

**ΟΙΚΟΝΟΜΙΚΟ  
ΠΑΝΕΠΙΣΤΗΜΙΟ  
ΑΘΗΝΩΝ**



**ATHENS UNIVERSITY  
OF ECONOMICS  
AND BUSINESS**

# Social Network Analysis

Student thesis

**Ioannis Xarchakos**

Athens University of Economics and Business

Supervised by:

**Prof. Vasilis Vassalos**

Athens, March 2016

## Abstract

This thesis investigates Social Network Theory using graph processing systems with purpose to find interesting properties and relationships in different kinds of social networks. We use several kinds of networks: online social networks, collaboration networks, product networks and business email networks. We use mainly four centrality metrics which is Degree centrality, Closeness centrality, Betweenness centrality and Eigenvector centrality in order to investigate interesting properties into the graphs. We use three processing systems; Stanford's Network Analysis Platform (SNAP), Neo4j and Apache Giraph. We use the four centrality metrics in order to compare the performance of the three systems. We also investigate the correlation between research productivity and research collaboration using Betweenness centrality metric over the years.

## Acknowledgements

First and foremost, I would like to thank professor V. Vassalos for his willingness to be my supervisor and for his valuable guidance.

# Table of Contents

<b>1. Overview .....</b>	<b>5</b>
1.1 Goals.....	5
1.2 Used techniques .....	5
1.3 Results .....	6
<b>2. Related Work .....</b>	<b>7</b>
2.1 Network analysis .....	7
2.2 Centrality measures .....	7
2.3 Graph analysis systems .....	8
<b>3. Centrality Measures .....</b>	<b>9</b>
3.1 Mathematical definitions .....	9
3.2 Intuitions.....	11
3.3 Examples .....	12
3.4 Pseudocode .....	14
<b>4. Graph Systems .....</b>	<b>16</b>
4.1 SNAP.....	16
4.1.1 Degree Centrality .....	18
4.1.2 Closeness Centrality.....	19
4.1.3 Betweenness Centrality.....	19
4.1.4 Eigenvector Centrality .....	20
4.1.5 Clustering Coefficient .....	20
4.1.6 K-Core.....	21
4.2 Neo4j .....	22
4.2.1 Degree Centrality - Cypher .....	26
4.2.2 Closeness Centrality.....	27
4.2.3 Betweenness Centrality.....	29
4.2.4 Eigenvector Centrality .....	30
4.2.5 Clustering Coefficient .....	32

4.3 Apache Giraph.....	33
4.3.1 Degree Centrality .....	40
4.3.2 Closeness Centrality.....	42
4.3.3 Eigenvector Centrality .....	47
4.4 Experimental Comparison.....	49
4.4.1 DBLP .....	49
4.4.2 Wiki-Talk .....	51
4.4.3 LiveJournal .....	52
4.4.4 Twitter .....	53
4.4.5 Amazon product network .....	54
4.4.6 Email Enron .....	56
<b>5. Conclusion .....</b>	<b>58</b>
5.1 Graph Analysis Platforms Performance Comparison .....	58
5.2 Research productivity and Research collaboration correlation .....	60
<b>Appendix A .....</b>	<b>65</b>

# 1. Overview

Social network analysis (SNA) is the process of investigating social structures through the use of network and graph theories. It characterizes networked structures in terms of nodes (individual actors, people, or things within the network) and the ties or edges (relationships or interactions) that connect them. The extreme popularity and rapid growth of these social networks represents a unique opportunity to study, understand, and leverage their properties. In this thesis, we investigate online social networks such as Twitter and LiveJournal, collaboration networks such as DBLP, product networks like Amazon's product and network and business email networks such as Enron's e-mail network.

## 1.1 Goals

This section presents the goals of this thesis. Our first goal is to apply the social network principles and methodologies using centrality metrics including Degree centrality, Betweenness centrality, Closeness centrality and Eigenvector centrality. The second goal is to compare the performance of the three graph analysis platforms which we use to calculate the previously mentioned metrics. Finally, we investigate the correlation between research productivity and research collaboration using Betweenness centrality metric calculated over the years in the DBLP collaboration network.

## 1.2 Used techniques

This section presents how we collect our data and how we calculate the different centrality metrics. We acquire the Twitter dataset from Arizona State University repository [15], the LiveJournal and the Amazon product network data sets acquired from Stanford's Large Network Dataset Collection repository [14]. We also construct DBLP and Enron's e-mail network dataset. DBLP is a computer science bibliography website hosted at University of Trier, in Germany. It was originally a database and logic programming bibliography site, and has existed at least since the 1980s. DBLP listed more than 3.5 million journal articles, conference papers, and other publications on computer science. We parse each publication and for each scientific collaboration between two or more authors we construct links among them in pairs. With this method we construct a dense collaboration network. In order to construct Enron's e-mail network data set, we acquire all the e-mails among Enron's employees from Enron Email Dataset repository which hosted by Carnegie Mellon University. The Enron corpus is a large database with over 600,000 emails generated by 158 employees of the Enron Corporation and acquired by the Federal Energy Regulatory Commission during its investigation after the company's collapse. Then, we parse every e-mail in the data set and create a directed link from the sender to recipient of every e-mail.

The next step is to calculate the centrality metrics for each one of our data sets. For this purpose, we use three different graph analysis systems. The first system which we used is the Stanford's Network Analysis Platform (SNAP). The SNAP is a general purpose network analysis and graph mining library. It follows a graph-centric programming

model and it can efficiently manipulate large graphs, calculate structural properties and support attributes on nodes and edges. The second system is the Neo4j graph database. Neo4j is an open-source well-known graph database implemented in Java which uses an intuitive graph-oriented model for data representation. The last system which we used is the Apache Giraph. Apache Giraph is an iterative graph processing system built for high scalability and it is the open-source counterpart to Pregel, the graph processing architecture developed at Google. Both systems are inspired by the Bulk Synchronous Parallel model of distributed computation. Apache Giraph is part of the Hadoop ecosystem and runs on Hadoop. Apache Hadoop is an open-source software framework for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware.

In order to visualize the node connections in a network and also visualize the results about the correlation between research productivity and research collaboration, we use the D3.js which is a Javascript library. It is a library for producing dynamic, interactive data visualizations in web browsers and it makes use of the widely implemented SVG, HTML5, and CSS standards.

### **1.3 Results**

In this section, we summarize briefly our observations. In section 5 we refer in detail to our results. We compare the performance of the three graph processing systems using four centrality algorithms on six different datasets. The first system, SNAP, it is very efficient in calculation of Degree and Eigenvector centrality on small or medium sized data sets but underperformed in huge data sets like Twitter and it is also not perform well in more complex algorithms. Neo4j executes the calculations for complex metrics such as Closeness and Betweenness centrality very fast, however, Neo4j needs lot of memory to store large graphs. Apache Giraph calculates the Degree centrality metric 4 times faster than the second best system in a very large data set such as Twitter which includes 11,316,811 nodes and 85,331,845 edges. In this thesis, it is also investigated the correlation between research productivity and research collaboration. In order to measure the research productivity, we count the total papers of each author using DBLP data set. Given all the analyses we performed it is very easy to use a system that can measure the research collaboration using Betweenness centrality metric. We calculate Betweenness centrality and total publications first as a baseline from 1990 to 1999 and then for every additional year till 2005. Unfortunately, our hypothesis that there is a relationship between publication count (i.e., productivity) and collaboration “strength” as measured by Betweenness centrality does not seem to be correct, however, we find out “important” nodes which meet our proposal.

## 2. Related Work

In this chapter, we describe prior work related to the topics presented in this thesis. As this thesis covers a number of different topics, the related work has been grouped into sections detailing (2.1) work that examines the properties of different networks, (2.2) work that examines the centrality measures in detail and (2.3) work that compares the efficiency of graph analysis systems.

### 2.1 Network analysis

As online social networks gained popularity, researchers have begun to investigate their properties extensively.

Mislove et al. [1] studied large scale online social networking sites like Orkut, YouTube, and Flickr, and found that these networks comply with the power-law, small-world, and scale free properties. They also observe that the indegree of user nodes tends to match the outdegree and the networks contain a densely connected core of high-degree nodes.

Cha et al. [2] examined the information propagation in the Flickr social network. They answering three key questions; how widely does information propagate in the social networks, how quickly does information propagate and what is the role of word-of-mouth exchanges between friends in the overall propagation of information in the network.

Viswanath et al. [3] investigate the evolution of user interaction in Facebook. They study the evolution of activity between users in the Facebook social network and find that links in the activity network tend to come and go rapidly over time, and the strength of ties exhibits a general decreasing trend of activity as the social network link ages.

Cha et al. [4] measure the user influence in Twitter social network. First, they find that popular users who have high indegree are not necessarily influential in terms of spawning retweets or mentions. Second, most influential users can hold significant influence over a variety of topics and finally, the influence is not gained spontaneously or accidentally, but through concerted effort such as limiting tweets to a single topic.

McAuley and Leskovec [5] study the problem of automatically discovering users' social circles. They introduce a dataset of 1,143 ego-networks from Facebook, Google+, and Twitter, for which they obtain hand-labeled ground-truth from 5,636 different circles. They use an unsupervised method which is able to automatically determine both the number of circles as well as the circles themselves.

### 2.2 Centrality measures

Bader and Madduri [6] discuss fast parallel algorithms for evaluating several centrality indices frequently used in complex network analysis. These algorithms have been optimized to exploit properties typically observed in real-world large scale networks.

Brandes and Pich [7] conducted a series of experiments to assess the practicality of heuristic methods for centrality computation in order to calculate centralities from a limited number of SSSP computations.

Ulrik Brandes [8] presents a faster algorithm for Betweenness centrality. He shows that Betweenness centrality can be computed exactly even for fairly large networks. He introduces more efficient algorithms based on a new accumulation technique that



integrates well with traversal algorithms solving the single-source shortest-paths problem, and thus exploiting the sparsity of typical instances. The range of networks for which Betweenness centrality can be computed is thereby extended significantly. Sariyuce et al. [9] provided fast incremental algorithms for closeness centrality computation. Their algorithms efficiently compute the closeness centrality values upon changes in network topology, i.e., edge insertions and deletions. Edmonds et al. [10] created a space-efficient parallel algorithm for computing Betweenness centrality in distributed memory. Parallelization over the vertex set of the standard algorithm, with a final reduction of the centrality for each vertex, is straightforward but requires  $O(|V|^2)$  storage. They present a new parallelizable algorithm with low spatial complexity that is based on the best known sequential algorithm. Their algorithm requires  $O(|V| + |E|)$  storage and enables efficient parallel execution.

## 2.3 Graph analysis systems

Han et al. [11] they conduct a study to experimentally compare Giraph, GPS, Mizan, and GraphLab on equal ground by considering graph and algorithm agnostic optimizations and by using several metrics. The systems are compared with four different algorithms (PageRank, single source shortest path, weakly connected components, and distributed minimum spanning tree) on up to 128 Amazon EC2 machines. They find that the system optimizations present in Giraph and GraphLab allow them to perform well.

Jouili and Vansteenbergh [12] they present a distributed graph database comparison framework and the results they obtained by comparing four important players in the graph databases market: Neo4j, OrientDB, Titan and DEX. Their study focused on two key issues: the growing size of the datasets and the increase of data complexity.

Koch et al. [13] they describe and compare programming models for distributed computing with a focus on graph algorithms for large-scale complex network analysis. Four frameworks – GraphLab, Apache Giraph, Giraph++ and Apache Flink – are used to implement algorithms for the representative problems Connected Components, Community Detection, PageRank and Clustering Coefficients. The implementations are executed on a computer cluster to evaluate the frameworks' suitability in practice and to compare their performance to that of the single-machine, shared-memory parallel network analysis package NetworKit. Out of the distributed frameworks, GraphLab and Apache Giraph generally show the best performance.

### 3. Centrality Measures

Graph theory and network analysis use various measures to determine the relative importance of a vertex within the graph. There are four centrality measures that are widely used in network analysis: Degree centrality, Closeness centrality, Betweenness centrality and Eigenvector centrality.

#### 3.1 Mathematical definitions

##### *Degree centrality*

Degree centrality defined as the number of links incident upon a node. It is a measurement of node's connectedness. In the case of a directed network we define two separate measures of Degree centrality, namely In-Degree and Out-Degree.

In-Degree is the total number of ties directed to the node and Out-Degree is the total number of ties that the node directs to other nodes of the network. The Degree centrality of a vertex  $v$ , for a given graph  $G: = (V, E)$  with  $|V|$  vertices and  $|E|$  edges, is defined as

$$C_D(v) = \deg(v)$$

Normalized version divides simple degree by the maximum degree possible, which is  $N-1$ , yielding measure ranging from 0 to 1.

##### *Closeness centrality*

Closeness centrality is based on the length of the average shortest path between a vertex and all vertices in the network. The more central a node is the lower its total distance from all other nodes. In directed graphs distances to a node are considered a more meaningful measure of centrality because a node usually has little control over its incoming links. The closeness centrality can be formally represented as

$$C(x) = \frac{1}{\sum_y d(y, x)}$$

Normalized version divides closeness centrality by the total nodes of the network, which is  $N-1$ , yielding measure ranging from 0 to 1.

##### *Betweenness centrality*

A more complex centrality measure is the Betweenness centrality. It relies on the concept of shortest paths. In order to compute the Betweenness centrality of a vertex, it is necessary to count the number of shortest paths that pass across the given vertex.

The betweenness centrality  $CB(v_i)$  of a vertex  $v_i$  is computed as

$$C_B(v_i) = \sum_{v_s \neq v_i \neq v_t \in V} \frac{\sigma_{st}(v_i)}{\sigma_{st}}$$

where  $\sigma_{st}$  is the number of shortest paths between vertices  $v_s$  and  $v_t$  and  $\sigma_{st}(v_i)$  is the number of shortest paths between  $v_s$  and  $v_t$  that pass through  $v_i$ .

### ***Brandes Betweenness centrality***

The fastest known algorithm for exactly computing betweenness of all the vertices, until now, designed by Ulrik Brandes and requires at least  $O(nm)$  time for unweighted graphs and  $O(nm + n^2 \log n)$  time for weighted graphs, where  $n$  is the number of vertices and  $m$  is the number of edges.

The dependency of a source vertex  $s \in V$  on a vertex  $v \in V$  defined as

$$\delta_{s*}(v) = \sum_{t \neq s \neq v \in V} \delta_{st}(v)$$

Then the betweenness centrality of  $v$  can be then expressed as

$$BC(v) = \sum_{s \neq v \in V} \delta_{s*}(v)$$

Also, let  $P_s(v)$  denote the set of predecessors of a vertex  $v$  on shortest paths from  $s$

$$P_s(v) = \{u \in V : \langle u, v \rangle \in E, d(s, v) = d(s, u) + w(u, v)\}$$

Brandes proved that the dependencies satisfy the following relation, which is the most important in the algorithm analysis.

The dependency of  $s \in V$  on any  $v \in V$

$$\delta_{s*}(v) = \sum_{w: v \in P_s(w)} \frac{\lambda_{sv}}{\lambda_{sw}} (1 + \delta_{s*}(w))$$

For every node in  $V$ ,  $n$  Single Source Shortest Paths computations are calculated. The predecessor sets  $P_s(v)$  are retained during these calculations. The next step is every node to compute the dependencies  $\delta_{s*}(v)$  for all other  $v \in V$  using the information from the shortest paths tree and predecessor sets along the paths. The calculation is over when the sum of all dependency values are computed.

### ***Eigenvector centrality***

Another way to assign the centrality in a vertex is based on the idea that if a vertex has many central neighbors, it should be central as well. This measure is called Eigenvector

centrality and establishes that the importance of a vertex is determined by the importance of its neighbors.

