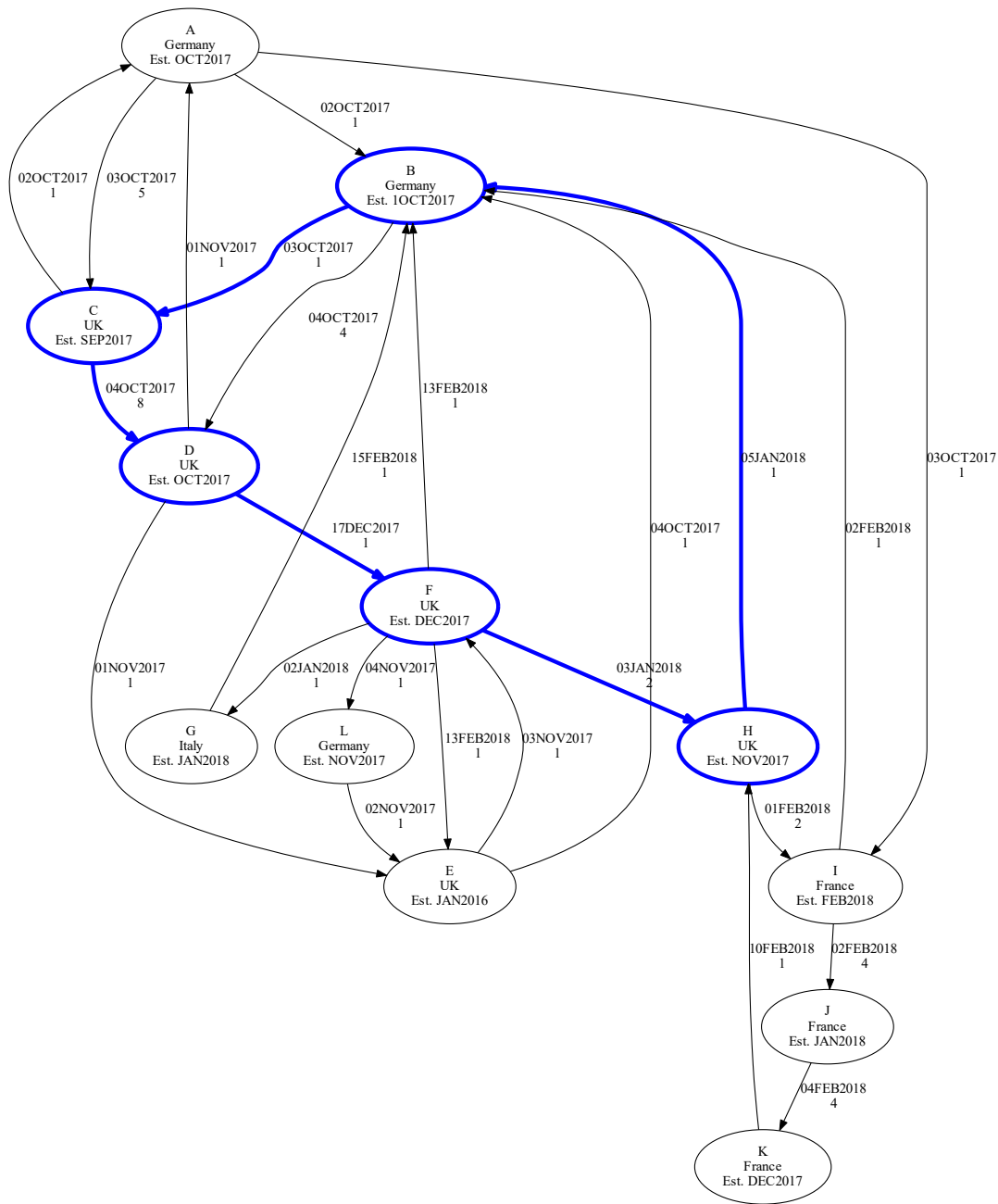
Figure 2.231 Subgraph for Suspicious Transaction (cyc1e4)

Figure 2.232 Subgraph for Suspicious Transaction (`cyc1e5`)

This example uses the `QUERYKEY=` option to search for sequential cycles of transactions whose lengths range from three to five links. To construct these patterns, you explicitly define all the nodes and links for all three query graphs. Alternatively, you can use the `EXPANDLOWER=` and `EXPANDUPPER=` options in the `LINKSQUERYVAR` statement to automatically generate the query graphs that represent the three cycles from one base query graph.

To construct this set of patterns by using path expansion, you can use the following `DATA` steps to represent

the query graphs:

```
data mylib.NodesQuery;
    input node $ @@;
    datalines;
1 2 3
;
data mylib.LinksQuery;
    input from $ to $ expandL expandU;
    datalines;
1 2 1 3
2 3 . .
3 1 . .
;
```

Because the node labels in the base query graph are character type, the node-pair and link-pair filter methods are updated as follows:

```
%macro nodePairFilterCode();
    function myNodePairFilter(nodeQ[*] $, country[*] $);
        /* The first node must be from a different country than the other nodes,
           and each subsequent node must be from the same country. */
        if (nodeQ[1] = '1') then
            return (country[1] ne country[2]);
        else if (nodeQ[2] ne '1') then
            return (country[1] = country[2]);
        else return (1);
    endsub;
%mend nodePairFilterCode;
%macro linkPairFilterCode();
    function myLinkPairFilter(fromQ[*] $, toQ[*] $, time[*], limitDays);
        /* All transactions must be sequential in time.
           The time between the first and last transactions must be
           less than limitDays days. */
        if (toQ[1] = '1') then
            return (1);
        else if (toQ[1] = fromQ[2]) then
            return (time[1] < time[2]);
        else if (fromQ[1] = '1' and toQ[2] = '1') then
            return (intck('DAY',time[1],time[2]) < limitDays);
        else return (1);
    endsub;
%mend linkPairFilterCode;
```

The following statements find all subgraphs that have the pattern that is specified by the expanded query graphs and the FCMP filter functions:

```
options noQuoteLenMax;
proc network
    direction      = directed
    nodes           = mylib.NodesVAT
    links           = mylib.LinksVAT
    nodesQuery      = mylib.NodesQuery
    linksQuery      = mylib.LinksQuery;
    nodesVar
```

```

    vars          = (country est);
linksVar
    vars          = (time score);
linksQueryVar
    expandLower    = expandL
    expandUpper    = expandU;
patternMatch
    code          =
    "
        %nodeFilterCode()
        %nodePairFilterCode()
        %linkPairFilterCode()
        %matchFilterCode()
    "
    nodeFilter     = myNodeFilter(nodes.est, 6, &startDate)
    nodePairFilter = myNodePairFilter(nodesQuery.node, nodes.country)
    linkPairFilter = myLinkPairFilter(linksQuery.from, linksQuery.to, links.time, 100)
    matchFilter    = myMatchFilter(links.score, 10)
    outMatchNodes  = mylib.OutMatchNodes
    outMatchLinks  = mylib.OutMatchLinks
    outQueryNodes  = mylib.OutQueryNodes
    outQueryLinks  = mylib.OutQueryLinks
    outSummary     = mylib.OutMatchSummary;
run;
%put &_NETWORK_;

```

The output data table `mylib.OutQueryNodes` contains the nodes of the query graphs that are constructed from the expansion of the base query graph, as shown in [Figure 2.233](#).

Figure 2.233 Query Graph Nodes

queryKey	node
key_1	1
key_1	2_1
key_1	3
key_2	1
key_2	2_1
key_2	2_2
key_2	3
key_3	1
key_3	2_1
key_3	2_2
key_3	2_3
key_3	3

The output data table `mylib.OutQueryLinks` contains the links of the query graphs that are constructed from the expansion of the base query graph, as shown in [Figure 2.234](#).

Figure 2.234 Query Graph Links

queryKey	from	to
key_1	1	2_1
key_1	2_1	3
key_1	3	1
key_2	1	2_1
key_2	2_1	2_2
key_2	2_2	3
key_2	3	1
key_3	1	2_1
key_3	2_1	2_2
key_3	2_2	2_3
key_3	2_3	3
key_3	3	1

Output 2.16.5 displays the output data table mylib.OutMatchSummary, which shows the summary information about the executed queries.

Output 2.16.5 Summary Information for Executed Queries

queryKey	nodes	links	matches	realTime
key_1	3	3	1	0.022659
key_2	4	4	1	0.029016
key_3	5	5	1	0.024120

Output 2.16.6 displays the output data table mylib.OutMatchNodes, which shows the mappings from nodes in the query graph to nodes in the input graph for each matching sequence of transactions.

Output 2.16.6 Node Mappings for Suspicious Transactions

queryKey	match	nodeQ	node	country	est
key_1	1 1	A	Germany	01OCT2017	
key_1	1 2_1	C	UK	01SEP2017	
key_1	1 3	D	UK	01OCT2017	
key_2	1 1	H	UK	01NOV2017	
key_2	1 2_1	I	France	01FEB2018	
key_2	1 2_2	J	France	01JAN2018	
key_2	1 3	K	France	01DEC2017	
key_3	1 1	B	Germany	01OCT2017	
key_3	1 2_1	C	UK	01SEP2017	
key_3	1 2_2	D	UK	01OCT2017	
key_3	1 2_3	F	UK	01DEC2017	
key_3	1 3	H	UK	01NOV2017	

Output 2.15.3 displays the output data table mylib.OutMatchLinks, which shows the subgraphs for each matching sequence of transactions.

