

Introducing Pattern Matching for Graph Queries in SAS[®] Viya[®] 3.4

Matthew Galati, Steve Harenberg, and Rob Pratt, SAS Institute Inc.

ABSTRACT

SAS[®] Visual Data Mining and Machine Learning 8.3 in SAS[®] Viya[®] 3.4 includes a new `patternMatch` action, which you can use to execute graph queries that search for copies of a query graph within a larger graph, with the option of respecting node or link attributes (or both). This feature is also available via the `PATTERNMATCH` statement in the `NETWORK` procedure. This paper presents examples of pattern matching in social network and anti-money laundering applications. It also provides a functional comparison to Neo4j's query language, Cypher, and computational comparisons to both iGraph and Neo4j.

INTRODUCTION

SAS Visual Data Mining and Machine Learning 8.3 in SAS Viya 3.4 includes the `network` action set and corresponding `NETWORK` procedure, which contain 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 other customers—such as friends and acquaintances that they regularly interact with—might contribute to the propensity of the 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` action set makes no assumptions about the context or application from which the network arises. It provides a number of network analysis algorithms that take an abstract graph or network as input, help explain network structure, and compute important network measures. Depending on the application, this type of network analysis can stand on its own and provide independent value, or it can support machine learning models—for example, by providing additional features that are derived from network measures such as node centrality.

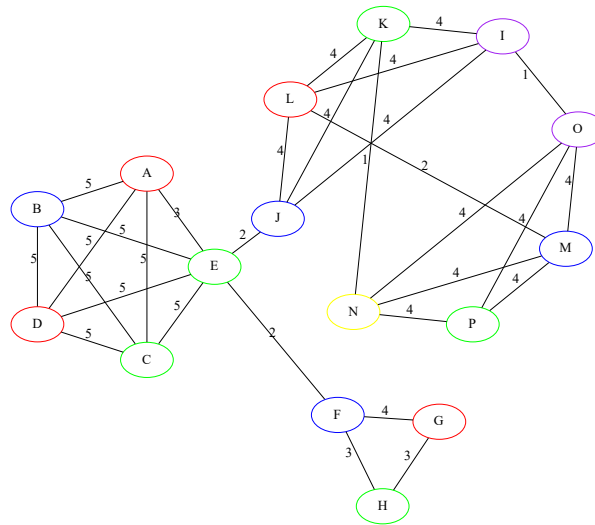
This paper uses the `NETWORK` procedure in the presentation of examples. For more information about the `NETWORK` procedure, see *SAS Visual Data Mining and Machine Learning: The NETWORK Procedure*. For more information about the `network` action set, see *SAS Visual Data Mining and Machine Learning: Programming Guide*. The general interface for using the `network` action set is the same for all languages that SAS Viya supports: CASL, Python, Java, Lua, and R. For more information about how SAS Viya supports these languages, see *An Introduction to SAS Viya Programming*. For the remainder of this paper, the authors refer to the network analytics package as *Network*, independent of the chosen interface language.

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.

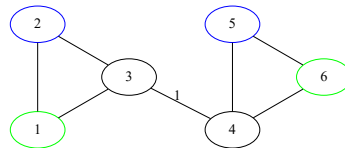
For example, consider the undirected graph G , shown in Figure 1. G has one link attribute (weight) and one node attribute (color).

Figure 1 Undirected Graph G



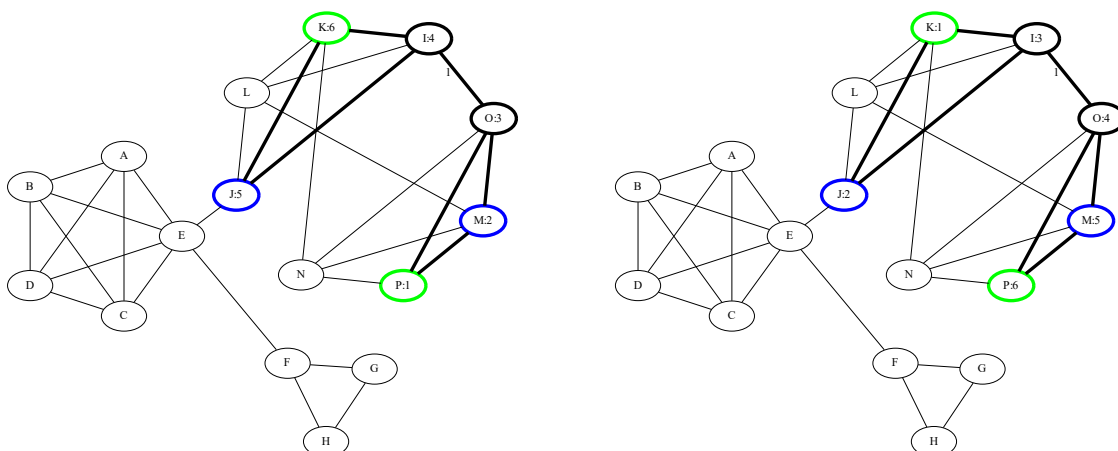
Now consider the query graph Q , shown in Figure 2.

Figure 2 Query Graph Q



There are two isomorphic mappings of the query graph Q in the main graph G , as shown in Figure 3.

Figure 3 Subgraphs



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).

The focus of this paper is to introduce a new action, `patternMatch` in the `network` action set (and the corresponding `NETWORK` procedure), in SAS Visual Data Mining and Machine Learning 8.3 and later, which can be used to solve the problem of pattern matching.

PATTERNMATCH EXAMPLES

In PROC NETWORK, you can find pattern matches by using the PATTERNMATCH statement.

The specification of a graph in PROC NETWORK consists of a nodes data table (optional) and a links data table. The nodes data table contains a list of nodes with one column for the node label and any number of additional columns for each node attribute. The links data table contains a list of links (specified as a pair of node labels: a *from* node and a *to* node) and any number of additional columns for each link attribute. The links data table is specified using the LINKS= option in the PROC NETWORK statement. The nodes data table is specified using the NODES= option in the PROC NETWORK statement.

The query graph is specified using 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. If you use attributes in the query data tables, the PATTERNMATCH statement returns subgraphs that exactly match the specified values in addition to matching the defined graph structure (topology).

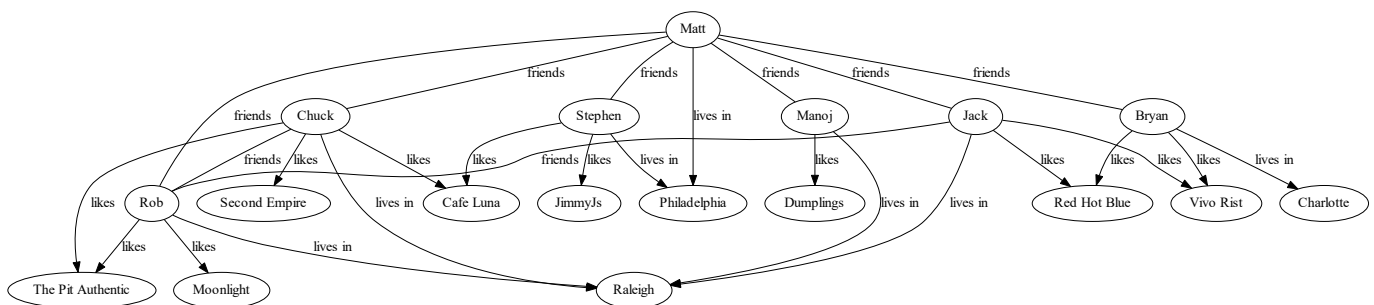
New in SAS Visual Data Mining and Machine Learning 8.4 is the ability to further customize the query using the SAS Function Compiler (FCMP), which provides greater flexibility than the use of query data tables. For a general overview of FCMP and examples of its use in creating user-defined functions, see the chapter “The FCMP Procedure” in *Base SAS Procedures Guide*. In PROC NETWORK, you can use FCMP to provide a set of functions, each of which, when associated with a specific pattern match query, defines an additional Boolean 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 you have in defining FCMP functions permits both exact and inexact attribute matching for individual nodes and links (by using the NODEFILTER= option and the LINKFILTER= option, respectively), and these functions can be further tailored to specific nodes and links in the query graph. In addition, the ability to use functions that apply to pairs of nodes and links (by using the NODEPAIRFILTER= option and the LINKPAIRFILTER= option, respectively) enables you to define conditions of more global scope than conditions on attributes of individual nodes and links.