For a given graph  $G = (V, E)$  with  $|V|$  number of vertices let  $A = (a_{v,t})$  be the adjacency matrix,  $a_{v,t} = 1$  if vertex  $v$  is linked to vertex  $t$ , and  $a_{v,t} = 0$  otherwise. The centrality score of vertex  $v$  can be defined as

$$x_v = \frac{1}{\lambda} \sum_{t \in M(v)} x_t = \frac{1}{\lambda} \sum_{t \in G} a_{v,t} x_t$$

where  $M(v)$  is a set of the neighbors of  $v$  and  $\lambda$  lambda is a constant. With a small rearrangement this can be rewritten in vector notation as the eigenvector equation

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

## 3.2 Intuitions

### *Degree centrality*

Degree centrality is the simplest measure of node connectivity. The node with the most connections is the most important. It is useful for finding very connected individuals, popular individuals, individuals who are likely to hold most information or individuals who can quickly connect with the wider network. With directed data, however, it can be important to distinguish centrality based on In-Degree from centrality based on Out-Degree. If a node receives many ties, they are often said to be prominent, or to have high prestige. Many other nodes seek to direct ties to them, and this may indicate their importance. Nodes who have unusually high Out-Degree are nodes who are able to exchange with many others, or make many others aware of their views. Nodes who display high Out-Degree centrality could be influential nodes.

Possible questions to answer:

- Who is the most / least popular person in the network?
- Who can call upon the most resource in the network?

### *Closeness centrality*

An intuitive definition of Closeness centrality is that the most important node is the one that is the most independent. Nodes that require the fewest number of edges to transfer information to all other nodes are considered the most independent. In practice, the Closeness centrality calculates the importance of a vertex on how close the given vertex is to the other vertices. Central vertices, with respect to this measure, are important as they can reach the whole network more quickly than non-central vertices. Closeness is most insightful when a network is sparsely connected. In a highly connected network you will often find all nodes have a similar score.

Possible questions to answer:

- Who can most efficiently obtain information on other nodes in the network?
- Who could most quickly spread information in a network?

### ***Betweenness centrality***

Betweenness centrality identifies the nodes with the most strategic location in a network. It favors nodes that join communities rather than nodes that lie inside a community, in other words shows which nodes act as ‘bridges’ between nodes in a network. Vertices with high values of Betweenness centrality are important because maintain an efficient way of communication inside a network and foster the information diffusion.

Possible questions to answer:

- Who or what can most strongly control information flow around the network?
- Who or what would cause the most disruption to flow if they were removed?

### ***Eigenvector centrality***

An intuitive definition of Eigenvector centrality is that the node that has the most important neighbors is the most important node in the network, in other words you are only important if you have important neighbors. High Eigenvector centrality score indicates a strong influence over other nodes in the network. It is useful because it indicates not just direct influence, but also implies influence over nodes more than one ‘hop’ away.

Possible questions to answer:

- Who or what holds wide-reaching influence in my network?
- Who or what is important in my network on a macro scale?

## **3.3 Examples**

The four centrality metrics have several applications on networks. We can utilize their properties and draw conclusions on social, collaboration, product and other types of networks.

### ***Degree centrality***

In Social networks the Degree centrality is useful for indicating active or popular users of the network. If an actor receives many ties, they are often said to be prominent, or to have high prestige. Actors who have unusually high Out-Degree are actors who are able to exchange with many others, or make many others aware of their views. Actors who display high out-degree centrality are often said to be influential actors. In Collaboration networks this measure is useful for indicating how collaborative each author is. In Product networks this metric is useful for indicating

popular products and products which purchased along with other products in the network.

### ***Closeness centrality***

In Social networks this centrality measure can help find good ‘broadcasters’, but in a highly connected network you will often find all nodes have a similar score. What may be more useful is using Closeness to find influencers within a small community (single cluster). A useful application of this metric is that users with high Closeness centrality inside a community can be recommended as potential friends to the other members of this community.

In Collaboration networks Closeness centrality shows authors that they possessed and controlled a great deal of research. In a collaborative network, the closer one author is to the other author, more easily are information communication and research collaboration. The closeness centrality is also useful for understanding the authors that are easily accessible by all authors on average.

In a Product network products with high Closeness centrality inside a “community” can be recommended as similar to the other products of this product “community”.

### ***Betweenness centrality***

In Social networks Betweenness centrality shows which nodes are more likely to be in communication paths between other nodes. High Betweenness centrality nodes connect large groups of people and it is possible to be able to diffuse information in large portion of the network. For example, they could promote a product or endorse a political party. High Betweenness centrality nodes do not necessarily have high Degree or Closeness centrality.

In Collaboration networks nodes with high Betweenness centrality indicated that they had the power to control collaborative relationship and possessed and controlled a great deal of research resource. They also tend to attract more new co-authors than the well-connected authors (high Degree centrality) or the authors who are close to all others (high Closeness centrality), in other words, new authors prefer to attach to the existing authors who are controlling the flow of information (communication) by having a brokering (or bridging) role in the collaboration network. Obviously, as the collaboration network grows authors with high Betweenness centrality gain more power and influence.

In Product networks this centrality measure identifies the nodes with the most strategic location in a network such as nodes which join communities. In a product network, products with high Betweenness could be the “core” products of the network. With the term “core”, we indicate products which regardless of what you purchase, it's relatively easy to end up at those products.

## ***Eigenvector centrality***

Eigenvector centrality is associated with prestige and influence. High Eigenvector score indicates a strong influence over other nodes in the network. A node may have a high Degree score (i.e. many connections) but a relatively low Eigenvector score if many of those connections are with similarly low-scored nodes. In other words you are important if you have important neighbors.

In Collaboration networks Eigenvector centrality is usually associated with prestige because authors with high values in this metric have formed collaborations with important authors.

In terms of a Product network, products which bought along with products with high Degree will have high Eigenvector centrality.

## **3.4 Pseudocode**

### ***Degree centrality***

```
for i from 1 to |V|
  for j from 1 to |V|
    if matrix[i][j] != 0 then
      degree[i]++;
    end if
```

### ***Closeness centrality***

```
for i from 1 to |V|
  execute dijkstra(i) -> vector path[i];
  for j from 1 to |V|
    path_length[i] += path[i][j];
  closeness_cent[i] = 1.0/path_length[i];
```

### ***Betweenness centrality***

```
for i from 1 to |V|
  execute dijkstra(i) -> vector path[i];
  for j from 1 to |V|
    path_length[i] += path[i][j];
  closeness_cent[i] = 1.0/path_length[i];
```

## *Eigenvector centrality*

```
for i from 1 to |V|
  eigenvector[i] <- 1;
for n from 1 to MAX_TIMES
  for i from 1 to |V|
    tmp_eigen[i] <- 0;
  for j from 1 to |V|
    tmp_eigen[i] += matrix[i][j]*eigenvector[j];
  norm_sq <- 0;
  for i from 1 to |V|
    norm_sq <- tmp_eigen[i]*tmp_eigen[i];
  norm <- sqrt(norm_sq);
  for i from 1 to |V|
    eigenvector[i] <- tmp_eigen[i]/norm;
```

## *Brandes Betweenness centrality*

```
CB[v] ← 0, v ∈ V;
for s ∈ V do
  S ← empty stack;
  P[w] ← empty list, w ∈ V;
  σ[t] ← 0, t ∈ V ; σ[s] ← 1;
  d[t] ← -1, t ∈ V ; d[s] ← 0;
  Q ← empty queue;
  enqueue s → Q;
  while Q not empty do
    dequeue v ← Q;
    push v → S;
    foreach neighbor w of v do
      if d[w] < 0 then
        enqueue w → Q;
        d[w] ← d[v] + 1;
      if d[w] = d[v] + 1 then
        σ[w] ← σ[w] + σ[v];
        append v → P[w];
  δ[v] ← 0, v ∈ V ;
  while S not empty do
    pop w ← S;
    for v ∈ P[w] do
      δ[v] ← δ[v] + σ[v]/σ[w] * (1 + δ[w]);
  if w != s then
    CB[w] ← CB[w] + δ[w];
```



## 4. Graph Systems

In this chapter, we describe the three graph processing systems which presented in this thesis and compare their performance in the used data sets. This part of the thesis has been grouped into sections detailing (4.1) the Stanford Network Analysis Platform (SNAP) and the algorithms which we developed using this graph library, (4.2) the Neo4j and the algorithms which we developed using this graph database system, (4.3) the Apache Giraph and the algorithms which we developed using this system and (4.4) the performance comparison of the three systems.

### 4.1 SNAP

#### *Introduction*

Stanford Network Analysis Platform (SNAP) is a general purpose network analysis and graph mining library. It is written in C++ and easily scales to massive networks with hundreds of millions of nodes, and billions of edges. It efficiently manipulates large graphs, calculates structural properties, generates regular and random graphs, and supports attributes on nodes and edges. SNAP also follows a graph-centric programming model.

The SNAP library is being actively developed since 2004 by Stanford University and is organically growing as a result of their research pursuits in analysis of large social and information networks.

Snap.py is a Python interface for SNAP. Snap.py provides performance benefits of SNAP, combined with flexibility of Python. Most of the SNAP functionality is available via Snap.py in Python.

	User interaction	Python
	Snap.py	Python
	SNAP	C++
Solution	Fast Execution	Easy to use, interactive
C++	✓	
Python		✓
Snap.py (C++, Python)	✓	✓

The latest version of Snap.py is 1.2 (May 12, 2015). Packages for Mac OS X, Linux (as CentOS) and Windows 64-bit are available at Stanford's Snappy repository.

## Code examples

To use Snap.py in Python, import the snap module.

```
import snap
```

Basic types in SNAP are TInt, TFlt, and TStr, which in Snap.py are automatically converted to Python types int, float, and str, respectively. In general, there is no need to explicitly work with SNAP types in Snap.py because of the automatic conversion.

```
i = snap.TInt(10)
print i.Val
```

**snap.TInt(10):** Create an integer with value 10.

**i.Val:** Return the number of variable i.

Vectors are sequences of values of the same type. Existing vector values can be accessed or changed by their index in the sequence. New values can be added at the end of a vector. Vector types in Snap.py and SNAP use a naming convention of being named as <type\_name>, followed by V. For example, a vector of integers is named TIntV.

```
v = snap.TIntV()
v.Add(1)
v.Add(2)
v.Add(3)
print v.Len()
print v[2]
v.SetVal(2,6)
print v[2]
```

**snap.TIntV():** Create an empty vector of integers

**v.Add(1):** Add a value at the end of a vector

**v.Len():** Get the number of values in the vector

**v[2]:** Get a value at a specific vector location

**v.SetVal(2,6):** Change a value at a specific vector location

Pairs contain two values. Each value has its own type. Pair types in Snap.py and SNAP use a naming convention of being named as <type1><type2>, followed by Pr. For example, a pair of (integer, string) is named TIntStrPr. If <type1> and <type2> have the same type, only one type name might be used, such as TIntPr. Vector of pairs could exist. For example TIntPrV is a vector of (integer, integer) pairs.

```
p = snap.TIntStrPr(1, "one")
print p.GetVal1()
print p.GetVal2()
```

**snap.TIntStrPr(1, "one"):** Create a pair of an integer and a string

**p.GetVal1():** Print the first value

**p.GetVal2():** Print the second value

With LoadEdgeList (GraphType, InFNm, SrcColId, DstColId) command, SNAP loads a (directed, undirected or multi) graph from a text file with one edge per line.

GraphType indicates the type of the graph which loaded:

- PNGraph: Directed graph
- PNEANet: Multi graph
- PUNGraph: Undirected graph

InFNm is a whitespace separated file of several columns:

<source node id>      <destination node id>

SrcColId and DstColId must be provided to indicate which column gives the source and which column gives the destination of the edge.

```
Graph= snap.LoadEdgeList(snap.PUNGraph,"edges.txt",0,1)
Graph= snap.LoadEdgeList(snap.PNGraph,"edges.txt",0,1)
```

**snap.LoadEdgeList(snap.PUNGraph,"edges.txt",0,1):**

Load the edges of file 'edges.txt' as Undirected graph

**snap.LoadEdgeList(snap.PNGraph,"edges.txt",0,1):**

Load the edges of file 'edges.txt' as Directed graph

Nodes and edges are traversed with iterators. For example:

```
for NI in Graph.Nodes():
    print "nodeId: %d" % (NI.GetId())
for EI in Graph.Edges():
    print "edge (%d, %d)" % (EI.GetSrcNid(), EI.GetDstNid())
```

**Graph.Nodes():** Traverse all the nodes using a node iterator

**NI.GetId():** Return the id of the node

**Graph.Edges():** Traverse all the edges using an edge iterator

**EI.GetSrcNid():** Return the id of the source edge

**EI.GetDstNid():** Return the id of the destination edge

## 4.1.1 Degree Centrality

### *In-Degree centrality*

In-Degree is the total number of ties directed to the node.

SnapInDegreeCentrality.py

```
import snap

Graph = snap.LoadEdgeList(snap.PNGraph,"edges.txt",0,1)
for NI in Graph.Nodes():
    x = str(NI.GetId()) + '\t' + str(NI.GetInDeg())
    print x
```

**NI.GetInDeg():** Returns the In-Degree of a node

## *Out-Degree centrality*

Out-Degree is the total number of ties that the node directs to other nodes of network.

SnapOutDegreeCentrality.py

```
import snap

Graph = snap.LoadEdgeList(snap.PNGraph, "edges.txt", 0, 1)
for NI in Graph.Nodes():
    x = str(NI.GetId()) + '\t' + str(NI.GetOutDeg())
    print x
```

**NI.GetOutDeg():** Returns the Out-Degree of a node

## *Degree centrality*

Degree centrality defined as the number of links incident upon a node.

SnapDegreeCentrality.py

```
import snap

Graph = snap.LoadEdgeList(snap.PNGraph, "edges.txt", 0, 1)
for NI in Graph.Nodes():
    x = str(NI.GetId()) + '\t' + str(NI.GetOutDeg() + NI.GetInDeg())
    print x
```

## 4.1.2 Closeness Centrality

Closeness centrality is based on the length of the average shortest path between a vertex and all vertices in the network. The more central a node is the lower its total distance from all other nodes.

SnapClosenessCentrality.py

```
import snap

Graph = snap.LoadEdgeList(snap.PUNGraph, "edges.txt", 0, 1)
for NI in Graph.Nodes():
    closenessValue = snap.GetClosenessCentr(Graph, NI.GetId())
    x = str(NI.GetId()) + '\t' + str(closenessValue)
    print x
```

**snap.GetClosenessCentr(Graph, NI.GetId()):**

Returns the Closeness centrality of a node, given the Graph as first parameter and id of the node as second parameter

## 4.1.3 Betweenness Centrality

To compute the Betweenness centrality of a vertex, it is necessary to count the number of shortest paths that pass across the given vertex.

### SnapBetweennessCentrality.py

```
import snap

Graph = snap.LoadEdgeList(snap.PUNGraph, "edges.txt", 0, 1)
Nodes = snap.TIntFltH()
Edges = snap.TIntPrFltH()
snap.GetBetweennessCentr(Graph, Nodes, Edges, 1.0)
for node in Nodes:
    x = str(node) + "\t" + str(Nodes[node])
    print x
```

**snap.TIntFltH():** Hash table mapping node ids to their corresponding betweenness centrality values.

**snap.TIntPrFltH():** Hash table mapping edges (provided as pairs of node ids) to their corresponding betweenness centrality values.

**snap.GetBetweennessCentr(Graph, Nodes, Edges, 1.0):**

Returns the Betweenness centrality of every node in the network. The last parameter used as indicator of quality of the approximation. Value 1.0 gives the exact Betweenness centrality.

## 4.1.4 Eigenvector Centrality

This measure is called Eigenvector centrality and establishes that the importance of a vertex is determined by the importance of its neighbors.

### SnapEigenvectorCentrality.py

```
import snap

Graph = snap.LoadEdgeList(snap.PUNGraph, "edges.txt", 0, 1)
NIdEigenH = snap.TIntFltH()
snap.GetEigenvectorCentr(Graph, NIdEigenH)
for item in NIdEigenH:
    x = str(item) + "\t" + str(NIdEigenH[item])
    print x
```

**snap.TIntFltH():** Hash table mapping node ids to their corresponding betweenness centrality values.

**snap.GetEigenvectorCentr(Graph, NIdEigenH):**

Returns the Eigenvector centrality of every node in the network.

## 4.1.5 Clustering Coefficient

Clustering Coefficient is a measure of the degree to which nodes in a graph tend to cluster together. More particularly, clustering coefficient of a selected user is defined as the probability that two randomly selected neighbors are connected to each other.

### SnapClusteringCoefficient.py

```
import snap

Graph = snap.LoadEdgeList(snap.PUNGraph, "edges.txt", 0, 1)
NIdCCfH = snap.TIntFltH()
snap.GetNodeClustCf(Graph, NIdCCfH)
for NI in NIdCCfH:
    x = str(NI) + "\t" + str(NIdCCfH[NI])
    print x
```

#### **snap.TIntFltH():**

Hash table mapping node ids to their corresponding betweenness centrality values.