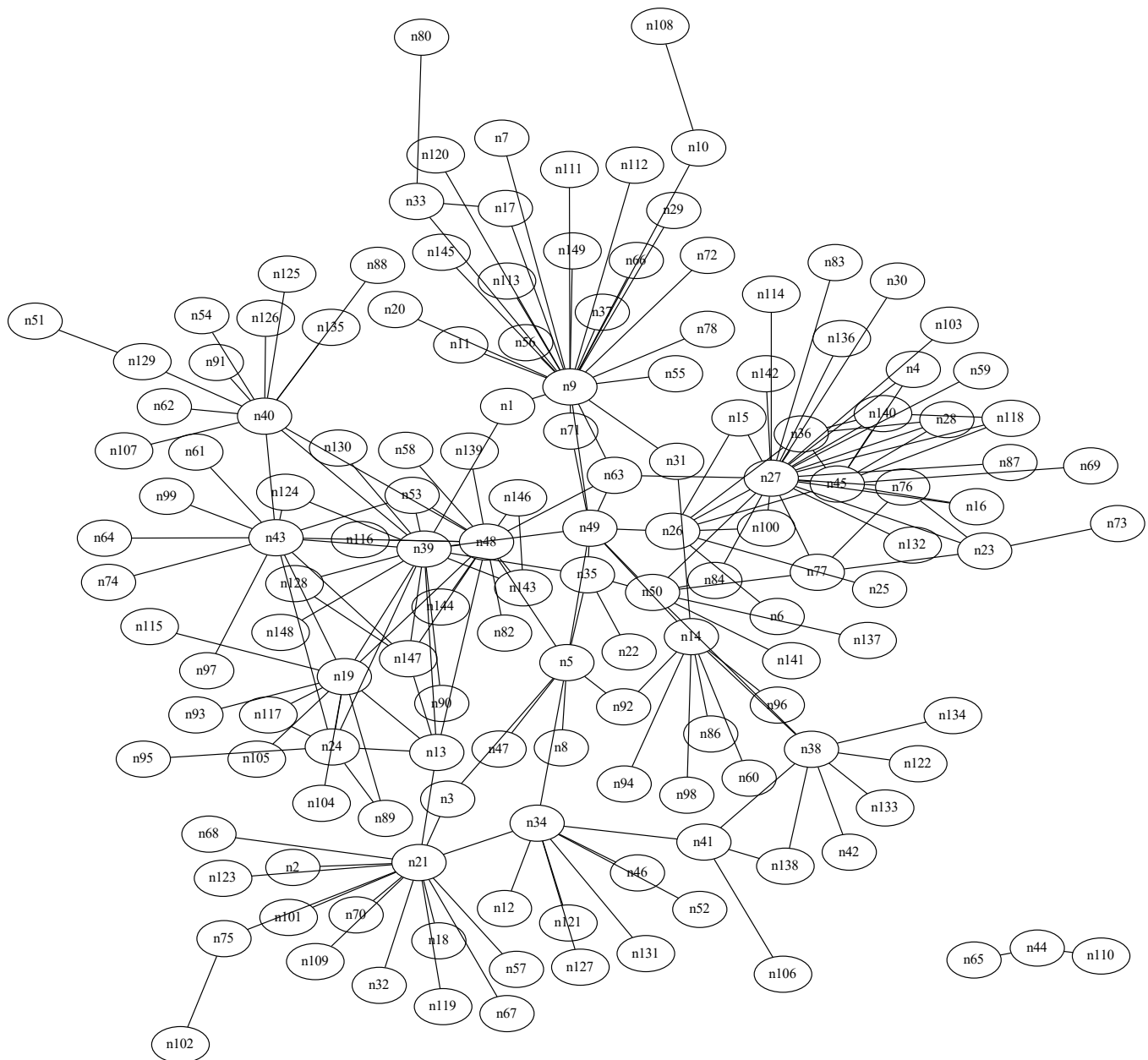
Output 2.16.7 Subgraphs for Suspicious Transactions

queryKey	match	from	to	time	score
key_1	1	A	C	03OCT2017	5
key_1	1	C	D	04OCT2017	8
key_1	1	D	A	01NOV2017	1
key_2	1	H	I	01FEB2018	2
key_2	1	I	J	02FEB2018	4
key_2	1	J	K	04FEB2018	4
key_2	1	K	H	10FEB2018	1
key_3	1	B	C	03OCT2017	1
key_3	1	C	D	04OCT2017	8
key_3	1	D	F	17DEC2017	1
key_3	1	F	H	03JAN2018	2
key_3	1	H	B	05JAN2018	1

Example 2.17: Node Similarity for Link Prediction

The goal of link prediction is to recover missing links—connections that are likely to exist between nodes but have not been recorded because of incomplete data. This example considers a network of 168 nodes and 219 links, where nodes represent individuals and links represent wiretap records. The data set is based on information from operation *Oversize*, an Italian criminal case against a mafia group that was involved in international drug trafficking, homicides, and robberies (Berlusconi et al. 2016). This network, an undirected graph G , is shown in [Figure 2.235](#).

Figure 2.235 Wiretap Network G



The data that are created by the following DATA step provide the connections that were obtained through wiretapping:

```
data mylib.LinkSetInWR;
  input from $ to $ @@;
  datalines;
n8   n5   n9   n1   n9   n7   n10  n9   n11  n9   n17  n9
n19  n13  n20  n9   n21  n3   n21  n13  n21  n18  n24  n13
```



```

n24 n19    n26 n6      n26 n15    n26 n25    n27 n15    n27 n16
n27 n23    n27 n26    n28 n27    n29 n9     n30 n27    n31 n9
n32 n21    n33 n9     n33 n17    n34 n5     n34 n12    n34 n21
n35 n5     n35 n22    n36 n27    n36 n28    n37 n9     n38 n14
n39 n1     n39 n13    n39 n19    n39 n24    n40 n39    n41 n34
n41 n38    n42 n38    n43 n19    n43 n24    n43 n39    n43 n40
n45 n16    n45 n26    n45 n27    n45 n28    n45 n36    n46 n34
n47 n5     n48 n5     n48 n19    n48 n39    n48 n40    n48 n43
n49 n5     n49 n9     n49 n14    n49 n26    n49 n39    n50 n27
n50 n35    n50 n38    n50 n49    n52 n34    n53 n39    n53 n43
n54 n40    n55 n9     n56 n9     n57 n21    n58 n48    n59 n27
n60 n14    n61 n43    n63 n9     n63 n27    n63 n48    n63 n49
n64 n43    n65 n44    n66 n9     n67 n21    n69 n45    n70 n21
n71 n9     n71 n49    n72 n9     n73 n23    n74 n43    n75 n21
n76 n27    n77 n23    n77 n27    n77 n50    n77 n76    n78 n9
n80 n33    n82 n48    n84 n27    n84 n50    n86 n14    n87 n27
n88 n40    n89 n19    n89 n24    n90 n39    n92 n5     n92 n14
n93 n19    n94 n14    n95 n24    n96 n14    n97 n43    n98 n14
n100 n26   n100 n27   n101 n21   n102 n75   n103 n27   n104 n19
n105 n19   n106 n41   n108 n10   n109 n21   n110 n44   n111 n9
n112 n9    n113 n9    n114 n27   n115 n19   n117 n19   n117 n24
n118 n27   n118 n45   n119 n21   n120 n9    n121 n34   n122 n38
n124 n39   n124 n43   n125 n40   n126 n40   n127 n34   n128 n39
n129 n40   n129 n51   n131 n34   n132 n27   n133 n38   n134 n38
n135 n40   n136 n27   n137 n50   n138 n38   n139 n48   n140 n27
n140 n36   n140 n45   n140 n118  n141 n50   n142 n27   n143 n39
n144 n48   n145 n9    n146 n48   n146 n143  n147 n13   n147 n39
n147 n43   n147 n48   n148 n39   n149 n9    n149 n43   n150 n48
n151 n9    n152 n34   n152 n50   n153 n34   n155 n14   n155 n133
n156 n9    n157 n9    n158 n9    n158 n49   n158 n133  n159 n9
n160 n35   n160 n49   n160 n81   n160 n84   n160 n85   n161 n40
n161 n43   n161 n125  n162 n9    n163 n27   n164 n33   n165 n9
n166 n35   n167 n40   n168 n40   n169 n40   n171 n43   n172 n19
n173 n79   n174 n27   n175 n27   n176 n9    n177 n27   n178 n49
n180 n14   n181 n27   n182 n44
;

```

The persons of interest—the two main traffickers within the criminal network, *n49* and *n27*; the drug dealer *n13*; *n49*'s brother and drug dealer, *n50*; and *n118*, the wife of a known drug dealer—are selected as a node subset to probe what other people in the network they are likely to be connected to.

The following statements define the nodes that are the sources of the new predicted links, with all other nodes as candidates:

```

data mylib.NodeSubSetIn;
  input node $ source;
  datalines;
n27 1
n49 1
n13 1
n50 1
n118 1
;

```

The following statements produce the output data table `mylib.NodeSim`, which contains the Jaccard similarity between the nodes in the input subset and all other nodes; higher similarity scores indicate a higher probability of connection.

```
proc network
  links          = mylib.LinkSetInWR
  nodesSubset    = mylib.NodeSubSetIn;
  nodeSimilarity
    outSimilarity = mylib.NodeSim;
run;
```

To select the best candidate links for investigators, the following steps determine five links that have the highest Jaccard coefficient for each of the nodes in the subset:

```
data NodeSim;
  set mylib.NodeSim;
run;
data LinkSetInWR;
  set mylib.LinkSetInWR;
run;

proc sql;
  create table PredictedLinks as
  select * from NodeSim
  where link=0 and source ne sink;
quit;
proc sort data=PredictedLinks;
  by source descending jaccard;
run;

data PredictedLinksTop5;
  retain count 0;
  set PredictedLinks;
  by source;
  if first.source then count=0;
  count=count+1;
  if count <= 5;
run;
```

Output 2.17.1 displays the output data table `PredictedLinksTop5`.

Output 2.17.1 Top Predicted Links between Suspected Criminals

source=n118			
count	sink	link	jaccard
1	n36	0	0.75000
2	n16	0	0.66667
3	n28	0	0.50000
4	n103	0	0.33333
5	n114	0	0.33333

Output 2.17.1 *continued***source=n13**

count	sink	link	jaccard
1	n117	0	0.40
2	n89	0	0.40
3	n43	0	0.25
4	n101	0	0.20
5	n104	0	0.20

source=n27

count	sink	link	jaccard
1	n49	0	0.083333
2	n137	0	0.035714
3	n141	0	0.035714
4	n25	0	0.035714
5	n6	0	0.035714