The following two subsections consider two example applications of the PATTERNMATCH statement in PROC NETWORK.

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 4.

Figure 4 Social Network G



The following data provide a snapshot of the social connections between Matt and a few of his friends:

```

data mycas.NodesSocial;
  infile datalines dsd;
  length node $40. type $40. subtype $20.;
  input node $ type $ subtype $;
  label=node;
  datalines;
Matt,          Person,
Rob,           Person,
Chuck,        Person,
Stephen,      Person,
Manoj,        Person,
Bryan,        Person,
Jack,         Person,
Raleigh,     City,
Philadelphia, City,
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 mycas.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, .
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, .
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

```

```

Bryan, Red Hot Blue, likes, 9
Bryan, Vivo Rist, likes, 6
Rob, Moonlight, likes, 10
;

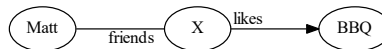
```

The nodes in the nodes data table **mycas.NodesSocial** represent people, cities, and restaurants. The node attribute **type** defines the node type. In the case of a restaurant, the node attribute **subtype** defines the type of restaurant.

The links in the links data table **mycas.LinksSocial** represent connections between the nodes. The type of connection is defined by the link attribute **connection**, 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.” A query graph that captures this pattern is shown in Figure 5.

Figure 5 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 mycas.NodesSocialQuery;
  infile datalines dsd;
  length node $40. label $40. type $40. subtype $20.;
  input node $ label $ type $ subtype $;
  datalines;
Matt, Matt, Person,
X,, Person,
BBQ,, Restaurant, BBQ
;
data mycas.LinksSocialQuery;
  infile datalines dsd;
  length from $40. to $40. connection $20.;
  input from $ to $ connection $;
  datalines;
Matt, X, friends
X, Matt, friends
X, BBQ, likes
;

```

The query 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          = mycas.NodesSocial
  links          = mycas.LinksSocial
  nodesQuery    = mycas.NodesSocialQuery
  linksQuery     = mycas.LinksSocialQuery;
  nodesVar
    vars        = (label type subtype);
  linksVar
    vars        = (connection);
  nodesQueryVar
    vars        = (label type subtype);
  linksQueryVar
    vars        = (connection);
  patternMatch
    outMatchNodes = mycas.OutMatchNodes
    outMatchLinks = mycas.OutMatchLinks;
run;

```

The progress of the procedure is shown in [Output 1](#).

Output 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 18.
NOTE: The number of links in the input graph is 35.
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 8 threads across 1 machines.
NOTE: The algorithm found 5 matches.
NOTE: Processing the pattern matching query used 0.00 (cpu: 0.00) seconds.
NOTE: The Cloud Analytic Services server processed the request in 0.007591
      seconds.
NOTE: The data set MYCAS.OUTMATCHNODES has 15 observations and 6 variables.
NOTE: The data set MYCAS.OUTMATCHLINKS has 15 observations and 4 variables.

```

[Output 2](#) displays the output data table **mycas.OutMatchNodes**, which shows the mappings from nodes in the query graph to nodes in the input graph for each match. For this query, five friends (X) match the specified criteria: Bryan, Chuck, Jack, Rob, and Stephen.

Output 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 3 displays the output data table **mycas.OutMatchLinks**, which shows the subgraphs for each match.

Output 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

Next, you can find “friends of Matt who like barbecue restaurants and live in Raleigh,” shown in [Figure 6](#), by using the data that are created by the following DATA steps and the same call to PROC NETWORK as before:

```

data mycas.NodesSocialQuery;
  infile datalines dsd;
  length node $40. label $40. type $40. subtype $20.;
  input node $ label $ type $ subtype $;
  datalines;
Matt,    Matt,    Person,
X,,      Person,
Raleigh, Raleigh, City,
BBQ,,    Restaurant, BBQ
;
data mycas.LinksSocialQuery;
  infile datalines dsd;
  length from $40. to $40. connection $20.;

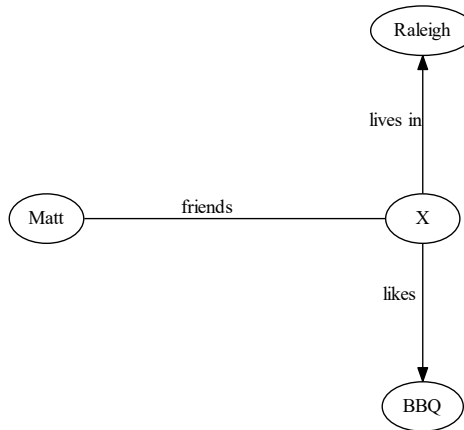
```

```

input from $ to $ connection $;
datalines;
Matt, X,      friends
X,  Matt,    friends
X,  Raleigh, lives in
X,  BBQ,     likes
;

```

Figure 6 Query Graph Q



Output 4 displays the output data table **mycas.OutMatchNodes**. For this query, three friends (X) match the specified criteria: Rob, Chuck, and Jack.

Output 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	Rob	Rob	Person	
2	BBQ	The Pit Authentic	The Pit Authentic	Restaurant	BBQ
2	Matt	Matt	Matt	Person	
2	Raleigh	Raleigh	Raleigh	City	
2	X	Chuck	Chuck	Person	
3	BBQ	Red Hot Blue	Red Hot Blue	Restaurant	BBQ
3	Matt	Matt	Matt	Person	
3	Raleigh	Raleigh	Raleigh	City	
3	X	Jack	Jack	Person	

Output 5 displays the output data table **mycas.OutMatchLinks**.

Output 5 Subgraphs for Friends Who Like Barbecue and Live in Raleigh

	match	from	to	connection
	1	Matt	Rob	friends
	1	Rob	Matt	friends
	1	Rob	Raleigh	lives in
	1	Rob	The Pit Authentic	likes
	2	Chuck	Matt	friends
	2	Chuck	Raleigh	lives in
	2	Chuck	The Pit Authentic	likes
	2	Matt	Chuck	friends
	3	Jack	Matt	friends
	3	Jack	Raleigh	lives in
	3	Jack	Red Hot Blue	likes
	3	Matt	Jack	friends

Next, you can find “friends of Matt who like barbecue restaurants, give the restaurant a rating of 9 or higher, and live in Raleigh,” by refining the query using an FCMP link filter function and PROC NETWORK call, as follows:

```
proc cas;
  source myFilter;
  function myLinkFilter(connectionQ $, rating);
    if (connectionQ='likes') then return (rating >= 9); else return (1);
  endsub;
endsource;

loadactionset "fcmpact";
setSessOpt{cmplib="casuser.myRoutines"}; run;
fcmpact.addRoutines /
  saveTable = true,
  funcTable = {name="myRoutines", caslib="casuser", replace=true},
  package = "myPackage",
  routineCode = myFilter;
run;
quit;
proc network
  direction = directed
  nodes = mycas.NodesSocial
  links = mycas.LinksSocial
  nodesQuery = mycas.NodesSocialQuery
  linksQuery = mycas.LinksSocialQuery;
  nodesVar
    vars = (label type subtype);
  linksVar
    vars = (connection rating);
  nodesQueryVar
    vars = (label type subtype);
  linksQueryVar
    vars = (connection);
  patternMatch
    linkFilter = myLinkFilter
    outMatchNodes = mycas.OutMatchNodes
    outMatchLinks = mycas.OutMatchLinks;
run;
```

Output 6 displays the output data table **mycas.OutMatchNodes**. For this query, only one friend (X) matches the specified criteria: Jack.

Output 6 Node Mappings for Friends Who Like Barbecue (with Rating ≥ 9) and Live in Raleigh

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	

Output 7 displays the output data table **mycas.OutMatchLinks**.

Output 7 Subgraphs for Friends Who Like Barbecue (with Rating ≥ 9) and Live in Raleigh

match	from	to	connection	rating
1	Jack	Matt	friends	.
1	Jack	Raleigh	lives in	.
1	Jack	Red Hot Blue	likes	9
1	Matt	Jack	friends	.

Finally, you can find “a pair of friends of Matt who like the same barbecue restaurant, live in Raleigh, and are friends of each other,” shown in [Figure 7](#), by using the data that are created by the following DATA steps:

```

data mycas.NodesSocialQuery;
  infile datalines dsd;
  length node $40. label $40. type $40. subtype $20.;
  input node $ label $ type $ subtype $;
  datalines;
Matt,    Matt,    Person,
X,,      Person,
Y,,      Person,
Raleigh, Raleigh, City,
BBQ,,    Restaurant, BBQ
;
data mycas.LinksSocialQuery;
  infile datalines dsd;
  length from $40. to $40. connection $20.;
  input from $ to $ connection $;
  datalines;
Matt, X,      friends
X,    Matt,   friends
Matt, Y,      friends
Y,    Matt,   friends
X,    Raleigh, lives in
Y,    Raleigh, lives in
X,    BBQ,    likes
Y,    BBQ,    likes
X,    Y,      friends
Y,    X,      friends
;

```

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 friends that 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. Person X and person Y must be friends with Matt.

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

```

proc network
  direction      = directed
  nodes          = mycas.NodesSocial
  links          = mycas.LinksSocial
  nodesQuery     = mycas.NodesSocialQuery
  linksQuery     = mycas.LinksSocialQuery;

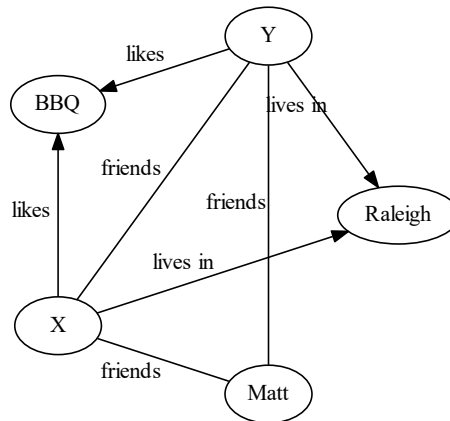
```

```

nodesVar
  vars      = (label type subtype);
linksVar
  vars      = (connection);
nodesQueryVar
  vars      = (label type subtype);
linksQueryVar
  vars      = (connection);
patternMatch
  outMatchNodes = mycas.OutMatchNodes
  outMatchLinks = mycas.OutMatchLinks;
run;

```

Figure 7 Query Graph Q



Output 8 displays the output data table **mycas.OutMatchNodes**. For this query, only one pair of friends (Chuck and Rob) match the specified criteria. There are two isomorphic mappings: one where X=Rob, Y=Chuck and one where X=Chuck, Y=Rob.

Output 8 Node Mappings for a Pair of Friends

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	Rob	Rob	Person	
1	Y	Chuck	Chuck	Person	
2	BBQ	The Pit Authentic	The Pit Authentic	Restaurant	BBQ
2	Matt	Matt	Matt	Person	
2	Raleigh	Raleigh	Raleigh	City	
2	X	Chuck	Chuck	Person	
2	Y	Rob	Rob	Person	

Output 9 displays the output data table **mycas.OutMatchLinks**.

Output 9 Subgraphs for a Pair of Friends

match	from	to	connection
1	Chuck	Matt	friends
1	Chuck	Raleigh	lives in
1	Chuck	Rob	friends
1	Chuck	The Pit Authentic	likes
1	Matt	Chuck	friends
1	Matt	Rob	friends
1	Rob	Chuck	friends
1	Rob	Matt	friends
1	Rob	Raleigh	lives in
1	Rob	The Pit Authentic	likes
2	Chuck	Matt	friends
2	Chuck	Raleigh	lives in
2	Chuck	Rob	friends
2	Chuck	The Pit Authentic	likes
2	Matt	Chuck	friends
2	Matt	Rob	friends
2	Rob	Chuck	friends
2	Rob	Matt	friends
2	Rob	Raleigh	lives in
2	Rob	The Pit Authentic	likes

Pattern Matching for Anti-Money Laundering

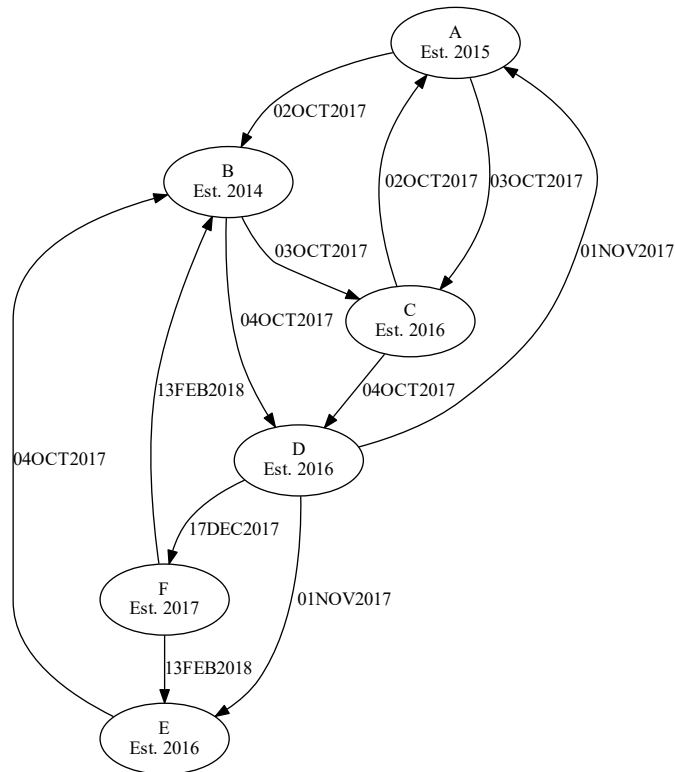
This example considers a subset of banking transactions between corporations. Money laundering can be accomplished by a series of sequential transactions through a subset of real (or fictitious) corporations that starts and ends at the same entity.

The following data provide a small subset of banking transactions between corporations:

```
data mycas.NodesAML;
  input node $ year @@;
  datalines;
A 2015 B 2014 C 2016 D 2016 E 2016 F 2017
;
data mycas.LinksAML;
  format time DATE9.;
  input from $ to $ time DATE9. @@;
  datalines;
A B 02OCT2017 A C 03OCT2017 B C 03OCT2017 B D 04OCT2017 C A 02OCT2017 C D 04OCT2017
D A 01NOV2017 D E 01NOV2017 D F 17DEC2017 E B 04OCT2017 F B 13FEB2018 F E 13FEB2018
;
```

The nodes in the nodes data table **mycas.NodesAML** represent corporations and the years they were established. The links in the links data table **mycas.LinksAML** represent banking transactions between the corporations and the date of the transaction. The data are shown graphically in [Figure 8](#).

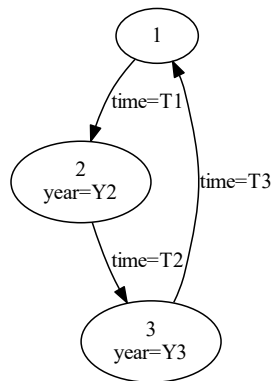
Figure 8 Banking Transactions Q



In order for investigators to focus their resources, one suspicious pattern they can search for is cyclic, sequential banking transactions through corporate entities that were established in the same year. For example, they can search for cycles of length 3.

The query graph Q that defines the pattern to search for is shown in Figure 9. In addition, the pattern requires that $Y2 = Y3$ and $T1 < T2 < T3$.

Figure 9 Query Graph Q



In order to construct this pattern, the query graph (a directed cycle of length 3) can be represented using the data that are created by the following DATA steps:

```

data mycas.NodesQuery;
  input node @@;
  datalines;
1 2 3
;
data mycas.LinksQuery;
  input from to @@;
  datalines;
1 2 2 3 3 1
;

```

The following statements specify the node and link pair filter functions. The node pair filter function **myNodePairFilter** enforces that the two inner corporations of the cycle were both established in the same year. The link pair filter function **myLinkPairFilter** enforces that the sequence of transactions in the cycle has increasing timestamps.

```

proc cas;
  source myPairFilter;
  function myNodePairFilter(nodeQ[*], year[*]);
    if (nodeQ[1]=2 and nodeQ[2]=3) then return (year[1]=year[2]); else return (1);
  endsub;
  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;
  endsource;

  loadactionset "fcmpact";
  setSessOpt{cmplib="casuser.myRoutines"}; run;
  fcompact.addRoutines /
    saveTable = true,
    funcTable = {name="myRoutines", caslib="casuser", replace=true},
    package = "myPackage",
    routineCode = myPairFilter;
  run;
quit;

```

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