#### **snap.GetNodeClustCf(Graph, NIdCCfH):**

Returns the Clustering Coefficient of every node in the network.

## 4.1.6 K-Core

A K-Core of a graph  $G$  is a maximal connected subgraph of  $G$  in which all vertices have degree at least  $k$ . In the same way, it is one of the connected components of the subgraph of  $G$  formed by repeatedly deleting all vertices of degree less than  $k$ .

### SnapKCore.py

```
import snap

Graph = snap.LoadEdgeList(snap.PUNGraph, "edges.txt", 0, 1)
CoreIDSzV = snap.TIntPrV()
kValue = snap.GetKCoreNodes(Graph, CoreIDSzV)
for item in CoreIDSzV:
    x = str(item.GetVal1()) + "\t" + str(item.GetVal2())
    print x
```

#### **snap.TIntPrV():**

Vector of (int, int) pairs.

#### **snap.GetKCoreNodes(Graph, CoreIDSzV):**

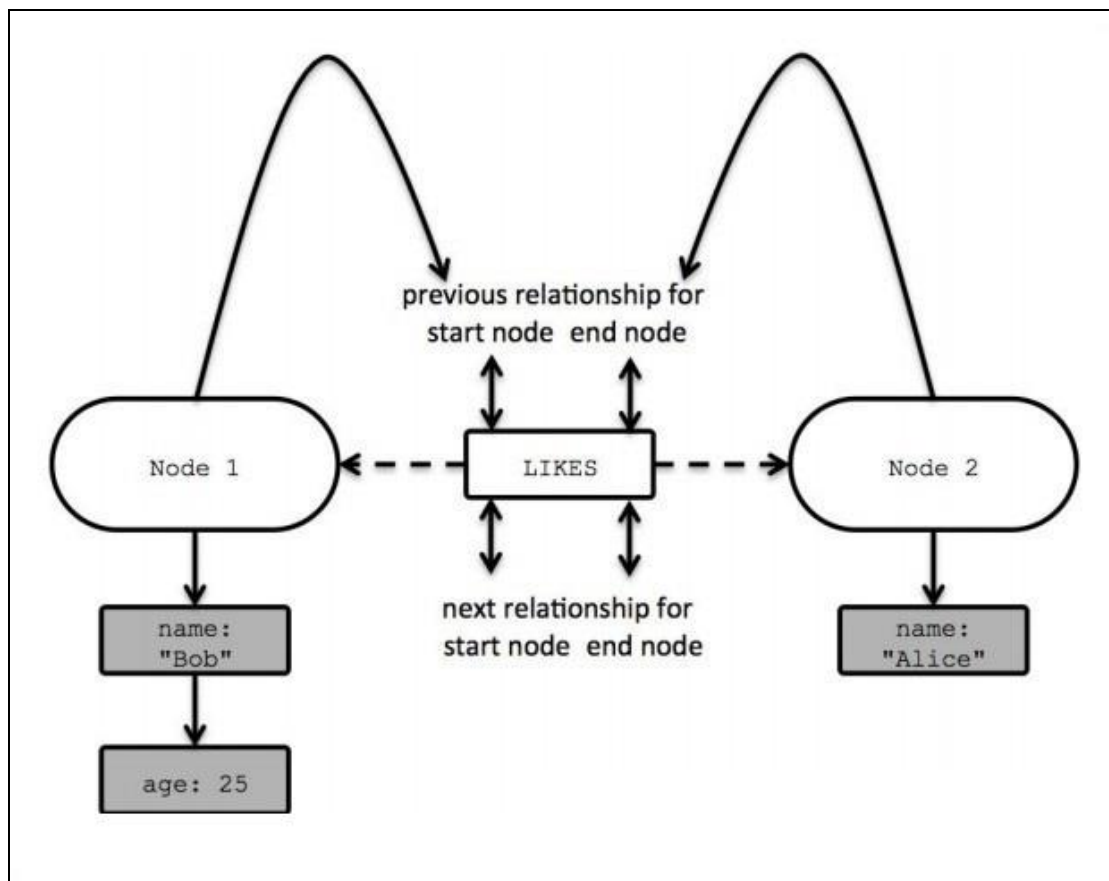
Returns the number of nodes for a given k-core.

## 4.2 Neo4j

### *Introduction*

Neo4j is an open-source graph database implemented in Java and it was first released in 2007. Neo4j is a well-known graph database system, currently it is the most popular. Neo4j provides two licenses: Enterprise and Community. The Enterprise edition offers incredible power and flexibility, with enterprise-grade availability, management and scale-up & scale-out capabilities. Neo4j Community edition is ideal for learning, and smaller do-it-yourself projects that require high levels of scaling. Excludes professional services and support. The current version is Neo4j 2.3.

Neo4j uses an intuitive graph-oriented model for data representation. Instead of static and rigid tables, it works with a flexible graph network consisting of nodes, relationships and properties. It is applicable for many web use cases such as tagging, social networks, real time recommendations and other network-shaped or hierarchical data sets.



Neo4j supports the ACID (Atomicity, Consistency, Isolation, Durability) transaction concept. Achieved by having in-memory transaction logs and a lock manager that applies locks on any altered database objects during the transaction. The changes in the log are flushed to disk after the successful completion of the transaction.

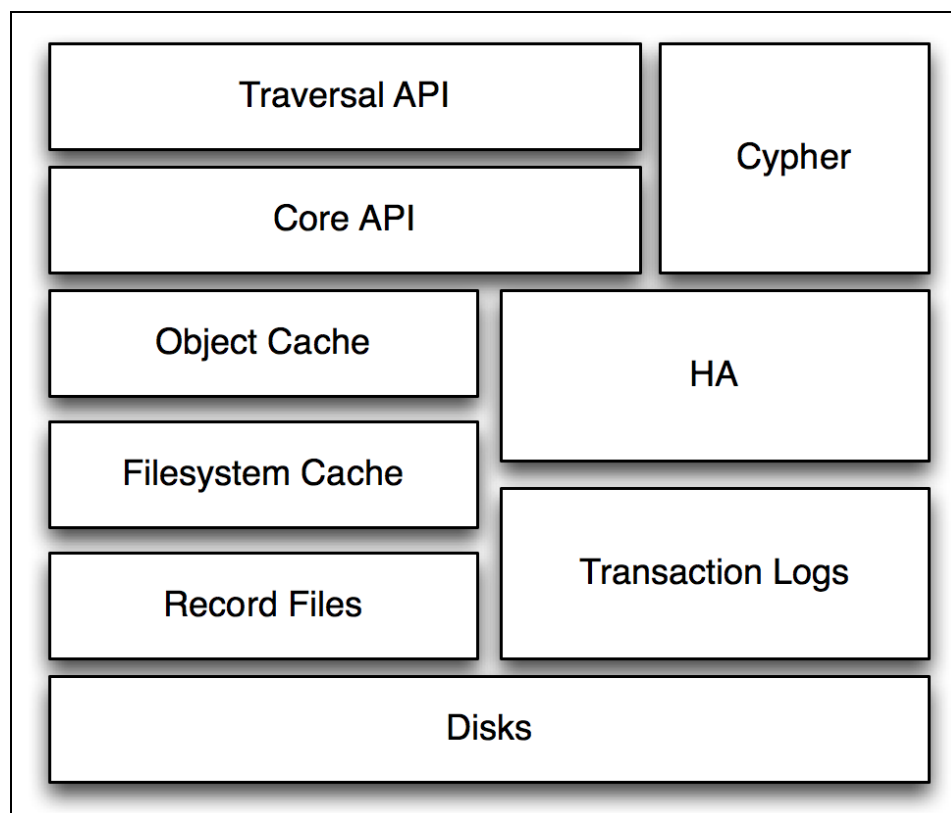
The default Neo4j Server has a powerful, customizable data visualization tool based on the built-in D3.js library wherewith it is very easy to style nodes and relationships setting different colors, sizes and titles on them.

Neo4j includes several ways to import data. For small or medium size CSV files it is easier to use Cypher Query language. It allows to load CSV data from any URL and creates simple or complex graph structures. For large CSV files which consists millions of nodes, the Batch Importer tool is capable of loading large amounts of data in Neo4j database very quickly. This tool supports only CSV files and the Data cannot be imported to an existing database. As input, the Batch Importer, takes a number of CSV files describing nodes and relationships and creates a graph database.

There are several ways of accessing the Neo4j database:

- Web UI
- Neo4j Shell
- Native Java API
  - Cypher Query Language
  - Core API
  - Traversal Framework
- REST API

A general view of Neo4j architecture



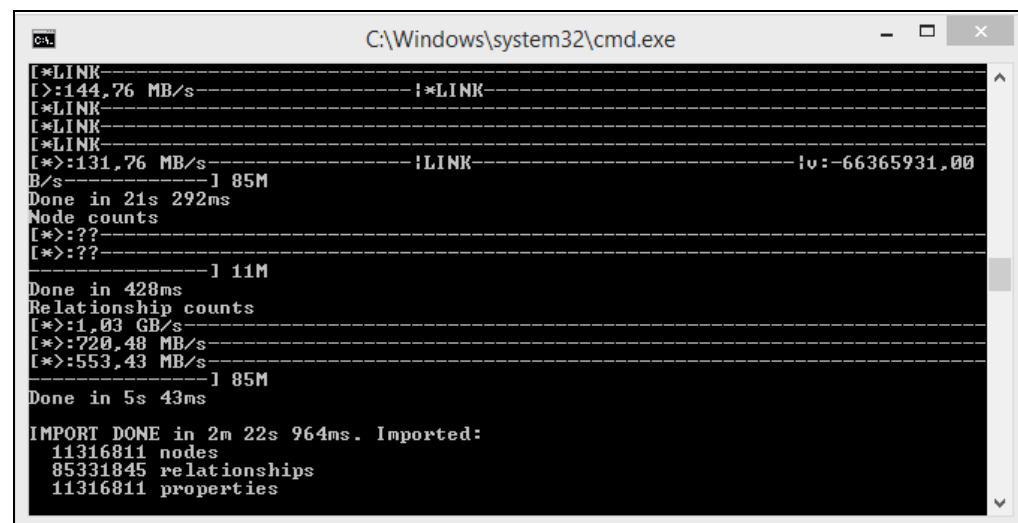


## Code examples

The following commands used for importing the CSV files of Twitter dataset into Neo4j named 'TwitterDB'

```
"C:\Program Files\Neo4j Community\bin\Neo4jImport.bat"  
--into C:\databases\TwitterDB  
--nodes "C:\datasets\Twitter\nodes.csv"  
--relationships "C:\datasets\Twitter\edges.csv"
```

After the execution of these commands a new Neo4j database was created with 11,316,811 nodes and 85,331,845 edges in 2 minutes and 23 seconds.



```
C:\Windows\system32\cmd.exe  
[*]LINK  
[>:144,76 MB/s-----!*LINK  
[*]LINK  
[*]LINK  
[*]LINK  
[>:131,76 MB/s-----!LINK  
B/s-----] 85M  
Done in 21s 292ms  
Node counts  
[*]:??  
[*]:??  
-----] 11M  
Done in 428ms  
Relationship counts  
[*]:1,03 GB/s-----  
[*]:720,48 MB/s-----  
[*]:553,43 MB/s-----] 85M  
Done in 5s 43ms  
IMPORT DONE in 2m 22s 964ms. Imported:  
11316811 nodes  
85331845 relationships  
11316811 properties
```

Find and show the total number of nodes in the graph using Cypher Query language.

```
MATCH (n)  
RETURN count(*) AS TotalNodes;
```

We present a complete example of how we can find and show the total number of nodes in the graph using Java API.

Open the database Test in folder Databases.

```
db = dbFactory.newEmbeddedDatabase("C:/Databases/Test");
```

Create a transaction with the selected database.

```
try(Transaction tx = db.beginTx())
```

A Java HashSet created. Every node in the graph added in the hash set 'allNodes'. The hash set size returns the total number of nodes in the graph.

```
Set<Node> allNodes = new HashSet<Node>();  
for (Node node : GlobalGraphOperations.at(db).getAllNodes()) {  
    allNodes.add (node);  
}  
System.out.println(allNodes.size());
```

Finally, we shut down the database server when the application finishes.

```
private static void shutDown() {
    Runtime.getRuntime().addShutdownHook( new Thread() {
        public void run() {
            db.shutdown();
        }
    });
}
```

The complete code of the previously explained example.

```
private static GraphDatabaseService db;
public static void main(String[] args) {
    GraphDatabaseFactory dbFactory = new GraphDatabaseFactory();
    db = dbFactory.newEmbeddedDatabase("C:/Databases/Test");
    try (Transaction tx = db.beginTx())
    {
        Set<Node> allNodes = new HashSet<Node>();
        for (Node node: GlobalGraphOperations.at(db).getAllNodes()) {
            allNodes.add(node);
        }
        System.out.println(allNodes.size());
        tx.success();
        shutDown();
    }
}
private static void shutDown() {
    Runtime.getRuntime().addShutdownHook( new Thread() {
        public void run() {
            db.shutdown();
        }
    });
}
public enum RelTypes implements RelationshipType {
    EDGE
}
```

## 4.2.1 Degree Centrality - Cypher

### *In-Degree centrality*

In-Degree is the total number of ties directed to the node.

#### Neo4jCypherInDegreeCentrality

```
start n = node(*)
WITH n, length (n<--()) as indegree
return n, indegree
```

### *Out-Degree centrality*

Out-Degree is the total number of ties that the node directs to other nodes of network.

#### Neo4jCypherOutDegreeCentrality

```
start n = node(*)
WITH n, length (n-->()) as outdegree
return n, outdegree
```

### *Degree centrality*

Degree centrality defined as the number of links incident upon a node.

#### Neo4jCypherDegreeCentrality

```
start n=node(*)
WITH n, length (n-->()) as outdegree, length(n<--()) as indegree
return n, outdegree+indegree AS degree
```

## 4.2.2 Closeness Centrality

Closeness centrality is based on the length of the average shortest path between a vertex and all vertices in the network. The more central a node is the lower its total distance from all other nodes.

Neo4jClosenessCentrality.java

```
private static GraphDatabaseService db;
public static void main(String[] args){
    GraphDatabaseFactory dbFactory = new GraphDatabaseFactory();
    db = dbFactory.newEmbeddedDatabase("C:/Databases/Test");
    try (Transaction tx = db.beginTx())
    {
        Set<Node> allNodes = new HashSet<Node>();
        for (Node node:
            GlobalGraphOperations.at(db).getAllNodes()) {
            allNodes.add(node);
        }
        SingleSourceShortestPath<Double>
        singleSourceShortestPath=getSingleSourceShortestPath();
        ClosenessCentrality<Double> closenessCentrality =
            new ClosenessCentrality<Double>(singleSourceShortestPath,
            new DoubleAdder(),0.0,allNodes,new CostDivider<Double>(){
                public Double divideByCost(Double d, Double c){
                    return d / c;
                }
                public Double divideCost(Double c, Double d){
                    return c / d;
                }
            });
        closenessCentrality.calculate();
        for (Node node:GlobalGraphOperations.at(db).getAllNodes()){
            Double value = closenessCentrality.getCentrality(node);
            System.out.println("Node:"+node.getId()+" ,value:"+value);
        }
        tx.success();
        shutdown();
    }
}

protected static SingleSourceShortestPath<Double>
getSingleSourceShortestPath()
{
    return new SingleSourceShortestPathDijkstra<Double>( 0.0, null,
        new CostEvaluator<Double>()
        {
            public Double getCost( Relationship relationship,
                Direction direction ){
                return 1.0;
            }
        }, new org.neo4j.graphalgo.impl.util.DoubleAdder(),
        new org.neo4j.graphalgo.impl.util.DoubleComparator(),
        Direction.BOTH, RelTypes.LINKS );
}
```

---

The second parameter is an object capable of adding distances. Needed since an "average" will be computed.

The third parameter is the starting value to the accumulator.

The fourth parameter is a set containing the nodes for which centrality values should be computed.

The fifth parameter is an object capable of inverting a distance.