source=n49

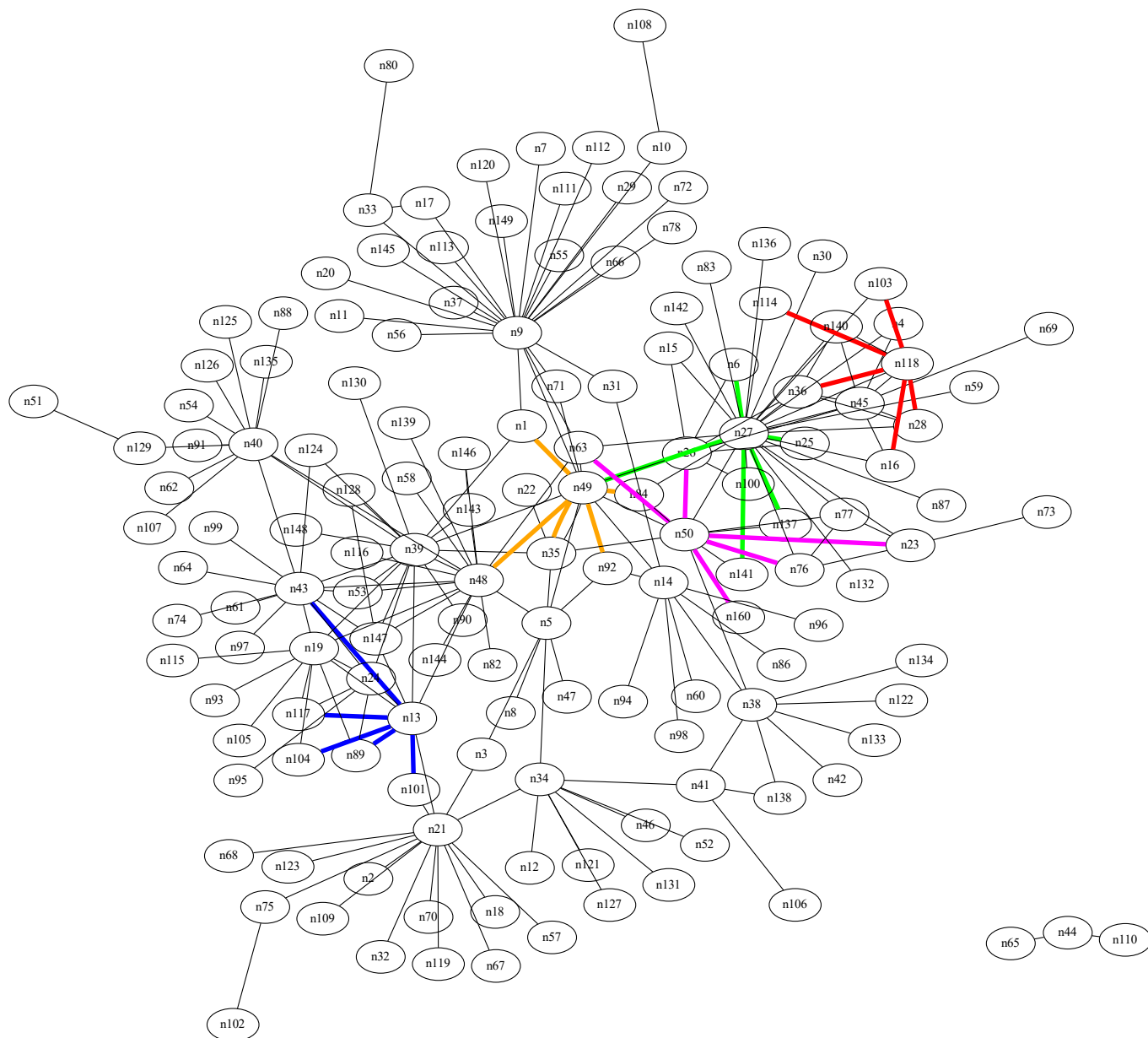
count	sink	link	jaccard
1	n35	0	0.23077
2	n1	0	0.18182
3	n92	0	0.18182
4	n84	0	0.16667
5	n48	0	0.14286

source=n50

count	sink	link	jaccard
1	n160	0	0.27273
2	n76	0	0.22222
3	n23	0	0.20000
4	n63	0	0.18182
5	n26	0	0.14286

The predicted links are highlighted as bold and colored (by source node) in the illustration in [Figure 2.236](#). Many of these predicted links have additional support from other sources of information, according to Berlusconi et al. (2016). For example, *n49* and *n48* lived in the same area and had key roles in the drug distribution; the husband of *n118* used to buy cocaine from *n36*; the link between *n13* and *n43* had been confirmed by further investigation; and so on.

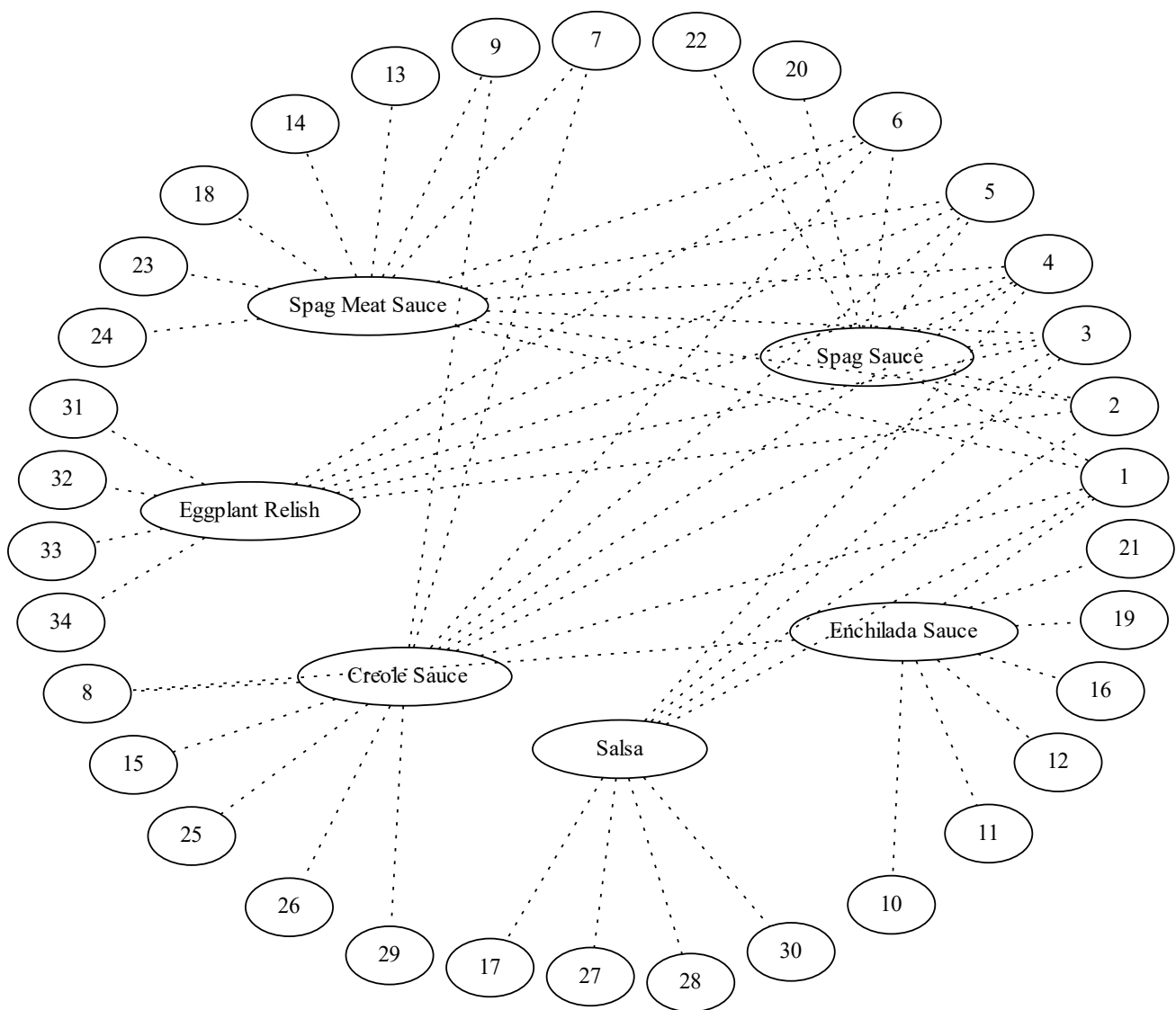
Figure 2.236 Predicted Links



Example 2.18: A Node Similarity Recipe Recommendation Engine

When recommending products to consumers, retailers often build models that are based on known relationships between consumers and products, such as purchase history or online reviews. These models are known as recommendation engines. This example builds a model that recommends recipes that are most similar and most dissimilar from each other, based on the similarity of their ingredient lists. The data set consists of six sauce recipes, each consisting of some subset of ingredients. The inclusion of ingredients in recipes is organized into a bipartite network G , shown in Figure 2.237.

Figure 2.237 Recipe-Ingredient Network G



The links of *G* are constructed from raw data that are specified in adjacency list format in the DATA step:

```
data mylib.LinkSetRecipesIn;
  infile datalines dsd flowover;
  length Recipe $20. Ingredient1-Ingredient13 $20.;
  input Recipe $ Ingredient1-Ingredient13 $;
  array ingredient Ingredient1 -- Ingredient13;
  from = Recipe;
  do over ingredient;
    if missing(ingredient) then leave;
    to = Ingredient;
    output;
  end;
  keep from to;
  datalines;
Spag Sauce,      Tomato,Garlic,Salt,Onion,TomatoPaste,OliveOil
                  Oregano,Parsley, , , , ,
Spag Meat Sauce, Tomato,Garlic,Salt,Onion,TomatoPaste,OliveOil
                  Celery,GreenPepper,BayLeaf,GroundBeef,Carrot
                  PorkSausage,RedPepper
Eggplant Relish, Garlic,Salt,Onion,TomatoPaste,OliveOil,Eggplant
                  GreenOlives,Capers,Sugar, , , ,
Creole Sauce,    Tomato,Salt,Onion,TomatoPaste,OliveOil,Celery
                  Broth,GreenPepper,BlackPepper,Paprika,Thyme
                  WorcestershireSauce,
Salsa,           Tomato,Garlic,Salt,Onion,Cilantro,Tomatillo
                  JalapenoPepper,Lime, , , , ,
Enchilada Sauce, Tomato,Broth,Cumin,Flour,BrownSugar,ChiliPowder
                  CayennePepper,Oil, , , , ,
;
```

In this example, the cosine node similarity scores between pairs of recipes are of interest.

The following DATA step defines the subset of nodes that represent the six recipes:

```
data mylib.NodesSubsetRecipesIn;
  infile datalines dsd;
  length node $20.;
  source = 1;
  sink = 1;
  input node;
  datalines;
Spag Sauce
Spag Meat Sauce
Eggplant Relish
Creole Sauce
Salsa
Enchilada Sauce
;
```

The following statements produce the output data table *mylib.NodeSim*, which contains the cosine similarity scores between the nodes in the input subset; higher similarity scores indicate greater overlap of the recipes' ingredient lists.

```

proc network
  links          = mylib.LinkSetRecipesIn
  nodesSubset    = mylib.NodesSubsetRecipesIn;
  nodeSimilarity
    jaccard      = false
    cosine       = true
    outSimilarity = mylib.SimilarityRecipesOut;
run;

```

The output data table `mylib.SimilarityRecipesOut` contains the cosine similarity score of each recipe node pair, as shown in [Output 2.18.1](#).