```

proc network
  direction = directed
  nodes = mycas.NodesAML
  links = mycas.LinksAML
  nodesQuery = mycas.NodesQuery
  linksQuery = mycas.LinksQuery;
  nodesVar
    vars = (year);
  linksVar
    vars = (time);
  patternMatch
    nodePairFilter = myNodePairFilter
    linkPairFilter = myLinkPairFilter
    outMatchNodes = mycas.OutMatchNodes
    outMatchLinks = mycas.OutMatchLinks;
run;

```

For these data, only one subset of transactions matches the specified pattern.

Output 10 displays the output data table **mycas.OutMatchNodes**, which shows the mappings from nodes in the query graph to nodes in the input graph.

Output 10 Node Mappings for Suspicious Banking Transactions

match	nodeQ	node	year
1	1	A	2015
1	2	C	2016
1	3	D	2016

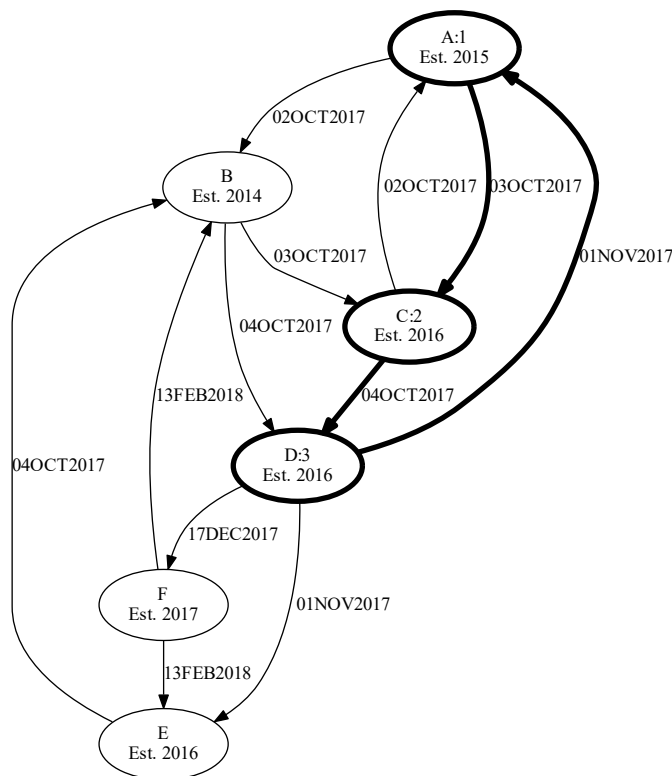
Output 11 displays the output data table **mycas.OutMatchLinks**, which shows the subgraph for the matching pattern.

Output 11 Subgraph for Suspicious Banking Transactions

match	from	to	time
1	A	C	03OCT2017
1	C	D	04OCT2017
1	D	A	01NOV2017

The result is shown graphically in Figure 10.

Figure 10 Subgraph for Suspicious Banking Transactions



COMPUTATIONAL COMPARISON

This section presents a computational and functional comparison to two popular frameworks for solving the pattern matching problem. The first framework, iGraph (Csárdi and Nepusz 2006), is a popular open-source network analysis package written in C with user interfaces to several languages, including R and Python. The second framework, Neo4j (Neo4j 2019), is a graph database platform with its own query language, Cypher.

The experiments were conducted on a machine (server) with Intel Xeon CPU X5550 @ 2.67 GHZ (2 sockets, 4 CPUs per socket) and 64 GB RAM, running Red Hat Enterprise Linux Server (release 6.3). The relevant software versions are listed below:

- SAS Visual Data Mining and Machine Learning 8.4 in SAS Viya 3.4
- iGraph 0.7.1
- Neo4j 3.5.1 Community Edition

The data, scripts, and logs used in the experiment can be found at <https://github.com/sascommunities/sas-global-forum-2019>.

Data

This experimental study considers three data sources. One data set (amazon-meta) comes from real data (Leskovec, Adamic, and Huberman 2007), two data sets (er_* and ba_*) are generated synthetically, and one data set comes from a standardized benchmark (Heflin 2019).

The amazon-meta data set is from a webcrawl over Amazon’s website. Nodes represent books, music CDs, video DVDs, and VHS tapes. Nodes are linked together if their products are likely to be purchased together. Each node is given a label that specifies the product category that the node belongs to, such as “Book: Home & Garden.”

The synthetic data (er_* and ba_*) are created by using the graph generator tools in iGraph. The er_* graphs are created according to the Erdős-Rényi model, where nodes are randomly connected using a fixed probability (Erdős and Rényi 1959). The ba_* graphs follow the Barabási-Albert model, which uses a preferential attachment mechanism when determining links between nodes. In this model, a node’s probability of being linked to other nodes is proportional to the number of existing links the other nodes have (Barabási and Albert 1999). As a result, unlike graphs generated from the Erdős-Rényi model, these graphs exhibit a power-law degree distribution, as found in scale-free networks. In both models, each node is randomly assigned a label attribute from a fixed number of unique labels. The last number in the graph name represents the number of unique labels. For example, each node in er_u_10_15_50 is labeled with one of 50 unique attributes.

The well-known Lehigh University Benchmark (LUBM) (Heflin 2019) is used as the final source of data. This is a semantic web data set that models universities’ students, professors, staff, courses, and so on, and their connections, such as a student being linked to a professor because the student is advised by that professor. LUBM is given as a Resource Description Framework (RDF) data set with an ontology that describes abstract relationships between objects. RDF, when interpreted as a graph, has a flat structure. Nodes do not have attributes. The method described in Gubichev and Then (2014) is used to convert this format into a property-graph-style format suitable for comparisons in this paper.

For each graph, the authors generated several different queries. For the amazon-meta and synthetic cases, they generated queries that had a variety of topologies, including path and path-like queries, denser topologies, and cliques (where every node is directly connected to every other node). Path and path-like queries have a diameter and link count close to the number of nodes in the query graph (for example, q01, q02, and q04 of ba_u_10_15_200). The authors generated queries of this type by performing a random walk in the graph and keeping the subgraph that was induced by the walked links. Queries with a denser topology (for example, q03 of ba_u_10_15_200) were also generated using a random walk, but keeping the subgraph induced by the nodes of the walk. The clique queries (for example, q05 of ba_u_10_15_200), which have a diameter of 1, were generated by simply finding and using one of the existing cliques of the graph.

LUBM has a set of predefined queries associated with it, and the authors used the same subset of these queries as were used in Gubichev and Then (2014). Like the data graph, these queries are converted to match the property-graph-style setup. In some cases, the queries have inexact attributes because of the ontology of LUBM. For example, if the original, predefined query looks for nodes of type student, then this paper’s query becomes an inexact match for nodes that are either of type UndergraduateStudent or of type GraduateStudent, because the ontology defines student as being one of these two types. In general, the LUBM queries have a simpler topology than the queries that are generated for the other data sets.

Table 1 gives the size of the main graph (G), the size of the query graph (Q) and its diameter (**Diam**), the number of exact (and inexact) attributes per node (**N**) and link (**L**), and the number of matches (**Matches**). Note that queries q08

and q12 of the LUBM data set are shown with a range of values because they were defined as having variable path lengths between some nodes.

Table 1 Computational Comparison Data

Main Graph			Query Graph				Attributes				Matches
Name	Nodes	Links	Name	Nodes	Links	Diam	Exact		Inexact		
							N	L	N	L	
amazon-meta	512,821	951,231	q01	4	3	2	1	0	0	0	169
			q02	10	17	3	1	0	0	0	98,680
			q03	10	9	9	1	0	0	0	48
ba_u_10_15_200	1,000,000	14,999,880	q01	5	4	4	1	0	0	0	1,486,258
			q02	5	4	4	1	0	0	0	248
			q03	5	7	2	1	0	0	0	12,132
			q04	10	9	9	1	0	0	0	11,897
			q05	10	45	1	1	0	0	0	2
ba_u_10_15_400	1,000,000	14,999,880	q01	5	4	4	1	0	0	0	265,324
			q02	5	4	4	1	0	0	0	55
			q03	7	11	3	1	0	0	0	27,939
			q04	10	9	9	1	0	0	0	51,480
			q05	10	45	1	1	0	0	0	2
			q06	20	19	18	1	0	0	0	2
er_u_10_15_20	1,000,000	15,000,000	q01	5	4	3	1	0	0	0	253,190
			q02	10	9	7	1	0	0	0	1,939,108
er_u_10_15_30	1,000,000	15,000,000	q01	5	4	3	1	0	0	0	33,065
			q02	10	9	7	1	0	0	0	31,435
			q03	20	19	14	1	0	0	0	31,307
er_u_10_15_50	1,000,000	15,000,000	q01	5	4	3	1	0	0	0	2,616
			q02	10	9	7	1	0	0	0	182
			q03	20	19	14	1	0	0	0	90
LUBM	1,082,818	3,298,813	q02	3	3	1	1	1	0	1	130
			q04	2	1	1	2	1	1	0	34
			q05	2	1	1	2	0	1	1	719
			q06	1	0	0	0	0	1	0	519,842
			q07	3	2	1	2	1	1	0	59
			q08	3-4	2-3	2-3	2	1	1	1	7,790
			q09	3	3	1	0	1	1	0	13,639
			q12	3-5	3-4	2-3	2	1	0	0	15
			q13	2	1	1	1	0	0	1	229
			q14	1	0	0	1	0	0	0	393,730

Computational Comparison to iGraph

This section compares the performance of Network to the **vf2** method in iGraph (Csárdi and Nepusz 2006) for solving the pattern matching problem. The implementation of pattern matching in iGraph (**vf2**) does not support multigraphs properly. Therefore, this computational experiment is restricted to simple graphs (excluding the amazon-meta and LUBM data sets).

Generally, the process for running a query occurs in two phases: a *loading phase* for each data set, and a *search phase* that consists of solving the pattern matching problem for a particular query.

Table 2 shows the elapsed real time (in seconds) for each data set and associated query, in addition to an overall speedup factor (**Speedup**¹) on the total time (**Total**) for executing a series of queries for a particular data set. Both

¹The speedup is calculated as a ratio of the total time it takes iGraph to perform a series of queries for a particular data set, divided by the time it takes Network to perform the same queries. A value greater than 1 implies that Network executes the queries faster.

iGraph (for a fixed random seed) and Network are deterministic in execution. That is, there is no variability with respect to running time (on an idle server). Therefore, the authors ran each experiment only once. To keep the performance comparison simple, the authors counted only the number of matches and did not create output tables.

The mechanisms for loading are slightly different between iGraph and Network. For iGraph, the *loading phase (Load)* consists of reading the graph data into memory and setting up the appropriate data structures. iGraph has the advantage that once the main graph data are loaded into memory, they can be used for subsequent queries (**Match**) on that graph without reloading. Although SAS Viya keeps the data table in memory (**LoadT**), it does not (currently) retain the graph data structures and therefore must execute part of the loading phase for each query (**Build**). An enhancement to Network to allow for persistence of the graph data structures is discussed in the section “**FUTURE DIRECTIONS**” on page 24.

Table 2 Computational Comparison iGraph versus Network

Main	Query	iGraph Time			Network Time				Speedup
		Load	Match	Total	LoadT	Build	Match	Total	
ba_u_10_15_200	q01		352			8.58	1.70		220.31
	q02		97			8.07	1.14		
	q03	45	8,933	11,776	4.45	8.11	1.21	53.45	
	q04		2,204			8.02	1.50		
	q05		144			8.64	2.03		
ba_u_10_15_400	q01		42			7.97	1.16		143.88
	q02		37			9.16	1.44		
	q03		9,525	10,478	5.00	9.15	1.71	72.83	
	q04	45	153			7.78	1.36		
	q05		68			7.77	1.46		
	q06		608			9.15	9.73		
er_u_10_15_20	q01		1,152	11,924	5.03	9.31	3.72	36.41	327.46
	q02	64	10,708			10.27	8.08		
er_u_10_15_30	q01		376			9.82	2.57		57.08
	q02	59	833	3,041	5.46	10.62	5.51	53.28	
	q03		1,774			9.85	9.44		
er_u_10_15_50	q01		126			11.20	2.30		10.52
	q02	60	144	479	5.09	9.78	2.83	45.56	
	q03		149			9.87	4.48		
Sum				37,699				261.53	
Average				12,566				87.18	151.85

From the results, you can see that Network vastly outperforms iGraph on this set of queries. On average, Network is 152 times as fast as iGraph.

Functional Comparison to Cypher

This section shows functional comparisons of pattern matching queries expressed in Network and queries expressed in the Cypher query language. Specifically, it compares three examples in detail: a straightforward query that has no inexact matches, a query that has inexact matches (requiring an FCMP node filter in Network), and a query that consists of a variable path length. The latter two examples come from the benchmark data that were used in these computational comparisons.

Cypher is the language for querying graphs that are stored in a Neo4j graph database. Neo4j employs a property graph model, where nodes and links can have an arbitrary number of attributes (key-value pairs). Nodes can also have labels associated with them, and these labels can be used to represent different groups or roles in your domain. For example, in the LUBM data set, nodes have labels such as UndergraduateStudent, AssociateProfessor, Department, and so on. When using Neo4j, you need to decide on your data model (attributes and labels) and load data into the database. Once the database is created and running, you can use Cypher to perform searches for the patterns.

Although the syntax is different, using Cypher is similar to pattern matching in Network in that you specify the topology of the subgraph along with the node and link attribute constraints that you want to match in the data graph. In Cypher,

the MATCH clause is used to specify the topology of the query graph by using ASCII-Art patterns to represent sets of nodes and links. Some node and link constraints can be specified in-line with the topology. In addition, you can specify constraints on subsets of nodes and links by using a WHERE clause after the topology specification. Generally, there are multiple ways of writing a Cypher query to obtain the same result.

As an example of the Cypher language, consider the social network in the section “[Pattern Matching in a Social Network](#)” on page 3 and the query shown in [Figure 6](#) that looks for “friends of Matt who like barbecue restaurants and live in Raleigh.” Using the PATTERNMATCH statement in PROC NETWORK to perform this query is explained in that section. To represent this query in Cypher, you can use the following syntax:

```
MATCH (Matt)-[:friends]->(X)-[:friends]->(Matt),
        (BBQ)<-[:likes]-(X)-[:`lives in`]->(Raleigh)
WHERE Matt.label="Matt" and Matt.type="Person" and X.type="Person" and
        Raleigh.label="Raleigh" and Raleigh.type="City" and
        BBQ.type="Restaurant" and BBQ.subtype="BBQ"
RETURN (X)
```

The MATCH clause specifies the topology of the query graph, which consists of an undirected link and two directed links. Some constraints (such as the type of the links) are specified in-line with the topology. The WHERE clause specifies additional constraints (such as the property values) that the nodes must have.

Next, consider query q07 of the LUBM data set, which looks for all students who take a course taught by a particular professor. Because there are two types of students, UndergraduateStudent and GraduateStudent, this query involves an inexact match on one of the nodes. The Cypher query used for q07 is as follows:

```
MATCH (S)-[:takesCourse]->(C:Course)<-[:teacherOf]-(P)
WHERE (S:GraduateStudent or S:UndergraduateStudent) and
        P.id='Department0.University0.AssociateProfessor0'
RETURN count(S)
```

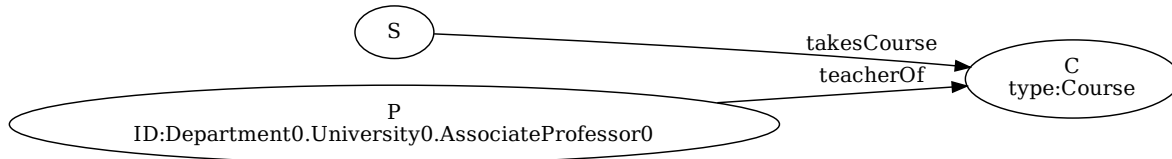
Again, the MATCH clause specifies the topology of the query graph that consists of two directed links. The type of the directed links and the requirement that node C must be labeled as Course is specified in-line with the topology. Unlike properties, to specify that some node n has label L, you simply write `n:L`. The WHERE clause constrains the professor’s ID and the inexact match that is allowed on the node that is represented by variable S, which must be labeled as a GraduateStudent or an UndergraduateStudent.

This same query can be executed in Network by specifying a topologically equivalent query graph, as shown in [Figure 11](#), along with an FCMP node filter function. The query graph can be created by the following DATA steps:

```
data sascas1.nodes;
  infile datalines dsd;
  length type $6. id $43.;
  input node $ type $ id $;
  datalines;
S,,
C, Course,
P,, Department0.University0.AssociateProfessor0
;

data sascas1.links;
  infile datalines dsd;
  length type $11.;
  input from $ to $ type $;
  datalines;
S, C, takesCourse
P, C, teacherOf
;
```

Figure 11 The Query Graph of LUBM Query q07



Along with this query graph, an FCMP node filter function is needed because node S, the unattributed node, can be a node whose type is either GraduateStudent or UndergraduateStudent. To handle this inexact match, you can use the following FCMP node filter function as part of the pattern match:

```

source myFilter;
function myNodeFilter(nodeQ $, type $);
  if(nodeQ='S') then return type in ('GraduateStudent' or 'UndergraduateStudent');
  else return (1);
endsub;
  
```

Finally, as a more complicated example, consider query q12, which has a variable path length. At a high level, q12 is looking for the number of subgraphs where a professor is the head of a department and works for a department that is a suborganization of University0. The Cypher syntax for finding the number of all such subgraphs is:

```

MATCH (P)-[:worksFor]->(D1:Department)-[:subOrganizationOf*1..2]->(U),
      (P)-[:headOf]->(D2:Department)
WHERE (P:FullProfessor) and U.id='University0'
RETURN count(P)
  
```

The suborganization relationship is transitive, meaning that in these data the path between department D1 and University0 might be of length 1 or length 2. In addition, the two departments have different variable names (D1 and D2) because (although unlikely in this context) they might not necessarily refer to the same department if a strict translation of the original, predefined RDF query is used. Interestingly, although D1 and D2 have distinct names, they are still allowed in Cypher to map to the same node in the data graph (unless an additional condition is specified in the WHERE clause). Thus, this query handles both cases where D1 and D2 are the same department and cases where D1 and D2 are different departments.

Network can perform the equivalent query with four pattern match invocations. An invocation is needed for each possible path length (in this case, length 1 and length 2), and for each of these lengths, D1 and D2 can refer to the same node or to different nodes. In total, this yields four distinct combinations whose query graphs and corresponding DATA steps are shown in Table 3. In this case, the pattern needs to be matched over the **id** and **type** variables of the nodes data and the **type** variable of the links data. No FCMP filter functions are needed here.

Table 3 Query Graphs and Corresponding DATA Steps of LUBM
Query q12

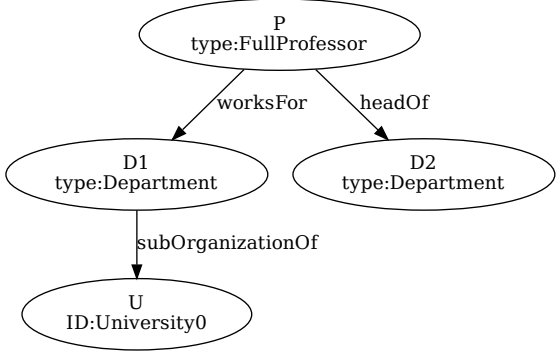
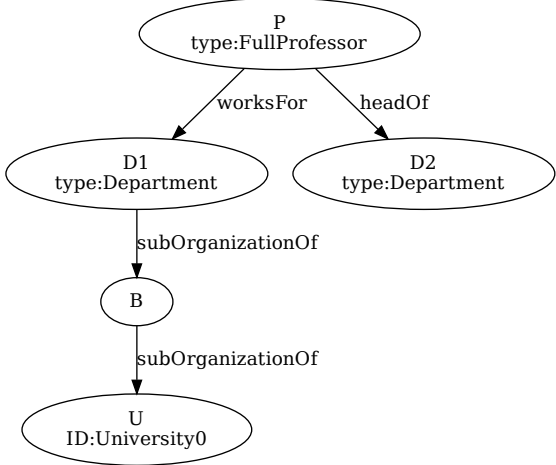
DATA Step	Query Graph
<pre> data sascas1.nodes; infile datalines dsd; length id \$11. type \$13.; input node \$ id \$ type \$; datalines; P,, FullProfessor D1,, Department D2,, Department U, University0, ; data sascas1.links; infile datalines dsd; length type \$17.; input from \$ to \$ type \$; datalines; P, D1, worksFor P, D2, headOf D1, U, subOrganizationOf ; </pre>	 <pre> graph TD P([P type:FullProfessor]) -- worksFor --> D1([D1 type:Department]) P -- headOf --> D2([D2 type:Department]) D1 -- subOrganizationOf --> U([U ID:University0]) </pre>
<pre> data sascas1.nodes; infile datalines dsd; length id \$11. type \$13.; input node \$ id \$ type \$; datalines; P,, FullProfessor D1,, Department D2,, Department U, University0, B,, ; data sascas1.links; infile datalines dsd; length type \$17.; input from \$ to \$ type \$; datalines; P, D1, worksFor P, D2, headOf D1, B, subOrganizationOf B, U, subOrganizationOf ; </pre>	 <pre> graph TD P([P type:FullProfessor]) -- worksFor --> D1([D1 type:Department]) P -- headOf --> D2([D2 type:Department]) D1 -- subOrganizationOf --> B([B]) B -- subOrganizationOf --> U([U ID:University0]) </pre>

Table 3 (continued)

DATA Step	Query Graph
<pre> data sascas1.nodes; infile datalines dsd; length id \$11. type \$13.; input node \$ id \$ type \$; datalines; P,, FullProfessor D,, Department U, University0, ; data sascas1.links; infile datalines dsd; length type \$17.; input from \$ to \$ type \$; datalines; P, D, worksFor P, D, headOf D, U, subOrganizationOf ; </pre>	
<pre> data sascas1.nodes; infile datalines dsd; length id \$11. type \$13.; input node \$ id \$ type \$; datalines; P,, FullProfessor D,, Department U, University0, B,, ; data sascas1.links; infile datalines dsd; length type \$17.; input from \$ to \$ type \$; datalines; P, D, worksFor P, D, headOf D, B, subOrganizationOf B, U, subOrganizationOf ; </pre>	

Computational Comparison to Neo4j

Neo4j is a graph database that has some graph analytical functionality. Network is a graph engine that is designed primarily for analytical functions and relies on standard SAS Viya data management tools (through tables) for manipulation of data. This computational study focuses on a performance comparison of the graph engines. Although the workflow for processing a query differs conceptually in Neo4j versus Network, the end results (the matches) are exactly the same.

There are a number of alternative ways to execute queries in Neo4j. The process chosen for running a query in Neo4j consists of two main phases in four parts. For a particular data set, the *loading phase* consists of creating the database and warming up the data. The first part of the loading phase consists of loading the data set into the database (**CreateDB**). The second part of the loading phase consists of *warming up* the database (**Warmup**), which means bringing the graph data into memory for faster execution. For each query of a particular data set, the *search phase* consists of constructing a plan for the search execution and then running the search. There is a great deal of variance in the observed execution time each time you run a query, particularly for queries that have more complex

topologies. It seems that the generation of plans is done nondeterministically and can have a large effect on overall execution time. That is, some plans perform much worse than other plans. For this reason, the authors ran the Neo4j search phase for each query five times and report the average execution time (**Match**).

Table 4 gives the elapsed real time (in seconds) for each data set and associated query.

Table 4 Computational Comparison Neo4j versus Network

Main	Query	Neo4J Time				Network Time				Speedup	
		Create	Warm	Match	Total	LoadT	Build	Match	Total		
amazon-meta	q01			1.33			1.02	0.31			
	q02	4.977	1.01	41.63	50.64	0.73	1.01	0.31	4.64	10.92	
	q03			1.69			1.00	0.26			
ba_u_10_15_200	q01			7.50			8.58	1.70			
	q02			3.18			8.07	1.14			
	q03	18.897	5.63	36.96	86.94	4.45	8.11	1.21	42.78	2.03	
	q04			14.77			8.02	1.50			
	q05			**			8.64	2.03			
ba_u_10_15_400	q01			5.25			7.97	1.16			
	q02			3.07			9.16	1.44			
	q03	17.321	5.55	327.14	394.09	5.00	9.15	1.71	63.60	6.20	
	q04			11.17			7.78	1.36			
	q05			**			7.77	1.46			
	q06			24.58			9.15	9.73			
er_u_10_15_20	q01			13.16			9.31	3.72			
	q02	17.873	5.60	44.29	80.92	5.03	10.27	8.08	36.41	2.22	
er_u_10_15_30	q01			9.38			9.82	2.57			
	q02	17.014	5.55	19.77	97.35	5.46	10.62	5.51	53.28	1.83	
	q03			45.64			9.85	9.44			
er_u_10_15_50	q01			6.74			11.20	2.30			
	q02	18.046	5.50	14.49	78.28	5.09	9.78	2.83	45.56	1.72	
	q03			33.51			9.87	4.48			
LUBM	q02			1.28			2.77	0.81			
	q04			0.25			3.04	0.70			
	q05			0.10			2.99	1.16			
	q06			1.14			2.36	0.27			
	q07			0.58			2.76	1.60			
	q08	20.265	4.27	0.13	33.82	4.10	5.76	2.26	60.04	0.56	
	q09			4.44			3.05	6.10			
	q12			0.12			12.07	1.55			
	q13			1.22			2.87	1.41			
	q14			0.02			2.31	0.09			
	Sum									822.04	306.31
	Average									227.64	86.85
											3.64

From the results, you can see that Network outperforms Neo4j on the majority of queries in this study. On average, Network is 3.6 times as fast as Neo4j. The one exception is the LUBM standard benchmark, where Neo4j is almost twice as fast as Network.

For LUBM, if you focus on just the search phase, the total execution time for the 10 queries is 9.29 seconds for Neo4j, and 15.96 seconds for Network. The difference is negligible. In this case, the disadvantage of needing to rebuild the graph data structures for each query adds significant overhead for Network. In addition, as discussed in the section “[Functional Comparison to Cypher](#)” on page 18, queries q08 and q12 require multiple executions of Network versus one execution of Neo4j to find the same set of matches. Overcoming this deficit is discussed in the section “[FUTURE DIRECTIONS](#)” on page 24.