The first parameter of the constructor of Closeness Centrality object calls the `getSingleSourceShortestPath()` method. This method returns a `SingleSourceShortestPath<Double>` object. This object encapsulates an algorithm able to solve the single source shortest path problem. It can find the shortest path(s) from a given start node to all other nodes in a network. To do this calls the above code.

```
return new SingleSourceShortestPathDijkstra<Double> ( 0.0, null, new
CostEvaluator<Double>() {
    public Double getCost( Relationship relationship,
                          Direction direction) {
        return 1.0;
    }
}, new org.neo4j.graphalgo.impl.util.DoubleAdder(),
new org.neo4j.graphalgo.impl.util.DoubleComparator(),
Direction.BOTH, RelTypes.LINKS);
```

The first parameter of this object is the starting cost for both the start node and the end node.

The second parameter is the starting node.

The third parameter is the cost function per relationship.

The `getCost (Relationship relationship, Direction direction)` is the general method for looking up costs for relationships.

The fourth parameter is adding up the path cost.

The fifth parameter is comparing to path costs.

The sixth parameter shows the relationship direction to follow.

The seventh parameter shows the kind of the relationship that should be included in the path.

## 4.2.3 Betweenness Centrality

To compute the Betweenness centrality of a vertex, it is necessary to count the number of shortest paths that pass across the given vertex.

### Neo4jBetweennessCentrality.java

```
private static GraphDatabaseService db;
public static void main(String[] args) {
    GraphDatabaseFactory dbFactory = new GraphDatabaseFactory();
    db = dbFactory.newEmbeddedDatabase("C:/Databases/Test");
    try (Transaction tx = db.beginTx())
    {
        Set<Node> allNodes = new HashSet<Node>();
        for (Node node:
            GlobalGraphOperations.at(db).getAllNodes()) {
            allNodes.add(node);
        }
        BetweennessCentrality<Double> betweennessCentrality =
            new BetweennessCentrality<Double>(
                getSingleSourceShortestPath(), allNodes);
        betweennessCentrality.calculate();
        for (Node node:
            GlobalGraphOperations.at(db).getAllNodes()) {
            Double value =
                betweennessCentrality.getCentrality(node);
            System.out.println("Node:" + node.getId()
                + ", value:" + value);
        }
        tx.success();
        shutdown();
    }
}

protected static SingleSourceShortestPath<Double>
getSingleSourceShortestPath() {
    return new SingleSourceShortestPathDijkstra<Double>( 0.0,
        null, new CostEvaluator<Double>() {
            public Double getCost( Relationship relationship,
                Direction direction ) {
                return 1.0;
            }
        }
    ), new org.neo4j.graphalgo.impl.util.DoubleAdder(),
    new org.neo4j.graphalgo.impl.util.DoubleComparator(),
    Direction.BOTH, RelTypes.LINKS );
}
```

The first parameter of the constructor of `BetweennessCentrality` object calls the `getSingleSourceShortestPath()` method. This method returns a `SingleSourceShortestPath<Double>` object. This object encapsulates an algorithm able to solve the single source shortest path problem. It can find the shortest path(s) from a given start node to all other nodes in a network.

The second parameter is a set containing the nodes for which centrality values should be computed.

## 4.2.4 Eigenvector Centrality

This measure is called Eigenvector centrality and establishes that the importance of a vertex is determined by the importance of its neighbors.

### Neo4jEigenvectorCentrality.java

```
public class Neo4jEigenvectorCentrality {
    private static GraphDatabaseService db;
    public static void main(String[] args) {
        GraphDatabaseFactory dbFactory = new GraphDatabaseFactory();
        db = dbFactory.newEmbeddedDatabase("C:/Databases/Test");
        try (Transaction tx = db.beginTx()) {
            EigenvectorCentralityPower eigen =
                new EigenvectorCentralityPower(
                    Direction.BOTH, (new CostEvaluator<Double>() {
                        public Double getCost(Relationship arg0, Direction arg1)
                        {
                            return 1.0;
                        }
                    }),
                    Sets.newHashSet(GlobalGraphOperations.
                        at(db).getAllNodes()),
                    Sets.newHashSet(GlobalGraphOperations.
                        at(db).getAllRelationships()), 0.01);
            eigen.calculate();
            for (Node node: GlobalGraphOperations.at(db).getAllNodes()) {
                Double value = eigen.getCentrality(node);
                System.out.println("NodeId:" + node.getId() +
                    ", value:" + value);
            }
            tx.success();
            shutdown();
        }
    }
    private static void shutdown() {
        Runtime.getRuntime().addShutdownHook( new Thread() {
            public void run() {
                db.shutdown();
            }
        });
    }
    public enum RelTypes implements RelationshipType {
        EDGE
    }
}
```

The first parameter of the constructor of EigenvectorCentralityPower object is the direction in which the paths should follow the relationships.

The second parameter is a CostEvaluator object. In order to make the solving of shortest path problems as general as possible, the algorithms accept objects handling all relevant tasks regarding costs of paths. This allows the user to represent the costs in any possible way, and to calculate them in any way. The `getCost (Relationship arg0, Direction arg1)` is the general method for looking up costs for relationships. This can do anything, like looking up a property or running some small calculation. In this example only returns the value 1.0 for every edge/relationship.

The third parameter is the set of nodes the calculation should be run on.

The fourth parameter is the set of relationships that should be processed.

The fifth parameter is the precision factor (ex. 0.01 for 1% error). This is not the error from the correct values, but the amount of change tolerated in one iteration.

```
EigenvectorCentralityPower eigen = new EigenvectorCentralityPower(
    Direction.BOTH, (new CostEvaluator<Double>() {
    public Double getCost(Relationship arg0, Direction arg1) {
        return 1.0;}}}
    ,
    Sets.newHashSet(GlobalGraphOperations.at(db).getAllNodes()),
    Sets.newHashSet(GlobalGraphOperations.at(db).getAllRelationships()),
    0.01);
```



## 4.2.5 Clustering Coefficient

Clustering Coefficient is a measure of the degree to which nodes in a graph tend to cluster together. More particularly, clustering coefficient of a selected user is defined as the probability that two randomly selected neighbors are connected to each other.

```
public class ClusteringCoefficient {
    private static GraphDatabaseService db;
    public static void main(String []args) {
        GraphDatabaseFactory dbFactory = new GraphDatabaseFactory();
        db = dbFactory.newEmbeddedDatabase("C:/Databases/Test");

        try (Transaction transaction = db.beginTx()) {
            for (Node node :
                GlobalGraphOperations.at(db).getAllNodes()) {
                double value = compute(node);
                System.out.println("NodeId: "+node.getId()+" ,value:"+
value);
            }
            transaction.success();
            shutdown();
        }

        private static double compute(Node node) {
            Set<Node> neighbours = new HashSet<>();
            for (Relationship edge :
                node.getRelationships(RelTypes.LINKS, Direction.BOTH)) {
                neighbours.add(edge.getOtherNode(node));
            }
            if (neighbours.size() <= 1) return 0.0;
            long numEdges = 0;
            for (Node neighbour : neighbours) {
                for (Relationship edge :
                    neighbour.getRelationships(RelTypes.LINKS, Direction.OUTGOING))
                {
                    if (neighbours.contains(edge.getOtherNode(neighbour))) {
                        numEdges++;
                    }
                }
            }

            long possibleEdges =
                (long)neighbours.size()*(neighbours.size()-1);
            return (double)numEdges / possibleEdges;
        }

        public enum RelTypes implements RelationshipType{
            LINKS
        }

        private static void shutdown() {
            Runtime.getRuntime().addShutdownHook( new Thread(){
                public void run() {
                    db.shutdown();
                }
            });
        }
    }
}
```

## 4.3 Apache Giraph

### *Introduction*

Apache Giraph is an iterative graph processing system built for high scalability. It is currently used at Facebook to analyze the social graph formed by users and their connections. Giraph originated as the open-source counterpart to Pregel, the graph processing architecture developed at Google. Both systems are inspired by the Bulk Synchronous Parallel model of distributed computation.

Giraph adds several features beyond the basic Pregel model:

- master computation
- shared aggregators
- edge-oriented input
- out-of-core computation

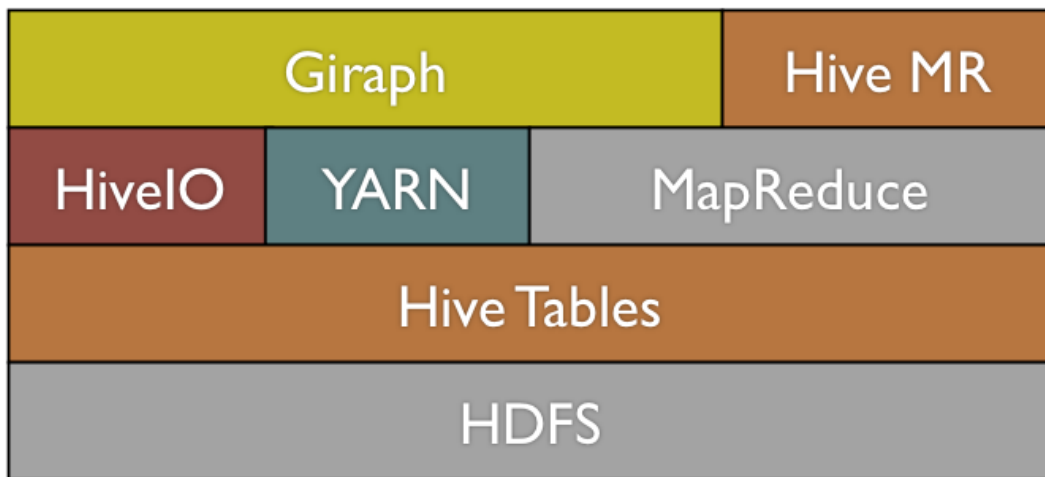
Apache Giraph releases:

- Giraph 0.1-incubating released in February 2012
- Giraph 1.0 released in January 2013
- Giraph 1.1 released in November 2014

Apache Giraph is part of the Hadoop ecosystem and runs on Hadoop. Apache Hadoop is an open-source software framework for distributed storage and distributed processing of very large data sets on computer clusters built from commodity hardware. All the modules in Hadoop are designed with a fundamental assumption that hardware failures are common and should be automatically handled by the framework.

The base of Apache Hadoop framework:

- Hadoop MapReduce: A programming model and a system for the processing of large datasets. It is based on scanning files in parallel across multiple machines, modelling data as keys and values. Giraph can run on Hadoop as a MapReduce job.
- Hadoop Distributed File System (HDFS): A distributed filesystem to store data across multiple machines. It provides high throughput and fault tolerance. Giraph can read data from and write data to HDFS
- Hadoop YARN: A resource-management platform responsible for managing computing resources in clusters and using them for scheduling of users' applications
- Apache Hive: A data warehouse for Hadoop which lets you express ad hoc queries and analytics for large data sets with a high-level programming language similar to SQL.



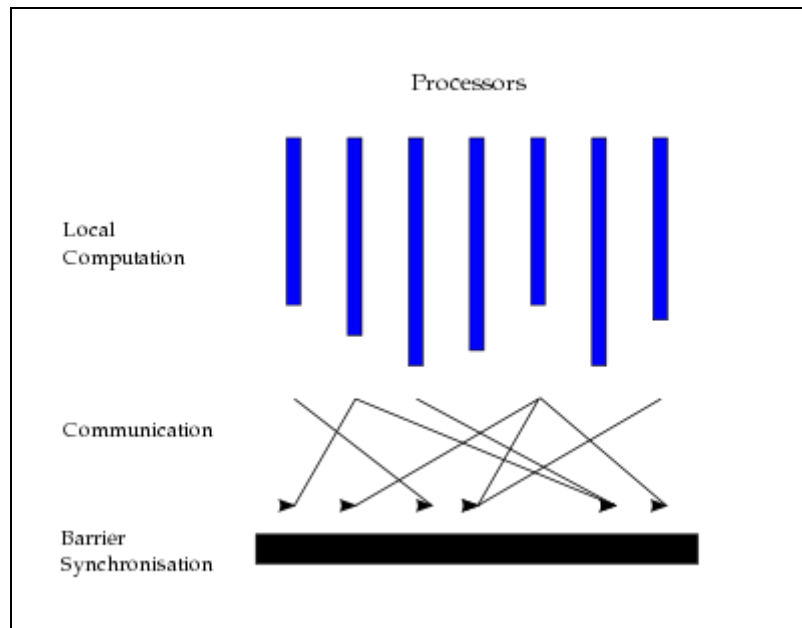
Apache Giraph uses the Bulk Synchronous Parallel (BSP) model. It is a bridging model for designing parallel algorithms. The BSP model was developed by Leslie Valiant of Harvard University during the 1980s.

A BSP computer consists of

- components capable of processing and/or local memory transactions
- a network that routes messages between pairs of such components
- a hardware facility that allows for the synchronization of the components

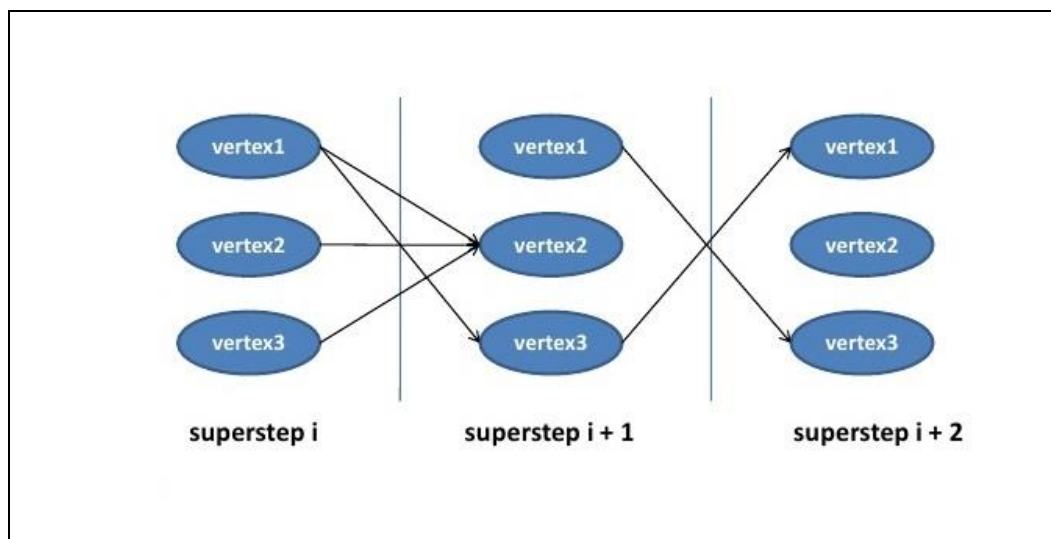
A BSP algorithm relies heavily on the third feature because a computation proceeds in a series of global supersteps, which consists of three components:

- Concurrent computation: every participating processor may perform local computations, i.e., each process can only make use of values stored in the local fast memory of the processor. The computations occur asynchronously of all the others but may overlap with communication.
- Communication: The processes exchange data between themselves to facilitate remote data storage capabilities.
- Barrier synchronization: When a process reaches this point (the barrier), it waits until all other processes have reached the same barrier.