Output 2.18.1 Recipe Cosine Similarity Output

source	sink	cosine	source	sink	cosine
Creole Sauce	Creole Sauce	1.00000	Salsa	Salsa	1.00000
Creole Sauce	Spag Meat Sauce	0.56045	Salsa	Spag Sauce	0.50000
Creole Sauce	Spag Sauce	0.51031	Salsa	Spag Meat Sauce	0.39223
Creole Sauce	Eggplant Relish	0.38490	Salsa	Eggplant Relish	0.35355
Creole Sauce	Salsa	0.30619	Salsa	Creole Sauce	0.30619
Creole Sauce	Enchilada Sauce	0.20412	Salsa	Enchilada Sauce	0.12500
Eggplant Relish	Eggplant Relish	1.00000	Spag Meat Sauce	Spag Meat Sauce	1.00000
Eggplant Relish	Spag Sauce	0.58926	Spag Meat Sauce	Spag Sauce	0.58835
Eggplant Relish	Spag Meat Sauce	0.46225	Spag Meat Sauce	Creole Sauce	0.56045
Eggplant Relish	Creole Sauce	0.38490	Spag Meat Sauce	Eggplant Relish	0.46225
Eggplant Relish	Salsa	0.35355	Spag Meat Sauce	Salsa	0.39223
Eggplant Relish	Enchilada Sauce	0.00000	Spag Meat Sauce	Enchilada Sauce	0.09806
Enchilada Sauce	Enchilada Sauce	1.00000	Spag Sauce	Spag Sauce	1.00000
Enchilada Sauce	Creole Sauce	0.20412	Spag Sauce	Eggplant Relish	0.58926
Enchilada Sauce	Salsa	0.12500	Spag Sauce	Spag Meat Sauce	0.58835
Enchilada Sauce	Spag Sauce	0.12500	Spag Sauce	Creole Sauce	0.51031
Enchilada Sauce	Spag Meat Sauce	0.09806	Spag Sauce	Salsa	0.50000
Enchilada Sauce	Eggplant Relish	0.00000	Spag Sauce	Enchilada Sauce	0.12500

The following statements produce a heat map visualization of the similarity between pairs of recipes:

```

proc sort data=mylib.SimilarityRecipesOut out=SimilarityRecipesSorted;
  by sink source;
run;
proc sgplot data=SimilarityRecipesSorted noautolegend;
  title "Cosine Similarity of Recipes";
  heatmap y=source x=sink / weight=cosine
    colormodel=(white green) outline x2axis;
  text y=source x=sink text=cosine / textAttrs=(size=10pt);
  xaxis display=none;
  x2axis display=(nolabel);
  yaxis display=(nolabel) reverse;
run;

```

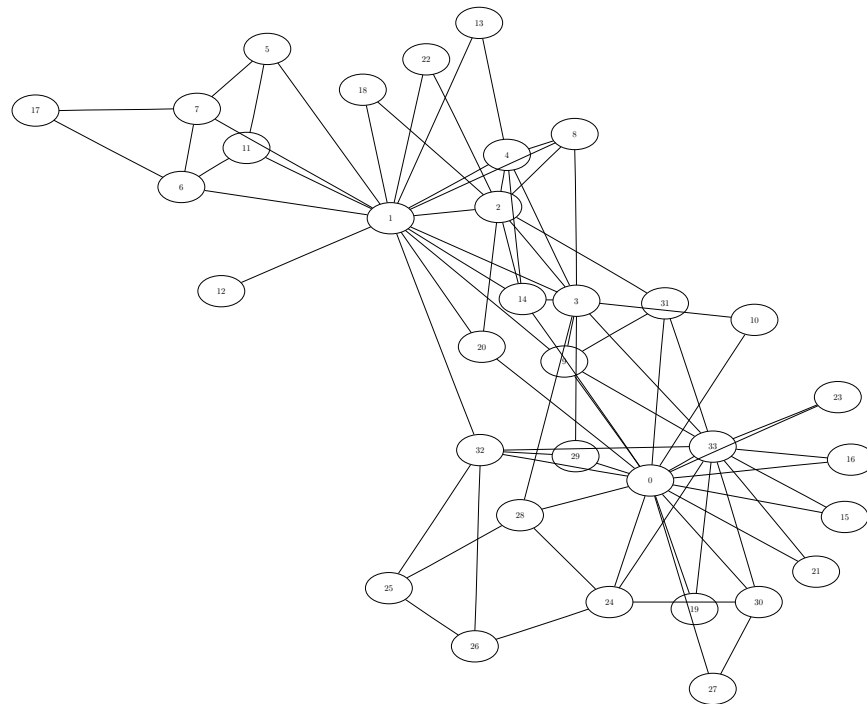
Output 2.18.2 Cosine Node Similarity between Recipe Pairs

Cosine Similarity of Recipes						
	Creole Sauce	Eggplant Relish	Enchilada Sauce	Salsa	Spag Meat Sauce	Spag Sauce
Creole Sauce	1	0.3849	0.2041	0.3062	0.5604	0.5103
Eggplant Relish	0.3849	1	0	0.3536	0.4623	0.5893
Enchilada Sauce	0.2041	0	1	0.125	0.0981	0.125
Salsa	0.3062	0.3536	0.125	1	0.3922	0.5
Spag Meat Sauce	0.5604	0.4623	0.0981	0.3922	1	0.5883
Spag Sauce	0.5103	0.5893	0.125	0.5	0.5883	1

From the heat map shown in [Output 2.18.2](#), you can conclude that the recipes most similar to the Creole sauce recipe by ingredient cosine similarity are the spaghetti meat sauce and spaghetti sauce recipes. Conversely, the enchilada sauce and eggplant relish recipes have zero cosine similarity because they have no ingredients in common.

Example 2.19: Node Embeddings for Zachary's Karate Club Data

This example uses Zachary's Karate Club data (Zachary 1977), which describes social network friendships among 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 node embedding algorithms. It contains 34 nodes and 78 links. The graph is shown in [Figure 2.238](#).

Figure 2.238 Zachary's Karate Club Graph

You can represent the graph by using the following links data table, `mylib.LinkSetIn`:

```
data mylib.LinkSetIn;
  input from to @@;
  datalines;
  0 9 0 10 0 14 0 15 0 16 0 19 0 20 0 21
  0 23 0 24 0 27 0 28 0 29 0 30 0 31 0 32
  0 33 2 1 3 1 3 2 4 1 4 2 4 3 5 1
  6 1 7 1 7 5 7 6 8 1 8 2 8 3 8 4
  9 1 9 3 10 3 11 1 11 5 11 6 12 1 13 1
  13 4 14 1 14 2 14 3 14 4 17 6 17 7 18 1
  18 2 20 1 20 2 22 1 22 2 26 24 26 25 28 3
  28 24 28 25 29 3 30 24 30 27 31 2 31 9 32 1
  32 25 32 26 32 29 33 3 33 9 33 15 33 16 33 19
  33 21 33 23 33 24 33 30 33 31 33 32
  ;
```

The following statements use the `EMBED=TRUE` option to generate node embeddings for the Zachary's Karate Club data:

```
proc network
  links      = mylib.LinkSetIn
  outNodes   = mylib.OutNodesEmb;
  nodeSimilarity
    jaccard      = false
    embed        = true
    proximityOrder = first
    nSamples     = 10000000;
run;
```

The output data table `mylib.OutNodesEmb` contains the embedding vectors for each node. The first three components of the first five node embedding vectors are shown in [Output 2.19.1](#).

Output 2.19.1 Embedding Vectors Output

node	vec_1	vec_2	vec_3
0	-0.07265	-0.001936	-0.01630
1	0.03212	-0.075448	-0.05439
2	0.12318	-0.038300	-0.08064
3	0.02962	-0.071463	0.08465
4	0.04965	0.010844	0.10781

For ease of visualization, it is common practice to collapse the multidimensional embeddings down to two dimensions. You can achieve this by using the TSNE procedure.

The following PROC TSNE statements produce a two-dimensional representation of the node embeddings:

```
proc tsne
  seed      = 1
  data      = mylib.OutNodesEmb
  nDimensions = 2
  perplexity = 15
  maxIters  = 8000;
  input     = vec_;;
  output
    out      = mylib.OutNodesTsne
    copyvars = (node);
run;
```

For validation, you can run community detection on the same graph.

The following PROC NETWORK statements produce the community labels that correspond to the karate club data:

```
proc network
  links    = mylib.LinkSetIn
  nodes    = mylib.OutNodesTsne
  outNodes = mylib.OutNodesComm;
  nodesVar
    vars = (_DIM_1_ _DIM_2_);
  community;
run;
```

Finally, you can plot the two-dimensional embeddings representation, colored by community label, by using the following PROC TEMPLATE and PROC SGRENDER statements. These statements produce the plot shown in [Output 2.19.2](#).

```
proc template;
  define statgraph vectorPlot;
  begingraph;
    entrytitle "TSNE Embeddings Colored by Community: Vector";
    layout overlay /
      yaxisopts=(labelFitPolicy=Split)

```

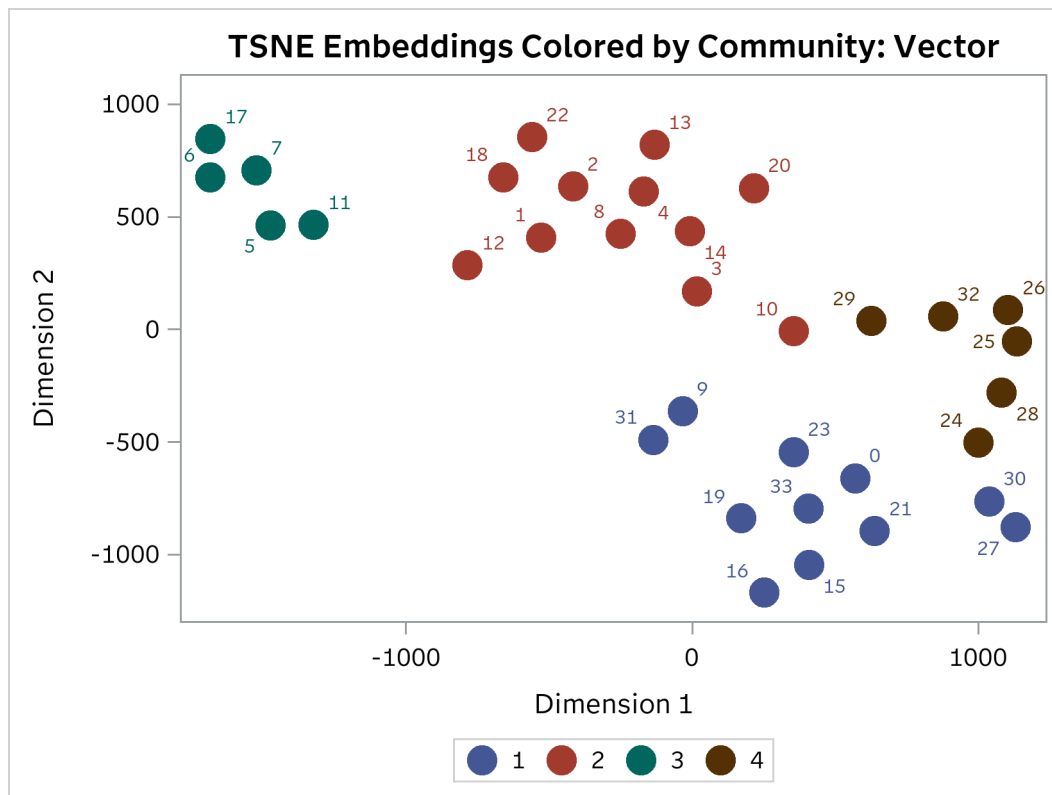
```

y2axisopts=(labelFitPolicy=Split);
ScatterPlot X=_DIM_1_ Y=_DIM_2_ /
    markerattrs=(symbol=circlefilled size=15)
    datalabel=node
    group=community_1
    name="scatter";
discretelegend "scatter";
endlayout;
endgraph;
end;
run;

proc sgrender data=mylib.OutNodesComm template=vectorPlot; /* use GTL template */
run;

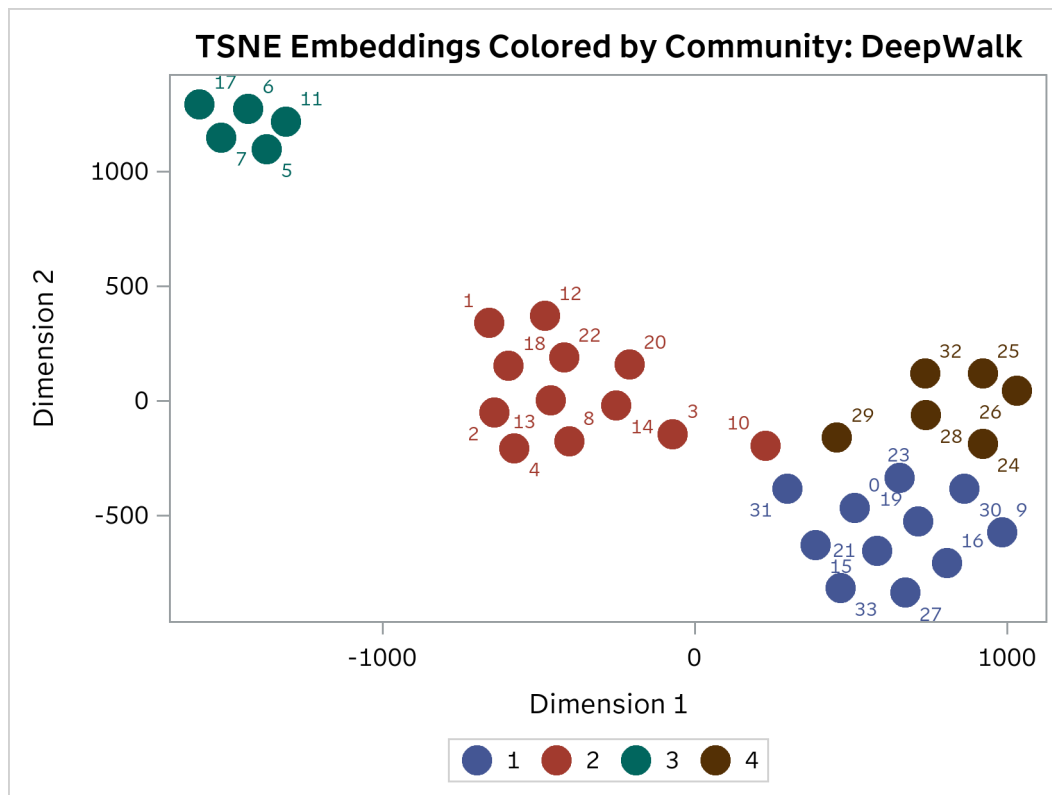
```