Despite these current deficiencies, Network still greatly outperforms Neo4j on all complex queries. Recall from [Table 1](#) that the LUBM queries are quite small (at most four links) in comparison to the majority of other queries in this study.

Another factor in comparing Neo4j and Network is the use of memory. The authors did not perform a thorough analysis of memory consumption. However, they did observe that for the two clique queries (ba_u_10_15_200/q05 and ba_u_10_15_400/q05) Neo4j failed to complete because of insufficient memory after more than one hour of execution time. In this study, the heap memory size given to Neo4j was 24GB, as recommended by their utility. That is, Neo4j requires more than 24GB to complete these queries. In contrast, the maximum amount of memory used by Network for any specific query was 1.9GB.

FUTURE DIRECTIONS

In future releases of Network, several new features are being considered to improve the convenience and flexibility of the interface, as well as to improve performance.

On the interface side, one particular area of interest is the automation of variable path topologies. As discussed in the section “[Functional Comparison to Cypher](#)” on page 18, Neo4j provides the ability to perform variable path length topologies using one Cypher language request (using the `1**n` construct). In order to perform the same queries in Network, the topology of each specific path length must be requested separately, leading to unnecessary overhead. In a future release, the authors will consider automating this type of query into one request.

On the performance side, allowing for persistent storage of graph data structures can improve overall execution time when numerous queries are to be requested against the same data source. As discussed in the section “[Computational Comparison to Neo4j](#)” on page 22, although the search phase for Neo4j and Network perform roughly the same for the LUBM example, the additional overhead of reloading hurts overall performance for Network. In a future release, it will be possible to retain the graph data structures within the same session. A prototype code yields the results that are shown in [Table 5](#).

Table 5 Computational Comparison Neo4j versus Network (Persistent Data)

Main	Query	Neo4J Time				Network Time				Speedup
		Create	Warm	Match	Total	LoadT	Build	Match	Total	
amazon-meta	q01			1.33				0.31		18.48
	q02	4.977	1.01	41.63	50.64	0.75	1.11	0.31	2.74	
	q03			1.69				0.27		
ba_u_10_15_200	q01			7.50				1.34		5.25
	q02			3.18				1.14		
	q03	18.897	5.63	36.96	86.94	2.88	8.53	1.22	16.58	
	q04			14.77				1.47		
	q05			**				1.60		
ba_u_10_15_400	q01			5.25				1.14		15.09
	q02			3.07				1.10		
	q03			327.14				1.34		
	q04	17.321	5.55	11.17	394.09	3.32	8.50	1.36	26.12	
	q05			**				1.46		
	q06			24.58				9.37		
er_u_10_15_20	q01			13.16				4.58		2.94
	q02	17.873	5.60	44.29	80.92	2.94	10.70	9.27	27.50	
er_u_10_15_30	q01			9.38				2.45		3.43
	q02	17.014	5.55	19.77	97.35	2.91	10.12	4.34	28.42	
	q03			45.64				8.60		
er_u_10_15_50	q01			6.74				1.91		3.24
	q02	18.046	5.50	14.49	78.28	2.94	11.68	2.85	24.16	
	q03			33.51				4.78		

Table 5 (continued)

Main	Query	Neo4J Time				Network Time				Speedup	
		Create	Warm	Match	Total	LoadT	Build	Match	Total		
LUBM	q02			1.28				0.76		1.68	
	q04			0.25				0.58			
	q05			0.10				1.00			
	q06			1.14				0.28			
	q07			0.58				1.62			
	q08	20.265	4.27	0.13	33.82	1.90	2.95	2.16	20.14		
	q09			4.44				6.13			
	q12			0.12				1.33			
	q13			1.22				1.35			
	q14			0.02				0.09			
	Sum								145.66		
	Average								41.22		7.16

From the results, you can now see that this prototype version of Network outperforms Neo4j across all data sources in this study. On average, this version of Network is 7.2 times as fast as Neo4j (1.7 times as fast on the LUBM benchmark).

CONCLUSION

This paper introduces a new feature in the network analytics package of SAS Visual Data Mining and Machine Learning for solving the pattern matching problem. The breadth of applications for this feature across several industries makes it an important area of focus. This paper gives some simple examples of usage with the NETWORK procedure. In addition, this paper describes a computational comparison against two popular network analysis frameworks, iGraph and Neo4j. For those familiar with Neo4j's Cypher language, the section on translating a Cypher query into a Network query should serve as a good starting reference point. In future releases, the authors intend to expand the set of available features to cover more complex queries.

From the results, it is clear that Network vastly outperforms both iGraph and Neo4j with respect to execution time.

REFERENCES

- Aggarwal, C. C., and Wang, H., eds. (2010). *Managing and Mining Graph Data*. Boston: Springer.
- Barabási, A.-L., and Albert, R. (1999). "Emergence of Scaling in Random Networks." *Science* 286:509–512.
- 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.
- Csárdi, G., and Nepusz, T. (2006). "The iGraph Software Package for Complex Network Research." *InterJournal of Complex Systems* 1695:1–9. <http://igraph.org>.
- Erdős, P., and Rényi, A. (1959). "On Random Graphs I." *Publicationes Mathematicae Debrecen* 6:290–297.
- 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.
- Gubichev, A., and Then, M. (2014). "Graph Pattern Matching: Do We Have to Reinvent the Wheel?" In *Proceedings of Workshop on GRAph Data Management Experiences and Systems (GRADES '14)*, 8:1–8:7. New York: ACM. <http://doi.org/10.1145/2621934.2621944>.
- Heflin, J. (2019). "SWAT Projects—the Lehigh University Benchmark (LUBM)." <http://swat.cse.lehigh.edu/projects/lubm/index.htm>.

- Leskovec, J., Adamic, L. A., and Huberman, B. A. (2007). "The Dynamics of Viral Marketing." *ACM Transactions on the Web* 1:1–39.
- Neo4j (2019). "Neo4j: The World's Leading Graph Database." <https://neo4j.com/>.
- SAS Institute Inc. (2018a). *Base SAS 9.4 Procedures Guide*. 7th ed. Cary, NC: SAS Institute Inc. <https://go.documentation.sas.com/?docsetId=proc&docsetTarget=titlepage.htm&docsetVersion=9.4&locale=en>.
- SAS Institute Inc. (2018b). *An Introduction to SAS Viya 3.4 Programming*. Cary, NC: SAS Institute Inc. https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=pgmdiff&docsetTarget=titlepage.htm&locale=en.
- SAS Institute Inc. (2018c). *SAS Visual Data Mining and Machine Learning 8.3: Programming Guide*. Cary, NC: SAS Institute Inc. https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=casactml&docsetTarget=titlepage.htm&locale=en.
- SAS Institute Inc. (2018d). *SAS Visual Data Mining and Machine Learning 8.3: The NETWORK Procedure*. Cary, NC: SAS Institute Inc. https://go.documentation.sas.com/?cdcId=pgmsascdc&cdcVersion=9.4_3.4&docsetId=casmlnetwork&docsetTarget=titlepage.htm&locale=en.
- 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.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author:

Matthew Galati
SAS Institute Inc.
610-232-4931
Matthew.Galati@sas.com

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.