### Vertex-Centric BSP:

- Each vertex has an id, a value, a list of its adjacent vertex ids and the corresponding edge values
- Each vertex is invoked in each superstep, can recompute its value and send messages to other vertices, which are delivered over superstep barriers



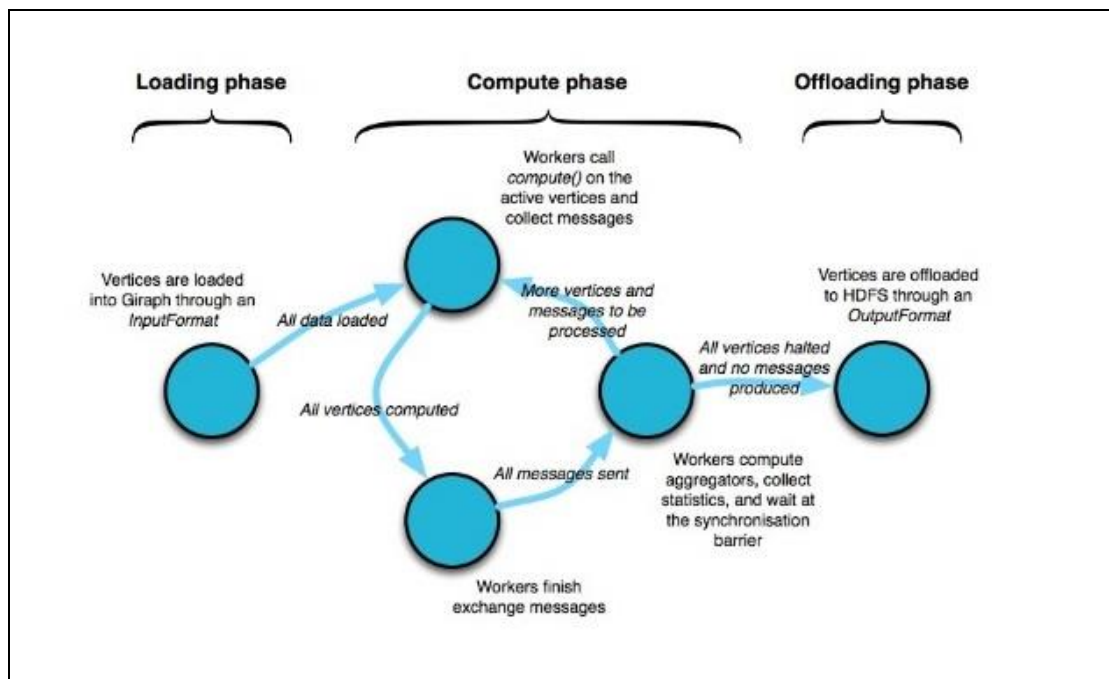
Apache Giraph Data model consists of the following:

1. A graph is a set of vertices and edges, and each vertex is defined as follows:
  - It has a unique ID, defined by a type (i.e., an integer)
  - It has a value, also defined by a type (i.e., a double)
  - It has a number of outgoing edges that point to other vertices
  - Each edge can have a value, also defined by a type (i.e., an integer)

2. The input data is modeled as directed graph and edges are assigned to their source vertex
3. Vertices are aware only of their outgoing edges

Apache Giraph job includes three phases:

- Loading phase: The graph loaded from the disk and each worker assigned with specific vertices. Giraph loads the graph using an Input format.
- Compute phase: In each superstep, the workers assign messages to vertices, iterate on active vertices, and call compute () method in each vertex.
- Offloading phase: Once the computation is done, the vertices are offloaded to HDFS using an Output format.



## Code examples

Apache Giraph includes some essential objects and interface instances for its functionality. Some of them presented in this section. This section also includes a complete example which explained in detail.

Each vertex object is an instance of the Vertex class. Using the interface of the Vertex class you can access the vertex value and the edges and their values and add and remove edges. Giraph has a default implementation of vertices and edges and it is not required these interfaces to be implemented by the user unless the user wants some specific behavior to extend.

The Edge class is simpler and it has only three methods: one to get the vertex ID of the other endpoint, one to get the value, and one to set the value.

## Vertex class

- function getId():

Returns the vertex ID, the type depends on the type of the ID.

- function getValue()

Returns the vertex value, the type depends on the type of the vertex value.

- function setValue(value)

Sets the vertex value, the type of the parameter depends on the type of the value.

- function getEdges()

Returns all the outgoing edges for the vertex.

- function getNumEdges()

Returns the total outgoing edges for the vertex.

- function getEdgeValue(targetId)

If the edge exists, returns the value of the first edge connecting to the target vertex.

- function setEdgeValue(targetId, value)

If the edge exists, sets the value of the first edge connecting to the target vertex.

- function getAllEdgeValues(targetId)

Returns the values of all the edges which is connected to a specific vertex.

- function voteToHalt()

Signaling the end of the computation for a given vertex.

- function addEdge(edge)

Adds an edge to the vertex which specified in the parameter.

- function removeEdges(targetId)

Removes all edges which is pointing the target vertex.

## Edge class

- function getTargetVertexId()

Returns the target vertex ID, the return type depends on the type of the ID.

- function getValue()

Returns the edge value, the return type depends on the type of the edge value.

- function setValue(value)

Sets the edge value, the type of the parameter depends on the type of the edge value.

## Example

The next example shows a complete implementation of one of the important algorithms in graph theory. It calculates the shortest paths in a graph using Dijkstra's algorithm. This algorithm, given a graph representation and a vertex, finds the shortest path between that vertex and every other one in the graph. It starts by assigning a value of 0 to a chosen vertex and a value of infinity to all the others. Then the algorithm proceed, in parallel, to calculate for every vertex a sum of vertex's value (the current shortest distance to it) plus the distance to all the vertex's neighbors. The algorithm converges when no more updates happen on the graph.

## SimpleShortestPathsComputation.java

```
import org.apache.giraph.graph.BasicComputation;
import org.apache.giraph.Algorithm;
import org.apache.giraph.conf.LongConfOption;
import org.apache.giraph.edge.Edge;
import org.apache.giraph.graph.Vertex;
import org.apache.hadoop.io.DoubleWritable;
import org.apache.hadoop.io.FloatWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.log4j.Logger;
import java.io.IOException;

public class SimpleShortestPathsComputation extends BasicComputation<
    LongWritable, DoubleWritable, FloatWritable, DoubleWritable> {
    public static final LongConfOption SOURCE_ID =
        new LongConfOption("SimpleShortestPathsVertex.sourceId", 1,
            "The shortest paths id");
    private static final Logger LOG =
        Logger.getLogger(SimpleShortestPathsComputation.class);
    private boolean isSource(Vertex<LongWritable, ?, ?> vertex) {
        return vertex.getId().get() == SOURCE_ID.get(getConf());
    }

    @Override
    public void compute(
        Vertex<LongWritable, DoubleWritable, FloatWritable> vertex,
        Iterable<DoubleWritable> messages) throws IOException {
        if (getSuperstep() == 0) {
            vertex.setValue(new DoubleWritable(Double.MAX_VALUE));
        }
        double minDist = isSource(vertex) ? 0d : Double.MAX_VALUE;
        for (DoubleWritable message : messages) {
            minDist = Math.min(minDist, message.get());
        }
        if (LOG.isDebugEnabled()) {
            LOG.debug("Vertex " + vertex.getId() + " got minDist = " +
minDist +
                " vertex value = " + vertex.getValue());
        }
        if (minDist < vertex.getValue().get()) {
            vertex.setValue(new DoubleWritable(minDist));
            for (Edge<LongWritable, FloatWritable> edge :
vertex.getEdges()) {
                double distance = minDist + edge.getValue().get();
                if (LOG.isDebugEnabled()) {
                    LOG.debug("Vertex " + vertex.getId() + " sent to " +
edge.getTargetVertexId() + " = " + distance);
                }
            }
            sendMessage(edge.getTargetVertexId(), new DoubleWritable(distance));
        }
        vertex.voteToHalt();
    }
}
```

The above code operates on a graph with every vertex having a LongWritable ID and a DoubleWritable distance to the source vertex as its vertex data. All the edges have FloatWritable labels associated with them, representing the distance between two connected vertices in the graph. A source vertex is given to the implementation via the SimpleShortestPathsVertex.sourceId command-line option and expected to be a LongWritable ID. Then, the algorithm measures the distance from this source vertex to all the other vertices in the graph. Initially it assigns a distance metric of Double.MAX\_VALUE, which is a representation of the infinity, to all the vertices except the source vertex. Every vertex waits to receive messages from the neighbors it is connected to, propagating the distance from the source vertex to each neighbor plus the distance between a neighbor vertex and a vertex that is receiving. The next step is to pick the smallest value, and if it happens to be smaller than the current distance, it assumes that a shorter path through one of the neighbors was found. Then it is send messages to the neighbors announcing that fact. Finally, a call to voteToHalt() Giraph method guarantees that the computation continues as long as there are unprocessed messages. When there are no messages left, it means every vertex has the shortest distance and all that is left is to record the state of the graph.

In Apache Giraph all the computation taking place in the compute() method that is called for every vertex, at least once initially at Superstep 0 and then for as long as there are messages sent to the vertex. It is necessary to define the compute() method, so to the next sections, we only describe the implementation of this method which is the core function of the Apache Giraph computations.



### 4.3.1 Degree Centrality

Degree centrality defined as the number of links incident upon a node.

#### DegreeCentrality.java

```
@Override
public void compute(Vertex<LongWritable, DoubleWritable,
FloatWritable> vertex, Iterable<DoubleWritable> messages)
throws IOException {
    // TODO Auto-generated method stub
    if (getSuperstep() == 0) {
        Iterable<Edge<LongWritable, FloatWritable>> edges =
            vertex.getEdges();
        for (Edge<LongWritable, FloatWritable> edge : edges) {
            sendMessage(edge.getTargetVertexId(), new
                DoubleWritable(1.0));
        }
    } else {
        long sum = 0;
        for (DoubleWritable message : messages) {
            sum++;
        }
        DoubleWritable vertexValue = vertex.getValue();
        vertexValue.set(sum + vertex.getNumEdges());
        vertex.setValue(vertexValue);
        vertex.voteToHalt();
    }
}
```

At Superstep 0, for every vertex we get the edges of the Vertex as an Iterable Object. Then every vertex for every edge which connected with, send a message. The first parameter of the message is the destination ID, the second parameter is a value. To calculate the in-Degree, we send only the value 1.0 in the message.

```
if (getSuperstep() == 0) {
    Iterable<Edge<LongWritable, FloatWritable>> edges =
        vertex.getEdges();
    for (Edge<LongWritable, FloatWritable> edge : edges) {
        sendMessage(edge.getTargetVertexId(), new
            DoubleWritable(1.0));
    }
}
```

At the next Superstep, every vertex counts in sum variable the messages which it receives.

```
else {
    long sum = 0;
    for (DoubleWritable message : messages) {
        sum++;
    }
}
```

Finally, every vertex sets the vertex value as the sum of the incoming messages and its total outgoing edges, and this summation calculates the Degree centrality. Then, every vertex calls the `voteToHalt()` method and signaling that it finishes the computation. When all vertices are complete their computation and no other message is sent through the network, the Degree centrality value of every vertex has been calculated.

```
DoubleWritable vertexValue = vertex.getValue();  
vertexValue.set(sum + vertex.getNumEdges());  
vertex.setValue(vertexValue);  
vertex.voteToHalt();
```

### 4.3.2 Closeness Centrality

Closeness centrality is based on the length of the average shortest path between a vertex and all vertices in the network. The more central a node is the lower its total distance from all other nodes.

In this section we split in parts the compute() function in order to improve the code readability.

```
public void compute(Vertex<LongWritable, VertexData, FloatWritable>
vertex, Iterable<Message> messages) throws IOException {
    if (getSuperstep() == 0) {
        closeness = new HashMap<Long, Double>();
        for (long i = START_ID; i < GRAPH_SIZE; i++) {
            if (i == vertex.getId().get()) {
                closeness.put(i, -1.0);
            } else {
                closeness.put(i, Double.MAX_VALUE);
            }
        }
        vertex.setValue(new VertexData(closeness));
        ArrayList<Long> vertexIds = new ArrayList<Long>();
        vertexIds.add(vertex.getId().get());
        Iterable<Edge<LongWritable, FloatWritable>> edges =
            vertex.getEdges();
        for (Edge<LongWritable, FloatWritable> edge : edges) {
            sendMessage(edge.getTargetVertexId(),
                new Message(1.0, vertexIds));
        }
    }
}
```

In Superstep 0, we create a Java HashMap with all the vertex IDs of the graph as key and as value we set the maximum number in Java. Each vertex has a HashMap with all the vertex IDs and this map records the distance between this vertex to each other in the graph. The variable GRAPH\_SIZE depicts the size of the graph which analyzed and must be specified before the algorithm starts. If the value for a vertex in this HashMap is equal to Double.MAX\_VALUE, then the current vertex has not discover this vertex yet. For the current vertex, we set a default value equal to -1. Then, the vertex set its value equal to the HashMap. By default, the vertex can store a double value. We extend this functionality and create the class VertexData [Appendix A.1] in order to store a value with HashMap type instead of type Double. We then send a message to all the vertices with which the current vertex is connected. The message includes the value 1.0, because the vertices have distance equal to 1, and a list with IDs. In Superstep 0, this list includes only its vertex value. By default the sendMessage() function take as first parameter the ID of the destination vertex and as second value a Double number. We extend this functionality using the class Message [Appendix A.2]. This class takes as first parameter a double value which is the distance between the source vertex and the destination vertex, and a list with ID values, which the destination vertex will update their values in its distance HashMap.

We explain the rest of compute's function code in three parts for better code readability.

```
else{
    vertexIds.clear();
    closeness = vertex.getValue().getCloseness();
    double value=0;
    ArrayList<Long> exist = new ArrayList<Long>();
    Iterable<Edge<LongWritable, FloatWritable>> edges =
        vertex.getEdges();
    for (Message message : messages) {
        value = message.getValue();
        for(int i=0;i<message.getIds().size();i++){
            if(closeness.get(message.getIds().get(i)) ==
                Double.MAX_VALUE
                && !exist.contains(message.getIds().get(i))) {
                vertexIds.add(message.getIds().get(i));
                exist.add(message.getIds().get(i));
            }
        }
    }
    if(vertexIds.size()>0){
        for (Edge<LongWritable, FloatWritable> edge : edges) {
            sendMessage(edge.getTargetVertexId(),
                new Message (value + 1.0,vertexIds));
        }
    }
}
```

At first, the vertex receives messages from the other vertices which it connects with. The value of every message which it receives is the same. Then, the current node creates a list with unique node IDs, if the value of the ID in distance HashMap is equal to Double.MAX\_VALUE, and send a message with these ID values to its neighbors in order to update their distance HashMap. At every round of messages the value of the messages increase by one. If this list is empty, no message is send from the vertex. If none of the vertices of the graph send a new message then the algorithm stops.

```
for (Message message : messages) {
    for(int i=0;i<message.getIds().size();i++){
        if(closeness.get(message.getIds().get(i))>message.getValue()){
            closeness.put(message.getIds().get(i),message.getValue());
        }
    }
}
```

In this part of the code, the current vertex updates the distance HashMap, if the received value for the specific ID value is less than the current value for this ID in the distance HashMap. Initially, the distance values are specified to Double.MAX\_VALUE. If the condition is true, then the distance map for this vertex is updated.

```

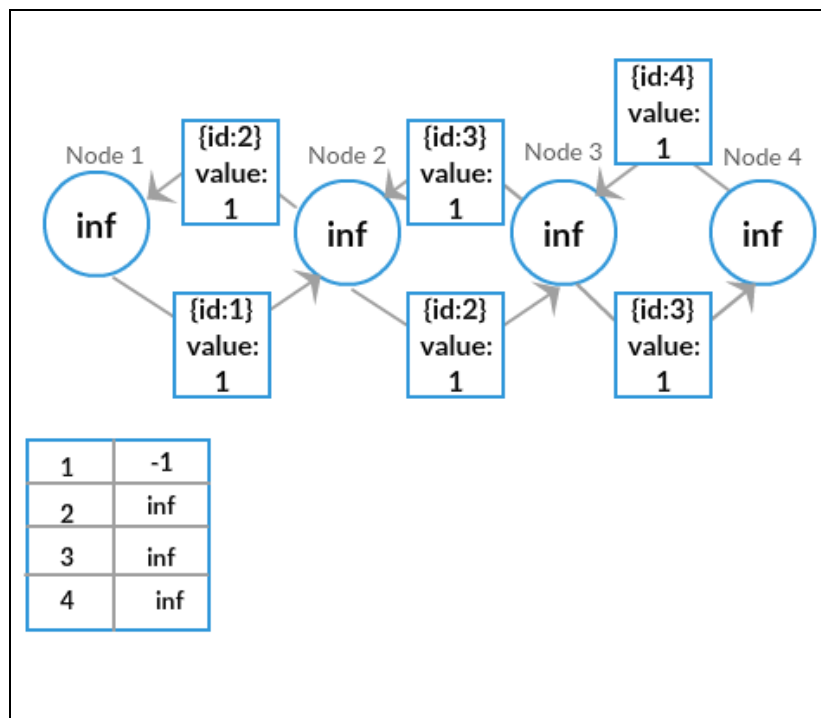
double counter = 0.0;
for (Map.Entry<Long,Double> entry : closeness.entrySet()){
    if(entry.getValue() != Double.MAX_VALUE
        && entry.getKey() != vertex.getId().get()){
        counter = counter + entry.getValue();
    }
}
double closeness_value = round((double)((GRAPH_SIZE-1)/counter),2);
closeness.put(GRAPH_SIZE, closeness_value);
vertex.setValue(new VertexData(closeness));
vertex.voteToHalt();

```

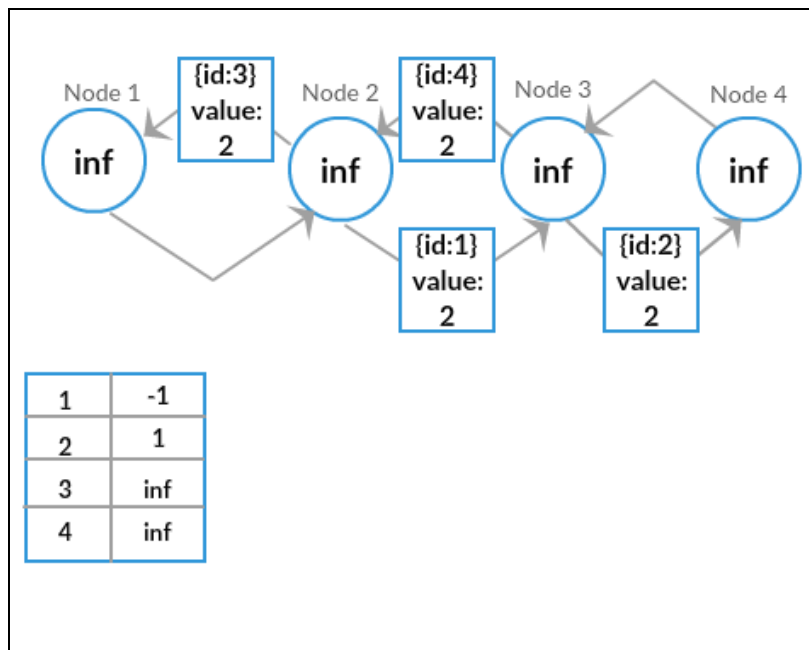
In the final part of the compute() function, the current vertex counts the distances from the other vertices in the network, if the value of the distance is not equal to Double.MAX\_VALUE. Subsequently, the vertex calculates its current Closeness centrality value and stores it in the last position of the distance HashMap. Then, the vertex updates its vertex value, with the updated distance HashMap and calls the voteToHalt() function. This function signaling that the current vertex completes its computation.