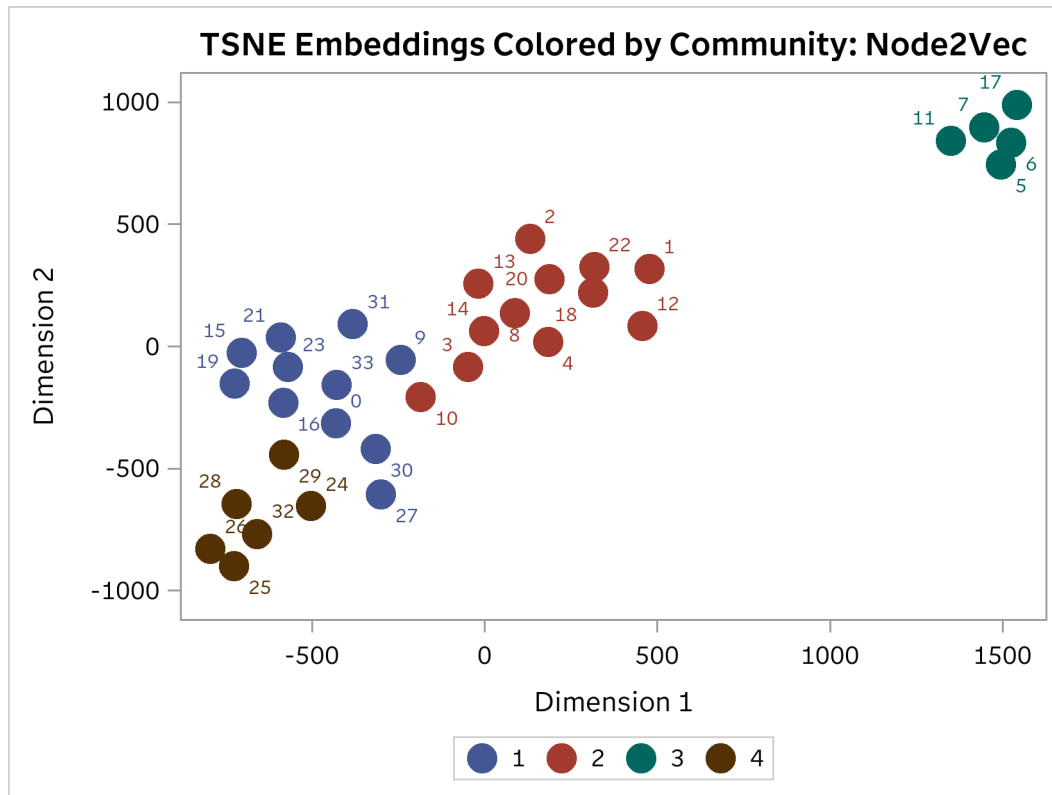
Output 2.19.2 TSNE Representation of Vector Node Embeddings



In the plot shown in [Output 2.19.2](#), note that the nodes in a particular community (color) appear to be clustered together in the embedded vector space.

The embeddings that are learned by using vector node similarity are functionally similar to the alternative techniques DeepWalk (Perozzi, Al-Rfou, and Skiena 2014; Perozzi 2014) and Node2Vec (Grover and Leskovec 2016; Grover 2016). Although these alternatives are not currently supported in PROC NETWORK, the results are provided here for comparison. The results that are obtained by DeepWalk for the same graph are shown in [Output 2.19.3](#), and the results that are obtained by Node2Vec for the same graph are shown in [Output 2.19.4](#).

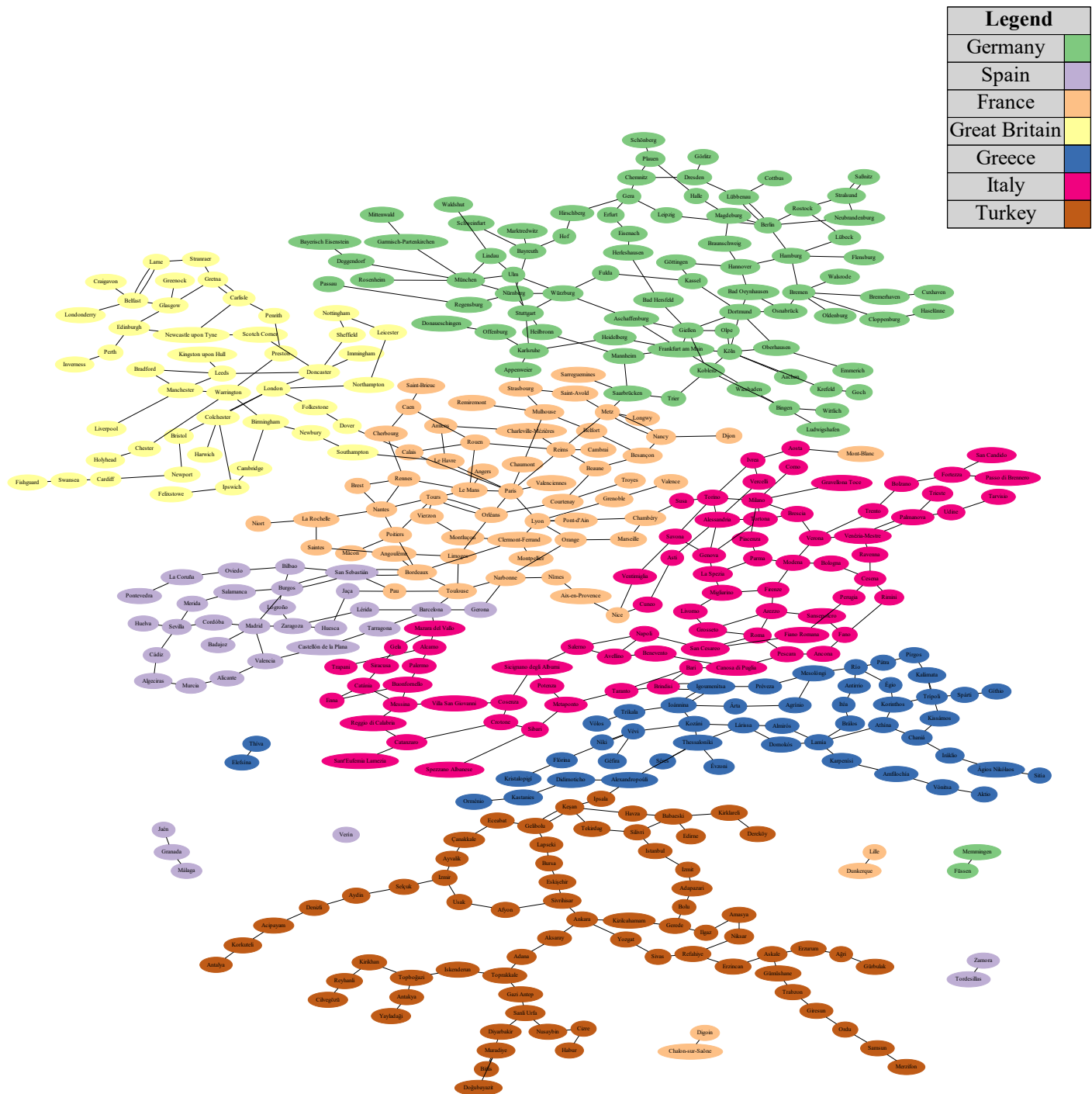
Output 2.19.3 TSNE Representation of DeepWalk Node Embeddings

Output 2.19.4 TSNE Representation of Node2Vec Node Embeddings

Example 2.20: Node Embeddings for European Roads Data

This example uses a European roads data set that is maintained by the Neo4j graph embeddings tutorial (Needham 2020). The graph includes 1,250 roadway connections among 894 cities and towns. You can compute node embeddings for these places by using vector node similarity. To simplify visualization, you can limit the data set to the seven most populous European countries: Spain, Great Britain, France, Turkey, Italy, Germany, and Greece. The graph is shown in [Figure 2.239](#).

Figure 2.239 European Roads Graph



You can represent the graph by using the following links data table, `mylib.LinkSetIn`. The following statements create the data table `mylib.Roads` from a local copy of the raw European roads data:

```

filename in 'roads.csv';
data mylib.Roads;
  infile in missover firstobs = 2 DSD;
  length from to $40;
  input
    road_number $
    from_country $
    from $
    to_country $
    to $
    distance
    watercrossing $;
run;

```

The following statements construct the nodes and links data tables, mylib.Nodes and mylib.Links, from the raw roads data:

```

%let country_subset = ("E", "GB", "F", "TR", "I", "D", "GR");
data mylib.nodes;
  set mylib.Roads(
    rename=(from=node from_country=country)
  )
  mylib.Roads(
    rename=(to=node to_country=country)
  );
  by country node;
  if FIRST.node
    and country in &country_subset;
  if      country EQ 'D' then color=1;
  else if country EQ 'E' then color=2;
  else if country EQ 'F' then color=3;
  else if country EQ 'GB' then color=4;
  else if country EQ 'GR' then color=5;
  else if country EQ 'I' then color=6;
  else if country EQ 'TR' then color=7;
  keep node country color;
run;
data mylib.Links;
  set mylib.Roads;
  where from_country in &country_subset and to_country in &country_subset;
run;

```

The following statements use the EMBED=TRUE option to generate node embeddings for the European roads data:

```

proc network
  links      = mylib.Links
  nodes      = mylib.Nodes
  outNodes   = mylib.OutNodesEmb;
  nodesVar
    vars     = (country color);
  nodeSimilarity
    jaccard   = false
    embed     = true

```

```

nDimensions      = 300
proximityOrder   = first
nSamples         = 10000000;
run;

```

The output data table `mylib.OutNodesEmb` contains the embedding vectors for each node. The first three components of five sample node embedding vectors are shown in [Output 2.20.1](#).

Output 2.20.1 Embedding Vectors Output

node	vec_1	vec_2	vec_3
Alessandria	-0.062927	-0.061893	-0.084126
Belfast	0.007215	0.084008	-0.089538
Hamburg	-0.015572	-0.061447	0.063758
Milano	-0.023096	-0.021641	0.077764
Rennes	-0.004545	0.095322	0.097337

For ease of visualization, it is common practice to collapse the multidimensional embeddings down to two dimensions. You can achieve this by using the TSNE procedure.

The following PROC TSNE statements produce a two-dimensional representation of the node embeddings:

```

proc tsne
  seed          = 1
  data          = mylib.OutNodesEmb
  nDimensions   = 2
  perplexity    = 15
  maxIters      = 3000;
  input         vec_;
  output
    out         = mylib.OutNodesTsne
    copyvars    = (node country color);
run;

```

For validation, you can visualize the node embeddings. You can plot the two-dimensional embeddings representation, colored by country, by using the following PROC TEMPLATE and PROC SGRENDER statements:

```

proc template;
  define statgraph vectorPlot;
    begingraph;
      entrytitle "TSNE Embeddings Colored by Country: Vector";
      legendItem type=marker name="d" /
        markerattrs=(color=CX7FC97F symbol=circlefilled)
        label="Germany" ;
      legendItem type=marker name="e" /
        markerattrs=(color=CXBEAED4 symbol=circlefilled)
        label="Spain" ;
      legendItem type=marker name="f" /
        markerattrs=(color=CXFD086 symbol=circlefilled)
        label="France" ;
      legendItem type=marker name="gb" /
        markerattrs=(color=CXFFFF99 symbol=circlefilled)
        label="Great Britain" ;
    endgraph;
  end;
run;

```

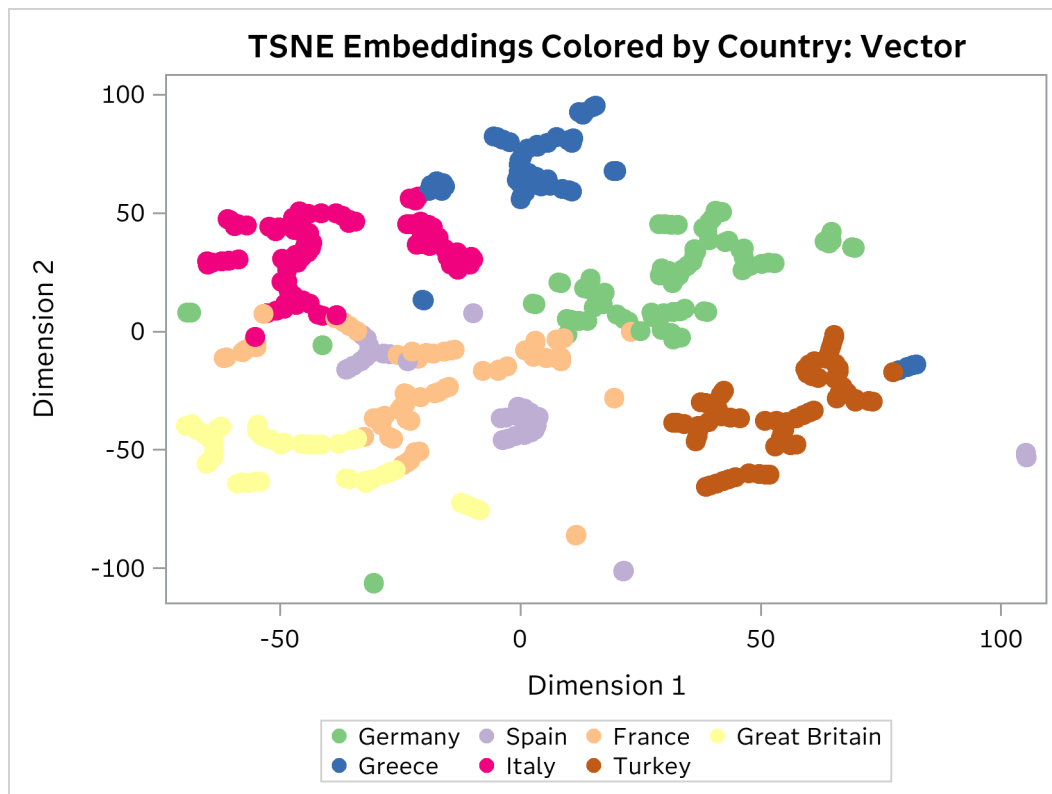


```

legendItem type=marker name="gr" /
markerattrs=(color=CX386CB0 symbol=circlefilled)
label="Greece" ;
legendItem type=marker name="i" /
markerattrs=(color=CXF0027F symbol=circlefilled)
label="Italy" ;
legendItem type=marker name="tr" /
markerattrs=(color=CXBF5B17 symbol=circlefilled)
label="Turkey" ;
layout overlay /
  yaxisopts=(labelFitPolicy=Split)
  y2axisopts=(labelFitPolicy=Split);
  ScatterPlot X=_DIM_1_ Y=_DIM_2_ /
    markerattrs=(symbol=circlefilled size=10)
    colormodel=(CX7FC97F CXBEAED4 CXFDC086 CXFFFF99 CX386CB0 CXF0027F CXBF5B17)
    colorresponse=color
    name="scatter";
    discretelegend "d" "e" "f" "gb" "gr" "i" "tr";
  endlayout;
endgraph;
end;
run;

proc sgrender data=mylib.OutNodesTsne template=vectorPlot; /* use GTL template */
run;

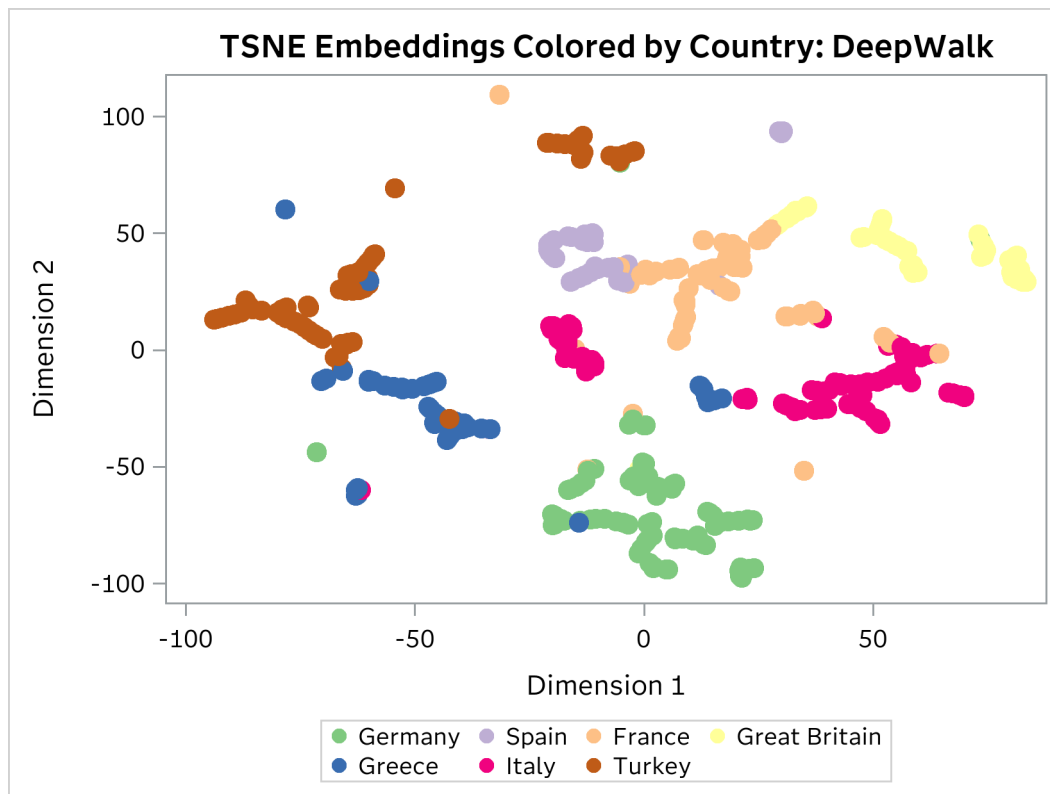
```