Next, we give a simple example of how the algorithm works in order to fully clarify the steps of the Closeness centrality algorithm in Apache Giraph. In this example, we only present the distance map of Node 1.

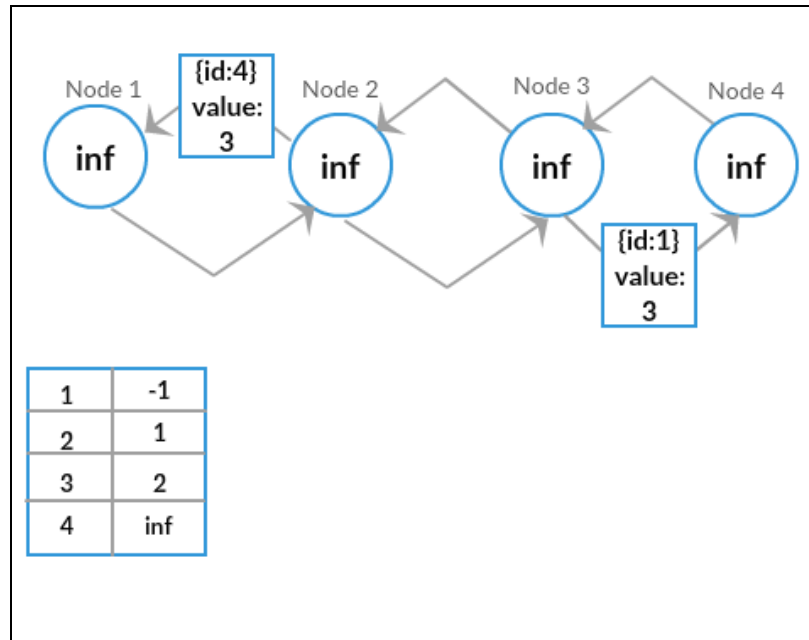
In Superstep 0, the value of each vertex in the distance map, excluding current vertex, is set to infinite. Every vertex has a distance map with the distances to all other nodes. Each vertex send a message to its neighbors with its id and value 1 which represents the distance between the two nodes, then every vertex votes to halt and signaling that they complete their computation.



In Superstep 1, each vertex sends a message to its neighbors with the ID values which receive from the previous Superstep from its neighbors and increase the message value by one. It is also updates the distance map for every ID with value less than the received and finally calculate the current Closeness centrality value. The vertices vote to halt.

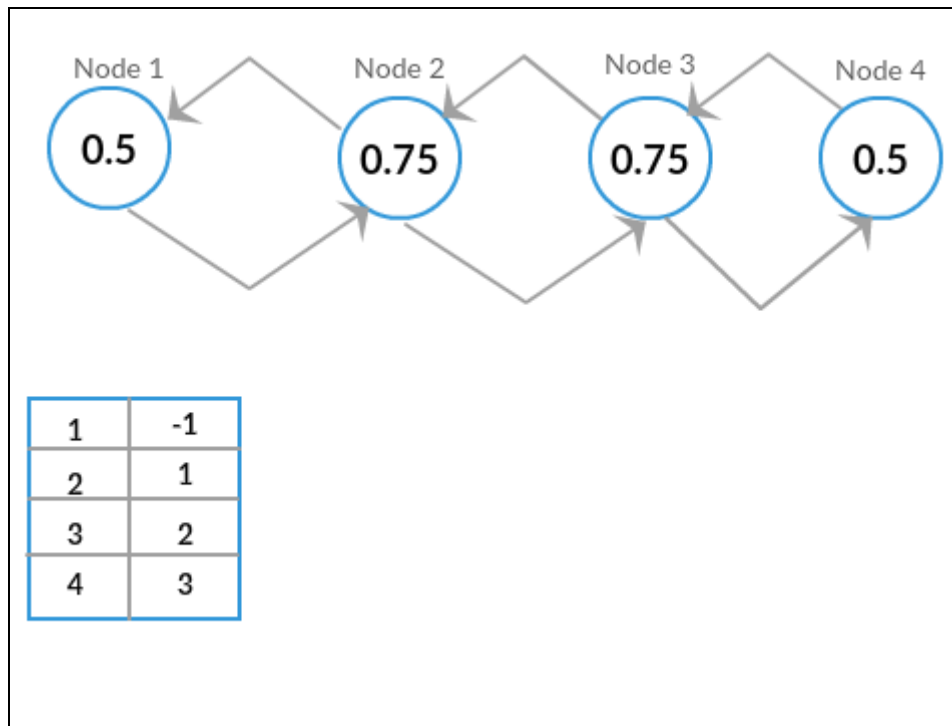


In the next Supersteps, the below functionality continues, and the distance map of every vertex updated.



In the final Superstep (in this example is Superstep 3), none of the nodes send a message. Every vertex calculate the Closeness centrality value and calls the vote to halt method. Because no message send through the network, in the last Superstep, the state

of every vertex in the graph become inactive, and the Closeness centrality calculation ends.



### 4.3.3 Eigenvector Centrality

This measure is called Eigenvector centrality and establishes that the importance of a vertex is determined by the importance of its neighbors.

#### EigenvectorCentrality.java

```
@Override
public void compute(
Vertex<LongWritable, DoubleWritable, FloatWritable> vertex,
Iterable<DoubleWritable> messages) throws IOException {
    if (getSuperstep() >= 1) {
        double sum = 0;
        for (DoubleWritable message : messages) {
            sum += message.get();
        }
        DoubleWritable vertexValue =
            new DoubleWritable((0.15f / getTotalNumVertices()) +
                0.85f * sum);
        vertex.setValue(vertexValue);
        aggregate(MAX_AGG, vertexValue);
        aggregate(MIN_AGG, vertexValue);
        aggregate(SUM_AGG, new LongWritable(1));
    } else {
        vertex.setValue(1 / getTotalNumVertices());
    }

    if (getSuperstep() < MAX_SUPERSTEPS) {
        long edges = vertex.getNumEdges();
        sendMessageToAllEdges(vertex,
            new DoubleWritable(vertex.getValue().get() / edges));
    } else {
        vertex.voteToHalt();
    }
}
```

The calculation of the Eigenvector centrality it is similar to PageRank calculation. The PageRank is a recursive process. The PageRank of each vertex depends on the PageRank of the incoming edges. Every vertex PageRank depends on that of its neighbors.

At Superstep 0, every vertex value set with the same initial value, 1 divided by the number of vertices in the graph.

```
else {
    vertex.setValue(1 / getTotalNumVertices());
}
```

At every Superstep next to 0 and until the algorithm completed, every vertex calculated a new value which is its current PageRank value. Each vertex computes its PageRank based on the incoming messages and the number of vertices and the constant  $d$  which is the dumping factor. The dumping factor is set with the value 0.85. The computation is thus completely independent for each vertex.



The `aggregate()` function enable global computation in the application. It can be used to check whether a global condition is satisfied. During a superstep, vertices provide values to aggregators. These values get aggregated by the system and the results become available to all vertices in the following superstep.

```

    if (getSuperstep() >= 1) {
        double sum = 0;
        for (DoubleWritable message : messages) {
            sum += message.get();
        }
        DoubleWritable vertexValue =
            new DoubleWritable((0.15f / getTotalNumVertices()) +
                               0.85f * sum);
        vertex.setValue(vertexValue);
        aggregate(MAX_AGG, vertexValue);
        aggregate(MIN_AGG, vertexValue);
        aggregate(SUM_AGG, new LongWritable(1));
    }

```

As long as the number of Supersteps they have not reach the maximum value (the number defined at the beginning of the Eigenvector centrality computation) every vertex send a message to all the vertices which it is connected to. The message value is the vertex current value divided by the total number of edges of the vertex.

```

if (getSuperstep() < MAX_SUPERSTEPS) {
    long edges = vertex.getNumEdges();
    sendMessageToAllEdges(vertex,
        new DoubleWritable(vertex.getValue().get() / edges));
}

```

Ideally the algorithm would complete at convergence. Because convergence is reached only asymptotically, PageRank is usually computed for a fixed number of iterations. The halting condition is based on this number.

```

else {
    vertex.voteToHalt();
}

```

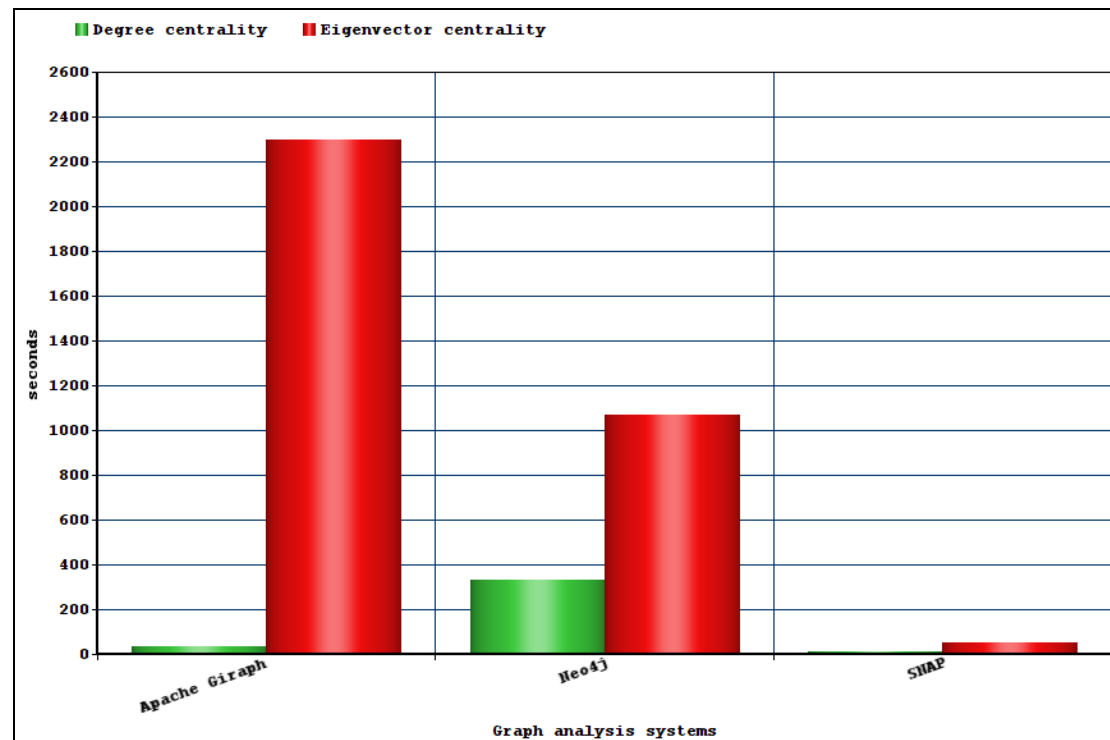
## 4.4 Experimental Comparison

In this chapter, we present and analyze our experimental results. In our experiments we use a single computer with a 4-core Intel i-5 processor at 3.2 GHz and 16 GB of RAM memory. We use six datasets which is DBLP, Wiki-Talk, LiveJournal, Twitter, Amazon product network and Email Enron. We performed experiments in three graph processing systems (Neo4j, SNAP and Apache Giraph) and for each dataset we execute the algorithms of Degree centrality, Closeness centrality, Betweenness centrality, Eigenvector centrality.

### 4.4.1 DBLP

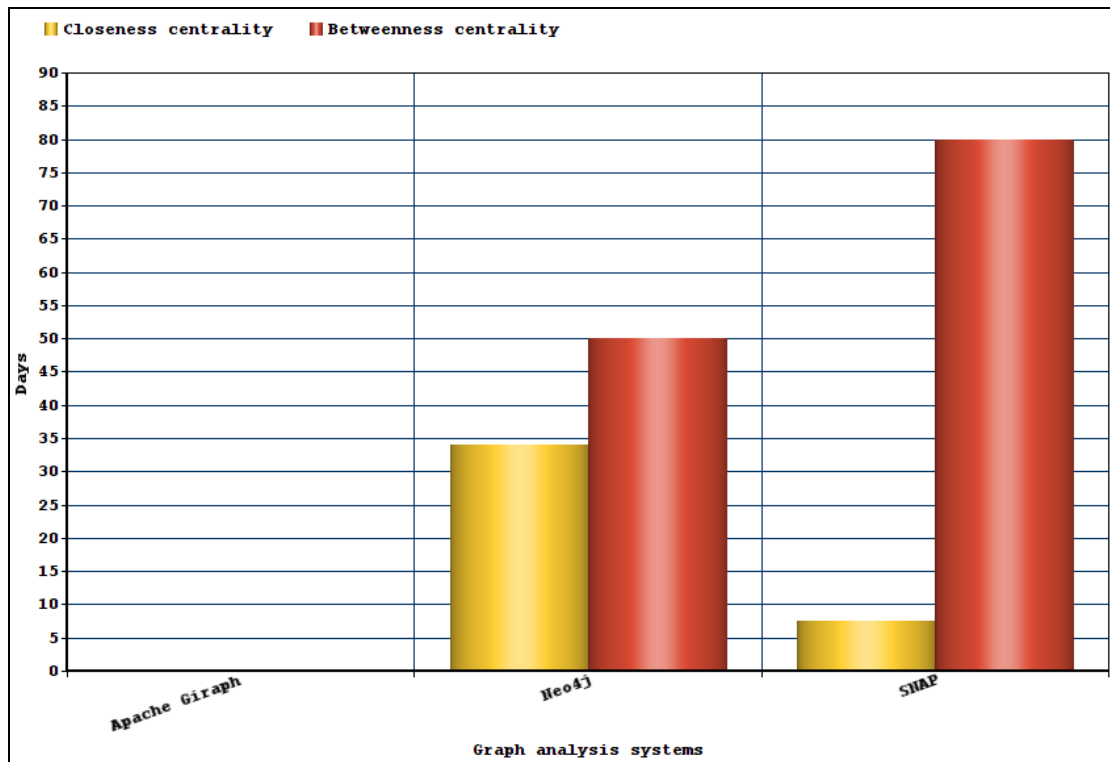
For this dataset, we extracted the nodes and the edges from DBLP database using the collaboration relationships among the authors of the scientific publications which exist in this database. The total number of nodes is 997,525 and also extracted 5,429,755 edges.

The following chart displays the running times in seconds for Degree centrality and Eigenvector centrality.



As we can observe the Stanford's Network Analysis Platform (SNAP) has the best performance in both algorithms. It took 10 seconds to calculate Degree centrality and 54 seconds for Eigenvector centrality. Besides SNAP, Apache Giraph has better performance from Neo4j for the Degree centrality but Neo4j it is much faster from Apache Giraph in Eigenvector calculation.

The following chart displays the running times in days for Closeness centrality and Betweenness centrality.



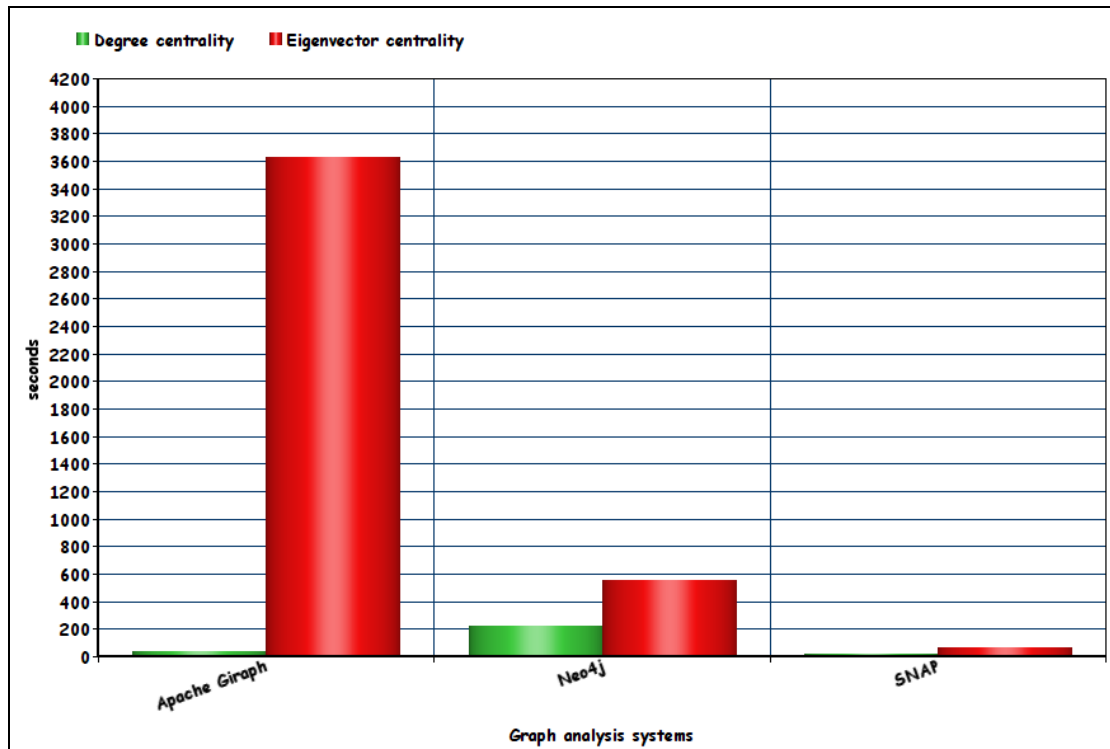
Closeness centrality on SNAP took 7.5 days to complete and executed faster than Neo4j. Neo4j performed better in calculation of Betweenness centrality because it executes calculations in parallel threads using a parallel Betweenness centrality algorithm.

The calculation of Closeness centrality and Betweenness centrality is not possible to execute in Apache Giraph using a single computer due to expensive calculations of these two centrality algorithms.

## 4.4.2 Wiki-Talk

This dataset acquired from Stanford's Large Network Dataset Collection repository [14]. The network contains all the users and discussion from the inception of Wikipedia till January 2008. Nodes in the network represent Wikipedia users and a directed edge represents that a user at least once edited a talk page of the other user. The total number of nodes is 2,394,385 and also extracted 5,021,410 edges.

The following chart displays the running times in seconds for Degree centrality and Eigenvector centrality.

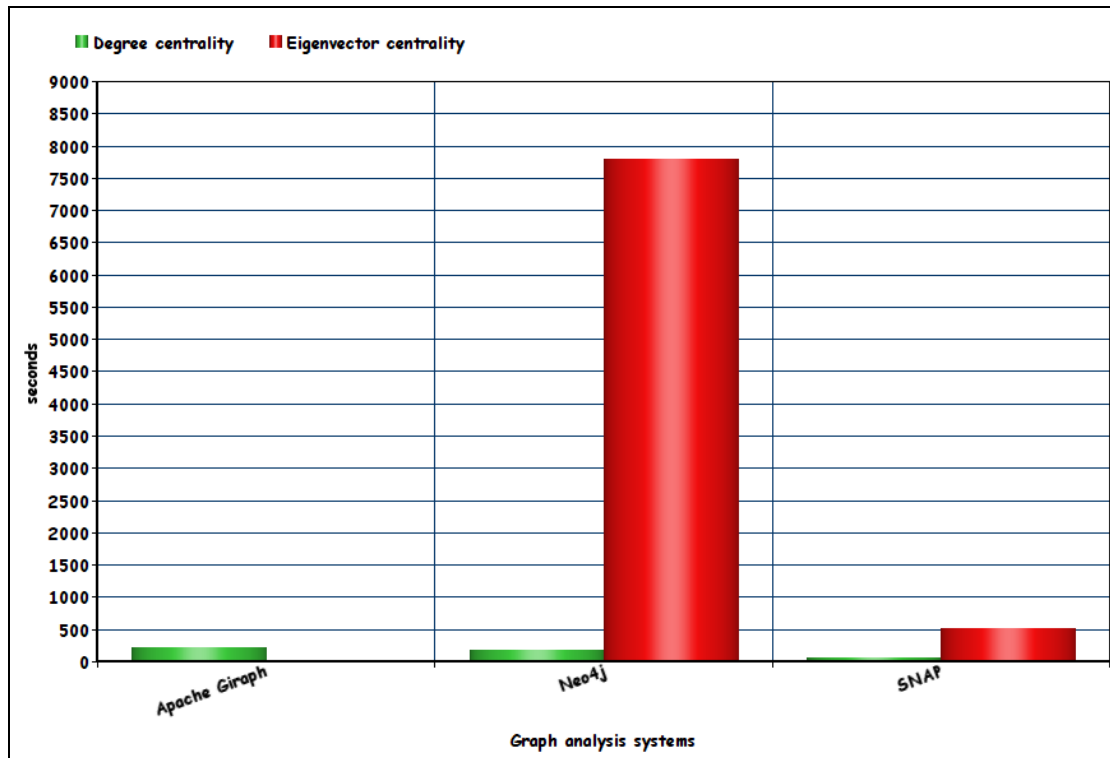


As we can observe the Stanford's Network Analysis Platform (SNAP) has the best performance in both algorithms. It took 17.8 seconds to calculate Degree centrality and 61 seconds for Eigenvector centrality. As in the previous experiment on DBLP dataset, Apache Giraph has better performance from Neo4j for the Degree centrality but Neo4j it is much faster from Apache Giraph in Eigenvector calculation. The calculation of Closeness centrality and Betweenness centrality it is not possible to run in any of the systems due to the size of the dataset and the available RAM memory.

### 4.4.3 LiveJournal

This dataset acquired from Stanford's Large Network Dataset Collection repository [14]. LiveJournal is a free on-line community which allows members to maintain journals, individual and group blogs, and it is also allows people to declare which other members are their friends they belong. The total number of nodes is 3,997,962 and also extracted 34,681,189 edges.

The following chart displays the running times in seconds for Degree centrality and Eigenvector centrality.



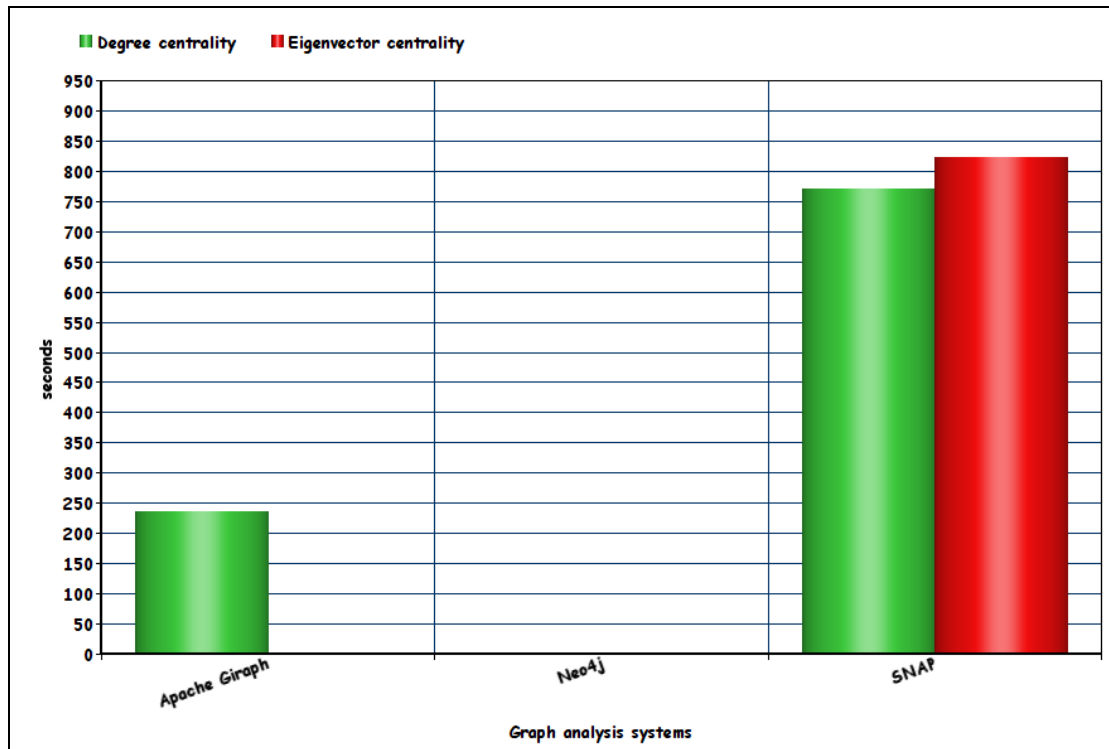
Stanford's Network Analysis Platform (SNAP) has the best performance in both algorithms. It took 49.8 seconds to calculate Degree centrality and 8 minutes 39 seconds for Eigenvector centrality. Neo4j executes Degree centrality algorithm in 2 minutes 50 seconds and Eigenvector centrality in 2 hours 11 minutes. Apache Giraph needs 3 minutes and 33 seconds to calculate Degree centrality and it was not possible to calculate Eigenvector centrality due to memory issues.

The calculation of Closeness centrality and Betweenness centrality it is not possible to run in any of the systems due to the size of the dataset and the available RAM memory.

## 4.4.4 Twitter

This dataset acquired from Arizona State University repository [15]. Twitter is a social news website. It can be viewed as a hybrid of email, instant messaging and sms messaging all rolled into one neat and simple package. It's a new and easy way to discover the latest news related to subjects you care about. The total number of nodes is 11,316,811 and also extracted 85,331,845 edges.

The following chart displays the running times in seconds for Degree centrality and Eigenvector centrality.



Apache Giraph has the best performance in Degree centrality. It completes the calculation in 3 minutes and 54 seconds. It was not possible to calculate Eigenvector centrality due to memory issues. SNAP calculates the Degree centrality in 12 minutes 50 seconds and Eigenvector centrality in 13 minutes and 43 seconds.

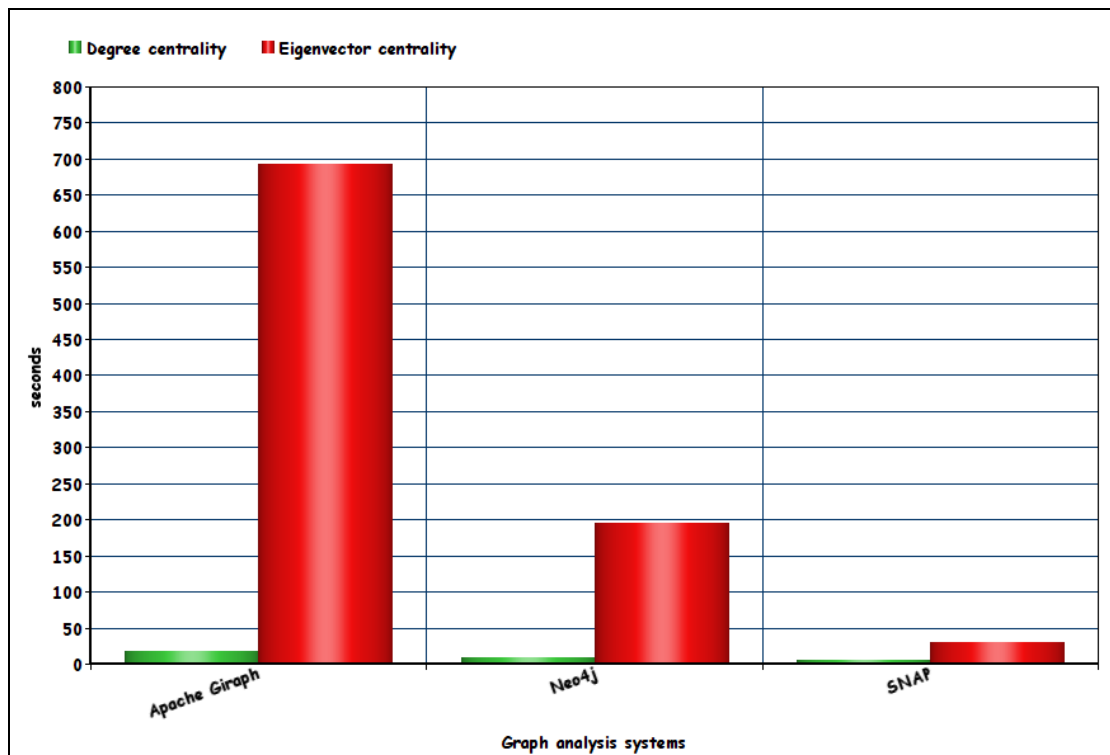
The calculation of Degree and Eigenvector centrality it is infeasible due to memory issues in Neo4j.

The calculation of Closeness centrality and Betweenness centrality it is not possible to run in any of the systems due to the size of the dataset and the available RAM memory.

### 4.4.5 Amazon product network

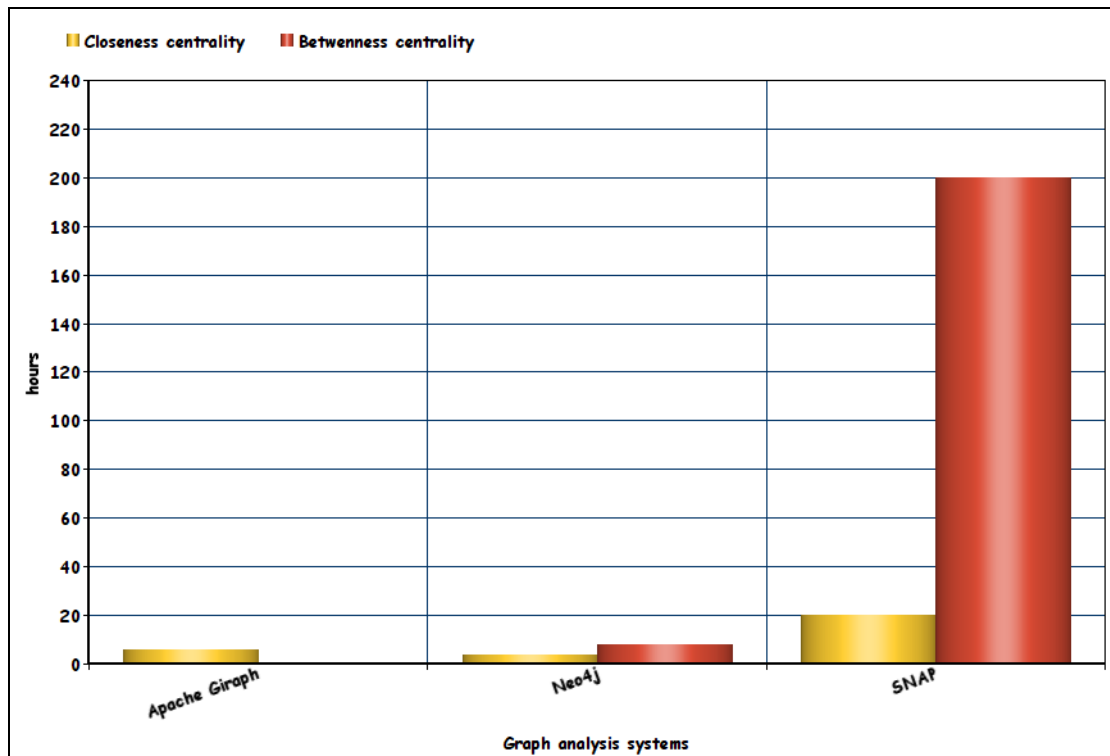
This dataset acquired from Stanford's Large Network Dataset Collection repository [14]. The network was collected by crawling Amazon website. It is based on Customers Who Bought This Item Also Bought feature of the Amazon website. If a product  $i$  is frequently co-purchased with product  $j$ , the graph contains an undirected edge from  $i$  to  $j$ . The total number of nodes is 334,863 and also extracted 925,872 edges.