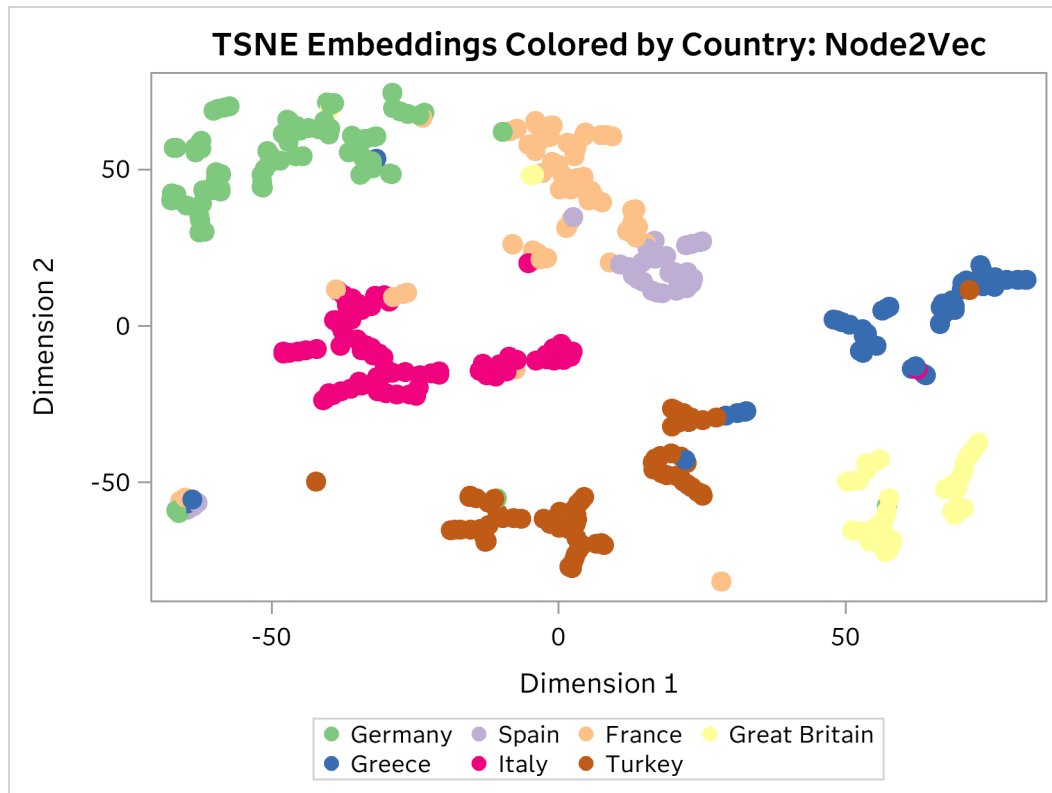
Output 2.20.2 TSNE Representation of Vector Node Embeddings

In the plot shown in [Output 2.20.2](#), note that the nodes representing places within the same country (color) appear to be clustered together in the embedded vector space.

The embeddings that are learned by using vector node similarity are functionally similar to the alternative techniques DeepWalk (Perozzi, Al-Rfou, and Skiena 2014; Perozzi 2014) and Node2Vec (Grover and Leskovec 2016; Grover 2016). Although these alternatives are not currently supported in PROC NETWORK, the results are provided here for comparison. The results that are obtained by DeepWalk for the same graph are shown in [Output 2.20.3](#), and the results that are obtained by Node2Vec for the same graph are shown in [Output 2.20.4](#).

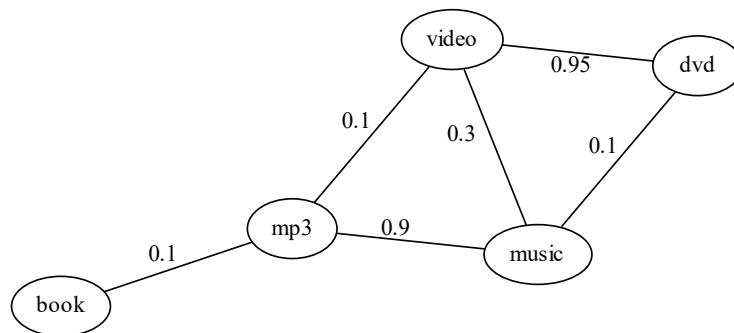
Output 2.20.3 TSNE Representation of DeepWalk Node Embeddings



Output 2.20.4 TSNE Representation of Node2Vec Node Embeddings

Example 2.21: Learning Word Embeddings from Synonym Pairs

This example illustrates how to learn word embeddings by using node similarity. Consider an undirected graph where the nodes are a set of five terms and the links are synonym relationships between pairs of terms. The assigned weight on each link represents the strength of the synonym relationship. You can compute word embeddings for these terms by using vector node similarity. The graph is shown in [Figure 2.240](#).

Figure 2.240 Synonym Pairs Graph

You can represent the graph by using the following links data table, `mylib.LinkSetIn`:

```
data mylib.SynonymLinks;
    input from $ to $ weight;
    datalines;
book mp3 0.1
dvd music 0.1
dvd video 0.95
mp3 music 0.9
mp3 video 0.1
music video 0.3
;
```

In practice, this input graph would typically be based on co-occurrence frequency in a large corpus and could contain millions of nodes. Word embeddings provide a compressed representation of the information in the original graph and are often inputs to low-dimensional visualizations and machine learning pipelines.

The following statements use the `VECTOR=TRUE` option to generate word embeddings for the terms in the data set:

```
%let nDim=5;
%let nSamples=100000;
%let convergenceThreshold=0.001;
proc network
    links                = mylib.SynonymLinks
    outNodes              = mylib.WordEmbeddings;
    nodeSimilarity
        vector           = true
        jaccard           = false
        proximityOrder    = first
        nDimensions       = &nDim
        nSamples          = &nSamples
        convergenceThreshold = &convergenceThreshold
        outSimilarity     = mylib.OutSimilarity;
run;
```

The output data table `mylib.WordEmbeddings` contains the five-element embedding vector for each node. The word embedding vector for each of the five terms is shown in [Output 2.21.1](#).

Output 2.21.1 Embedding Vectors Output

node	vec_1	vec_2	vec_3	vec_4	vec_5
book	-0.21486	-0.42298	0.14155	-0.84803	0.18904
dvd	-0.43874	-0.46665	-0.48534	0.13360	-0.57995
mp3	-0.34548	-0.11227	-0.48567	0.11979	0.78601
music	-0.08957	-0.04459	-0.78993	0.04740	0.60312
video	-0.35923	-0.31128	-0.71243	0.01254	-0.51608

The output data table `mylib.OutSimilarity` contains the pairwise vector similarity between terms, which excludes redundant node pairs where source comes lexicographically after sink. For display purposes, it is helpful to create a symmetric results table that also includes pairs where source comes lexicographically after sink. You can create the table `mylib.OutSimilaritySymmetric`, shown in [Output 2.21.2](#), by using the following

DATA step:

```
data mylib.OutSimilaritySymmetric;
  set mylib.OutSimilarity;
  output;
  if source < sink then do;
    temp = source;
    source = sink;
    sink = temp;
    output;
  end;
  drop temp;
run;
```

Output 2.21.2 Vector Similarity Output

source=book

sink	link	vector
book	0	1.00000
mp3	1	0.54999
music	0	0.50006
dvd	0	0.50001
video	0	0.49991

source=dvd

sink	link	vector
dvd	0	1.00000
video	1	0.97481
music	1	0.55002
book	0	0.50001
mp3	0	0.49992

source=mp3

sink	link	vector
mp3	0	1.00000
music	1	0.94967
video	1	0.55046
book	1	0.54999
dvd	0	0.49992

source=music

sink	link	vector
music	0	1.00000
mp3	1	0.94967
video	1	0.64908
dvd	1	0.55002
book	0	0.50006

Output 2.21.2 *continued*

source=video		
sink	link	vector
video	0	1.00000
dvd	1	0.97481
music	1	0.64908
mp3	1	0.55046
book	0	0.49991

To validate the quality of the word embeddings, you can use the scalar product between embedding vector pairs to reconstruct the original graph. Note that although vector scalar products range from -1 to 1 , the vector similarity variable is normalized to lie in the range 0 to 1 . Unassociated words, therefore, are assigned a vector similarity score of 0.5 .

To reconstruct an approximation of the original graph, you can scale the scores back to the range -1 to 1 and discard any pairs whose weight is less than 0.001 . You can create the table `mylib.ReconstructedLinks` by using the following DATA step:

```
data mylib.ReconstructedLinks;
  set mylib.OutSimilarity(where=(source < sink));
  weight = vector*2-1;
  keep source sink weight;
  if weight >= 0.001;
  rename source=from sink=to;
run;
```

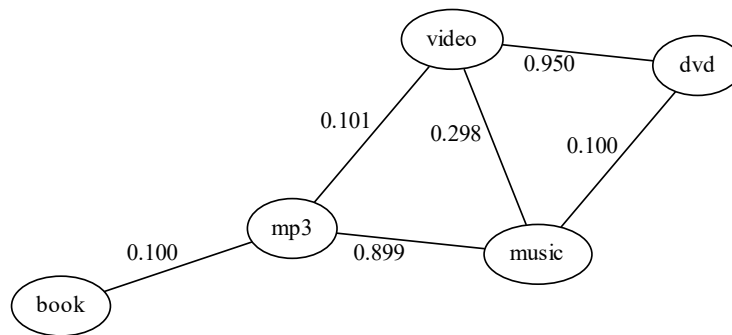
For ease of comparison, you can join the original link weights to the `mylib.ReconstructedLinks` table, shown in [Output 2.21.3](#). The following statements use PROC FEDSQL to perform this join:

```
proc fedsql sessref=mysess;
  create table ReconstructedLinks {options replace=true} as
  select a.*, b.weight as "originalWeight"
  from ReconstructedLinks as a
  join SynonymLinks as b
  on a.from = b.from and a.to = b.to;
quit;
```

Output 2.21.3 Links Reconstructed from Vector Similarity

from	to	weight	originalWeight
book	mp3	0.09997	0.10
dvd	music	0.10004	0.10
dvd	video	0.94961	0.95
mp3	music	0.89933	0.90
mp3	video	0.10092	0.10
music	video	0.29816	0.30

The reconstructed graph is shown in [Figure 2.241](#).

Figure 2.241 Reconstructed Synonym Pairs Graph

Example 2.22: Using Projection to Recommend Recipe Ingredient Pairings

The task of recipe recommendation that is demonstrated in “[Example 2.18: A Node Similarity Recipe Recommendation Engine](#)” on page 341 can also be achieved by using projection. This example solves a similar recommendation problem that considers ingredient pairings instead of recipe pairings. By using the PROJECTION statement, you can find pairs of ingredients that occur together in one or more recipes. Consider again the data table `mylib.LinkSetRecipesIn`, which is used in [Example 2.18](#). The inclusion of ingredients in recipes is organized into a bipartite network G , which is shown in [Figure 2.237](#).

In this example, the numbers of common neighbors (recipe nodes) between ingredient node pairs are of interest. The PROJECTION statement requires a nodes data table with a column that indicates which nodes are recipes and which nodes are ingredients. You can use the following statements to generate the nodes data table `mylib.NodeSetRecipesIn` (which has a single variable called `node`):

```
proc network
  links      = mylib.LinkSetRecipesIn
  outNodes   = mylib.NodeSetRecipesIn;
run;
```

Then, you can use the following DATA step to define a partition variable:

```
data mylib.NodeSetRecipesIn;
  set mylib.NodeSetRecipesIn;
  if node in (
    'Spag Sauce'
    'Spag Meat Sauce'
    'Eggplant Relish'
    'Creole Sauce'
    'Salsa'
    'Enchilada Sauce'
  ) then partition = 0;
  else partition = 1;
run;
```

The partition variable has a value of 0 for recipe nodes and a value of 1 for ingredient nodes. Therefore, PROC NETWORK computes the network projection onto ingredient nodes through recipe nodes. The following statements produce the output data table `mylib.ProjectionLinkSetOut`, which contains links between ingredient

pairs that appear in at least one common recipe. In addition, the output data table `mylib.ProjectionListOut` contains the lists of recipes that are common to each pair of ingredients.

```
proc network
  nodes = mylib.NodeSetRecipesIn
  links = mylib.LinkSetRecipesIn;
  projection
    partition          = partition
    commonNeighbors    = true
    outProjectionLinks = mylib.ProjectionLinkSetOut
    outNeighborsList   = mylib.ProjectionListOut;
run;
```

The output is displayed in [Output 2.22.1](#); it contains the ingredient pairings that occur in three or more recipes. For example, garlic and olive oil have three common neighbors, which means they occur together in three recipes.

Output 2.22.1 Top Ingredient Pairings

from	to	commonNeighbors
Onion	Salt	5
Garlic	Onion	4
Garlic	Salt	4
OliveOil	Onion	4
OliveOil	Salt	4
OliveOil	TomatoPaste	4
Onion	Tomato	4
Onion	TomatoPaste	4
Salt	Tomato	4
Salt	TomatoPaste	4
Garlic	OliveOil	3
Garlic	Tomato	3
Garlic	TomatoPaste	3
OliveOil	Tomato	3
Tomato	TomatoPaste	3

You can also use the following statements to display the particular recipes that contain both garlic and olive oil:

```
proc print data=mylib.ProjectionListOut
  (where=(from = 'Garlic' and to = 'OliveOil'))
  noobs;
run;
```

The output is shown in [Output 2.22.2](#); it indicates that eggplant relish, spaghetti sauce, and spaghetti meat sauce are the three recipes that contain both garlic and olive oil.

Output 2.22.2 Top Ingredient Pairings

from	to	neighbor_id	neighbor
Garlic	OliveOil	1	Eggplant Relish
Garlic	OliveOil	2	Spag Meat Sauce
Garlic	OliveOil	3	Spag Sauce

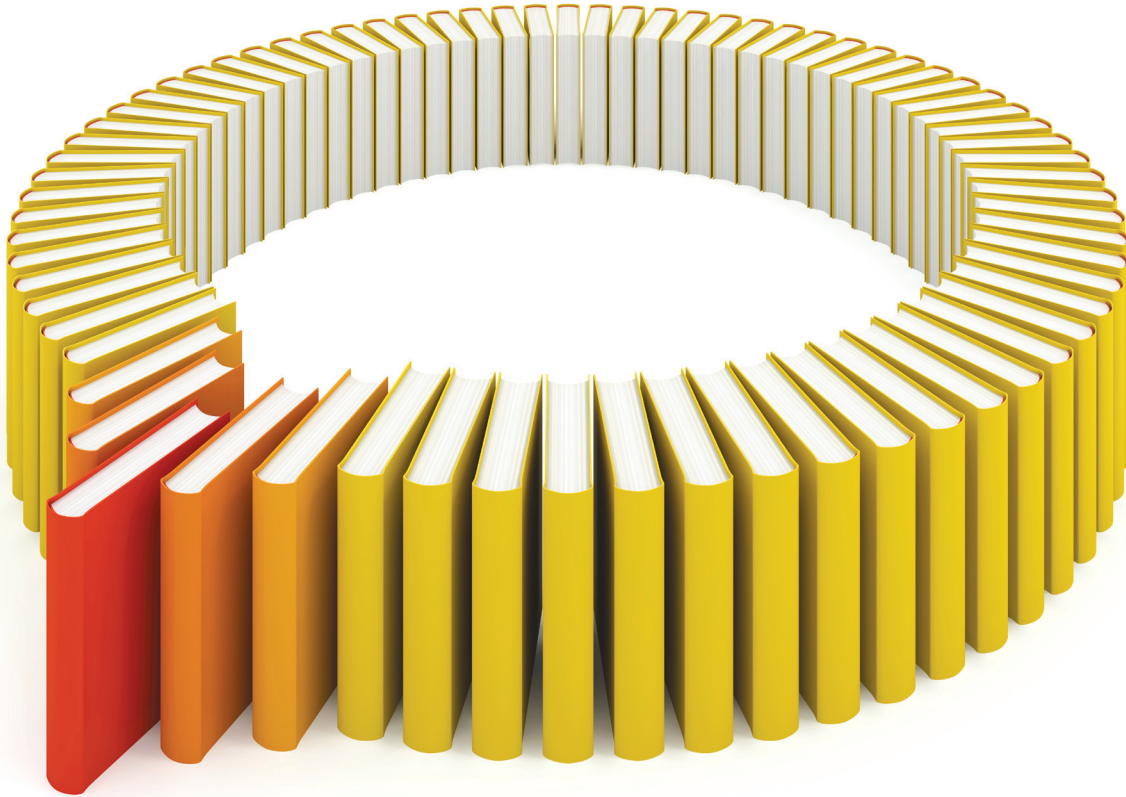
References

- Adamic, L. A., and Adar, E. (2003). “Friends and Neighbors on the Web.” *Social Networks* 25:211–230.
- Aggarwal, C. C., and Wang, H., eds. (2010). *Managing and Mining Graph Data*. Boston: Springer.
- Ahuja, R. K., Magnanti, T. L., and Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice-Hall.
- AlGhamdi, Z., Jamour, F., Skiadopoulos, S., and Kalnis, P. (2017). “A Benchmark for Betweenness Centrality Approximation Algorithms on Large Graphs.” In *SSDBM '17: Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, 1–12. New York: ACM. <https://doi.org/10.1145/3085504.3085510>.
- Batagelj, V., and Zaversnik, M. (2003). “An O(m) Algorithm for Cores Decomposition of Networks.” *Computing Research Repository* cs.DS/0310049.
- Berlusconi, G., Calderoni, F., Parolini, N., Verani, M., and Piccardi, C. (2016). “Link Prediction in Criminal Networks: A Tool for Criminal Intelligence Analysis.” *PLoS ONE* 11:1–21. <https://doi.org/10.1371/journal.pone.0154244>.
- 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.
- CNN (2012). “Record Kidney Transplant Chain.” CNN Health. Video. Accessed April 2, 2019. <https://www.cnn.com/videos/health/2012/02/20/dnt-kidney-transplant-chain.wls>.
- Conte, D., Foggia, P., Sansone, C., and Vento, M. (2004). “Thirty Years of Graph Matching in Pattern Recognition.” *International Journal of Pattern Recognition and Artificial Intelligence* 18:265–298.
- 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>.
- Dice, L. R. (1945). “Measures of the Amount of Ecologic Association between Species.” *Ecology* 26:297–302.
- Fowler, J. H., and Joen, S. (2008). “The Authority of Supreme Court Precedent.” *Social Networks* 30:16–30. <https://doi.org/10.1016/j.socnet.2007.05.001>.
- Gallagher, B. (2006). “Matching Structure and Semantics: A Survey on Graph-Based Pattern Matching.” In *AAAI Fall Symposium: Capturing and Using Patterns for Evidence Detection*. Menlo Park, CA: Association for the Advancement of Artificial Intelligence.

- Google (2011). “Google Maps.” Accessed March 16, 2011. <http://maps.google.com>.
- Grover, A. (2016). “Node2Vec Reference Implementation.” Accessed January 2021. <https://github.com/aditya-grover/node2vec>.
- Grover, A., and Leskovec, J. (2016). “node2vec: Scalable Feature Learning for Networks.” In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 855–864. New York: Association for Computing Machinery. <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5108654/>.
- 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. <https://firstmonday.org/ojs/index.php/fm/article/view/941/863>.
- Lamensch, M., and Ceci, E. (2018). *VAT Fraud: Economic Impact, Challenges and Policy Issues*. Technical Report PE 626.076, European Parliament Policy Department for Economic, Scientific and Quality of Life Policies. <http://www.europarl.europa.eu/cmsdata/156408/VAT%20Fraud%20Study%20publication.pdf>.
- 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.
- Langville, A. N., and Meyer, C. D. (2006). *Google’s PageRank and Beyond: The Science of Search Engine Rankings*. Princeton, NJ: Princeton University Press.
- 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.
- Needham, M. (2020). “Neo4j Graph Embeddings.” The data are available at <https://github.com/neo4j-examples/graph-embeddings>.

- Newman, M. E. J. (2002). “Assortative Mixing in Networks.” *Physical Review Letters* 89:208701 (1–4).
- Newman, M. E. J. (2003). “Mixing Patterns in Networks.” *Physical Review E* 67:026126 (1–13).
- Newman, M. E. J. (2010). *Networks: An Introduction*. Oxford: Oxford University Press.
- Perozzi, B. (2014). “DeepWalk, Version 1.0.2.” Accessed January 2021. <https://github.com/phanein/deepwalk>.
- Perozzi, B., Al-Rfou, R., and Skiena, S. (2014). “Deepwalk: Online Learning of Social Representations.” In *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 701–710. New York: Association for Computing Machinery. <https://doi.org/10.1145/2623330.2623732>.
- 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. <https://infoscience.epfl.ch/record/200525/files/%5BEN%5DASNA09.pdf>.
- Ronhovde, P., and Nussinov, Z. (2010). “Local Resolution-Limit-Free Potts Model for Community Detection.” *Physical Review E* 81:46114–46129.
- Saltz, M. W. (2013). “A Fast Algorithm for Subgraph Pattern Matching on Large Labeled Graphs.” Master’s thesis, University of Georgia.
- Shasha, D., Wang, J. T. L., and Giugno, R. (2002). “Algorithmics and Applications of Tree and Graph Searching.” In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 39–52. New York: ACM.
- 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.
- Sørensen, T. (1948). “A Method of Establishing Groups of Equal Amplitude in Plant Sociology Based on Similarity of Species Content and Its Application to Analyses of the Vegetation on Danish Commons.” *Biologiske Skrifter* 5:1–34.
- Sutton, M., Ben-Nun, T., and Barak, A. (2018). “Optimizing Parallel Graph Connectivity Computation via Subgraph Sampling.” In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 12–21. New York: IEEE. <https://doi.org/10.1109/IPDPS.2018.00012>.
- Tang, J., Qu, M., Wang, M., Zhang, M., Yan, J., and Mei, Q. (2015). “LINE: Large-Scale Information Network Embedding.” In *WWW ’15: Proceedings of the 24th International Conference on World Wide Web*, 1067–1077. Geneva: International World Wide Web Conferences Steering Committee.
- 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>.
- Yen, J. Y. (1971). “Finding the K Shortest Loopless Paths in a Network.” *Management Science* 17:712–716.
- Zachary, W. W. (1977). “An Information Flow Model for Conflict and Fission in Small Groups.” *Journal of Anthropological Research* 33:452–473.



Gain Greater Insight into Your SAS[®] Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

 support.sas.com/bookstore
for additional books and resources.


THE POWER TO KNOW.®

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies. © 2013 SAS Institute Inc. All rights reserved. S107969US.0613