The following chart displays the running times in seconds for Degree centrality and Eigenvector centrality.



Stanford's Network Analysis Platform (SNAP) has the best performance in both algorithms. It took 4.6 seconds to calculate Degree centrality and 29.3 seconds for Eigenvector centrality. Neo4j has the 2<sup>nd</sup> best performance among the three systems. It executes the calculation of the Degree centrality in 8.1 seconds and the Eigenvector centrality in 3 minutes and 15 seconds. Apache Giraph it calculates the Degree centrality in 18.1 seconds and Eigenvector centrality in 11 minutes and 32 seconds.

In the following chart displayed the running times in hours for Closeness centrality and Betweenness centrality.



Both Closeness centrality and Betweenness centrality executed faster in Neo4j. The execution time of Closeness centrality was 3 hours and 20 minutes and 7 hours and 53 minutes for Betweenness centrality. In Apache Giraph, Closeness centrality executed in approximately 6 hours. Finally, in Neo4j the execution time of Closeness centrality was 19 hours and 55 minutes and approximately 15 days for the complete execution of Betweenness centrality.

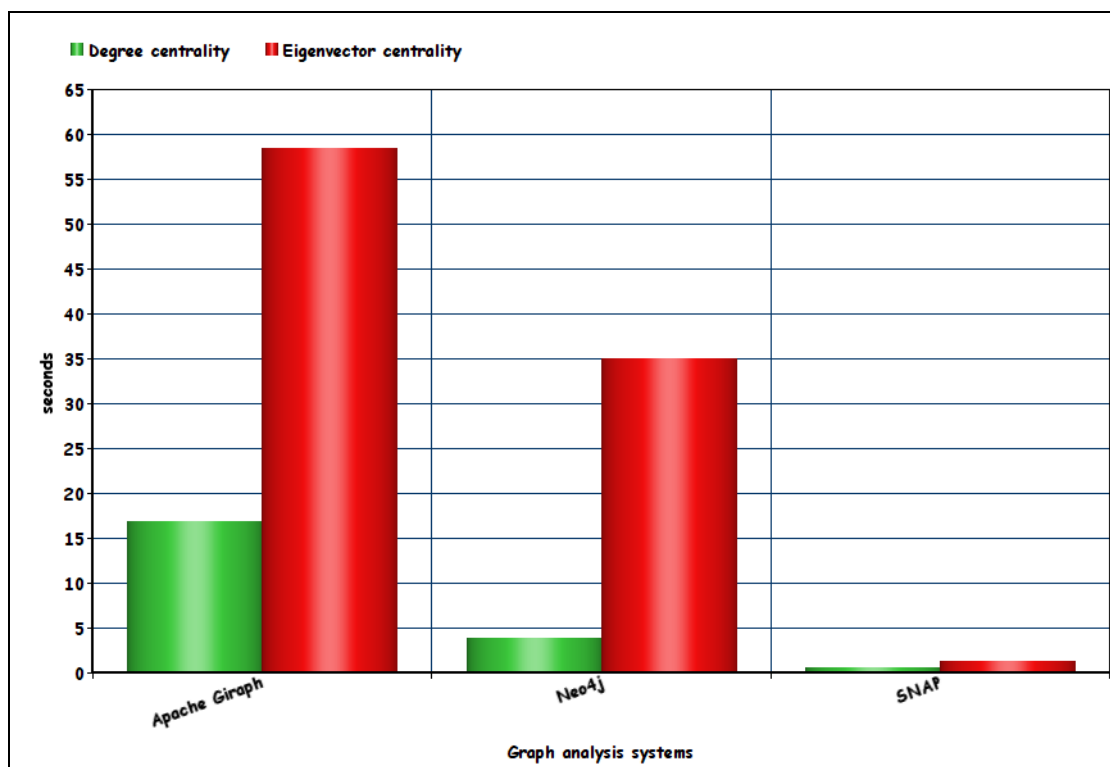
The Betweenness centrality is not calculated in Apache Giraph.



## 4.4.6 Email Enron

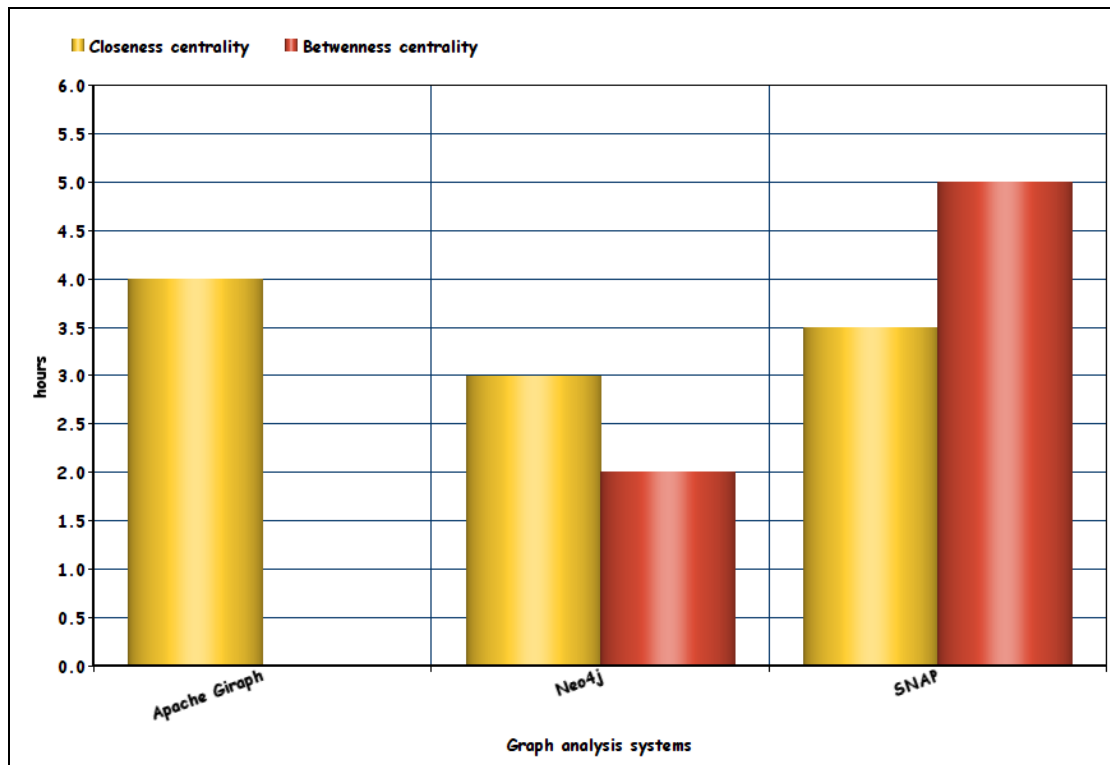
In order to construct Enron's e-mail network dataset we first acquire all the e-mails among Enron's employees from Enron Email Dataset repository which hosted by Carnegie Mellon University. The Enron Corpus is a large database of over 600,000 emails generated by 158 employees of the Enron Corporation and acquired by the Federal Energy Regulatory Commission during its investigation after the company's collapse. Then, we parse every e-mail in the data set and create a directed link from the sender to recipient. The total number of nodes is 36,692 and also extracted 183,831 edges.

The following chart displays the running times in seconds for Degree centrality and Eigenvector centrality.



Stanford's Network Analysis Platform (SNAP) has the best performance in both algorithms. It took 0.55 seconds to calculate Degree centrality and 1.3 seconds for Eigenvector centrality. Neo4j has the 2nd best performance among the three systems. It executes the calculation of the Degree centrality in 3.9 seconds and the Eigenvector centrality in 35 seconds. Apache Giraph it calculates the Degree centrality in 16.8 seconds and Eigenvector centrality in 58.4 seconds.

The following chart displays the running times in hours for Closeness centrality and Betweenness centrality.



Both Closeness centrality and Betweenness centrality executed faster in Neo4j. The execution time of Closeness centrality was 2 hours 55 minutes and 2 hours and 3 minutes for Betweenness centrality. In Apache Giraph, Closeness centrality executed in approximately 4 hours. Finally, in Neo4j the execution time of Closeness centrality was 7 hours and 53 minutes and 5 hours and 5 minutes for the complete execution of Betweenness centrality.

The Betweenness centrality it is not calculated in Apache Giraph.

## 5. Conclusion

Graph processing systems are increasingly important as more and more problems require dealing with graphs. The comparison of the three systems using data sets with different size and properties help us to understand better the individual properties of each graph processing system and studied them in depth.

Furthermore, the centrality metrics of social networks has been studied in depth and the correlation between research productivity and research collaboration has been examined.

### 5.1 Graph Analysis Platforms Performance Comparison

We presented a thorough comparison of three recent graph processing systems, SNAP, Neo4j and Apache Giraph, on six datasets using four different algorithms: Degree centrality, Closeness centrality, Betweenness centrality and Eigenvector centrality. We used a single machine with a 4-core Intel i-5 processor at 3.2 GHz and 16 GB of RAM memory.

The first of graph processing systems which we used, Stanford's Network Analysis Platform, is a light weighted graph library which it has Python and C++ distributions. It is very easy to understand it and code for. It is well documented and supported by Stanford University. Comparing to the other two systems it uses a very efficient way to manage the memory which occupied by the loaded graph. Even if the graph size is huge, like Twitter network which consists of 11,316,811 nodes and 85,331,845 edges, SNAP loads it using only few GBs of RAM. As far as its performance on the implemented algorithms time execution, it is by far the most efficient processing system in calculation of Degree centrality and Eigenvector centrality on small or medium sized data sets. However, in huge data sets like Twitter or in more complex algorithms like Closeness centrality or Betweenness centrality it is not performed as well as the other two systems. One of its disadvantages is that it uses single thread computations and that is the reason which not performed as well as Neo4j in very complex centrality algorithms.

The next graph system which we used is Neo4j. It is an open-source graph database implemented in Java. There are several ways of accessing the Neo4j database. We can use a Web UI and execute queries using Cypher Query Language, access a Neo4j database using Neo4j shell commands, we can use the native Java API and execute Cypher queries or we can use Java core API which offers a well implemented library to access the Neo4j database. For our experiments we use the Java core API to execute the four algorithms which we implemented. It is very easy to understand the core functions of Neo4j's Java API, and it is also includes a well-equipped documentation. Although it is not performed as well as SNAP in Degree centrality and Eigenvector centrality algorithms, Neo4j executes very fast the calculations for complex metrics such as Closeness and Betweenness centrality. The computation of these algorithms is faster because Neo4j executes the calculations using multiple threads and decrease the waste associated with unused slots. One disadvantage of Neo4j was the memory management. Due to its building elements it tends to store a lot of components which

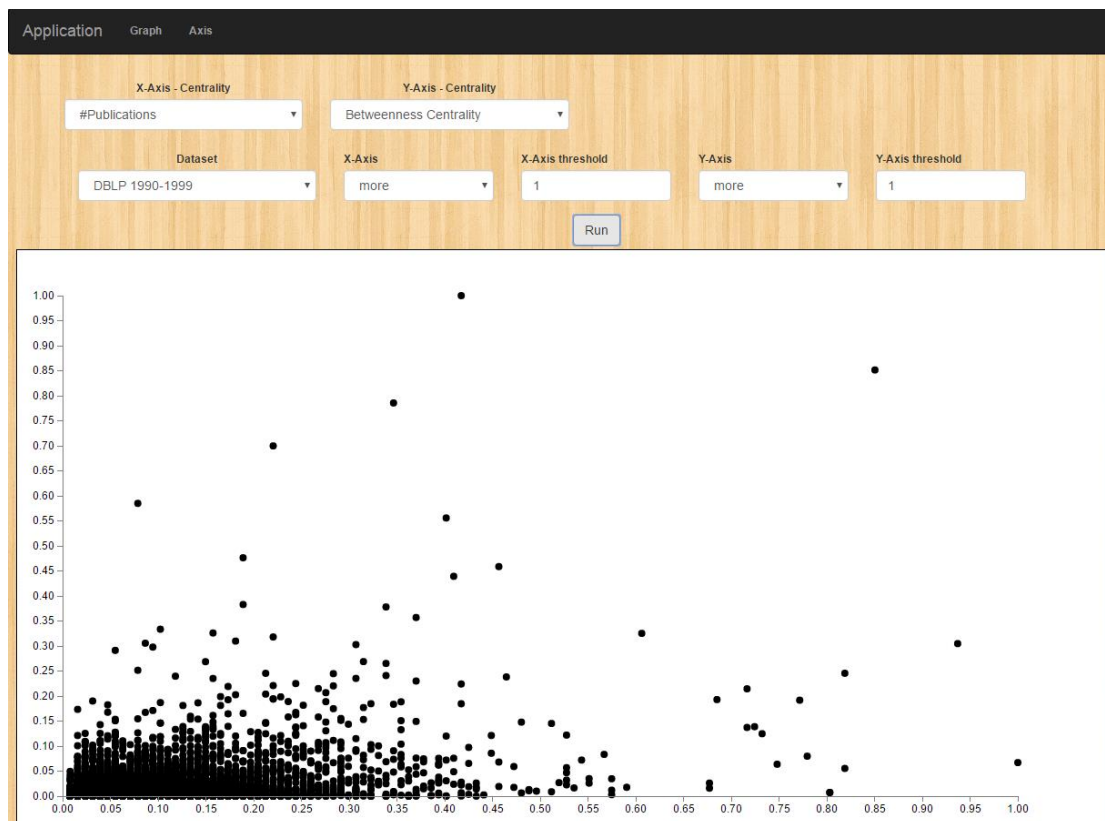
are meaningless for some of the algorithms, as a result Neo4j wastes a significant amount of memory. As an example, we cannot run any of the centrality algorithms on Twitter data set because the 16GBs of RAM are not enough to store the whole graph. The last graph processing system which we used is Apache Giraph. It is an iterative graph processing system built for high scalability. Apache Giraph using a vertex-centric programming model, where users express their algorithms by “thinking like a vertex”. Each vertex contains information about itself and all its outgoing edges, and the computation is expressed at the level of a single vertex. That is one of the main differences comparably to other two systems which they used a graph-centric approach. This vertex-centric model is very easy to program and has been proved to be useful for many graph algorithms. Apache Giraph is well documented, however, it has not a very active community to support it. Apache Giraph created to execute computations mainly on computer clusters using huge data sets. Using a single computer Apache Giraph cannot prove its capabilities, however, in a very large data set such as Twitter which includes 11,316,811 nodes and 85,331,845 edges, it calculates the Degree centrality metric 4 times faster than the second best system, SNAP. In small data sets it is not perform very well because added an overhead in centralities calculation because it needs additional time to load the graph during the startup of the algorithm execution. We faced memory issues during some expensive calculations, e.g. Closeness centrality, but we believe that issues can be eliminated on a concurrent environment.

## 5.2 Research productivity and Research collaboration correlation

In this thesis, it is also investigated the correlation between research productivity and research collaboration. Given all the analyses we performed it is very easy to use a system that can measure the research collaboration using Betweenness centrality metric. Intuitively, Betweenness centrality identifies the nodes with the most strategic location in a network. It favors nodes that join communities rather than nodes which lie inside a community, in other words shows which nodes act as ‘bridges’ between nodes in a network. In a collaboration network, nodes with high Betweenness centrality indicate that they have the power to control collaborative relationships. They also tend to attract more new co-authors because they prefer to attach to the existing authors who are controlling the flow of information by having a brokering (or bridging) role in the collaboration network.

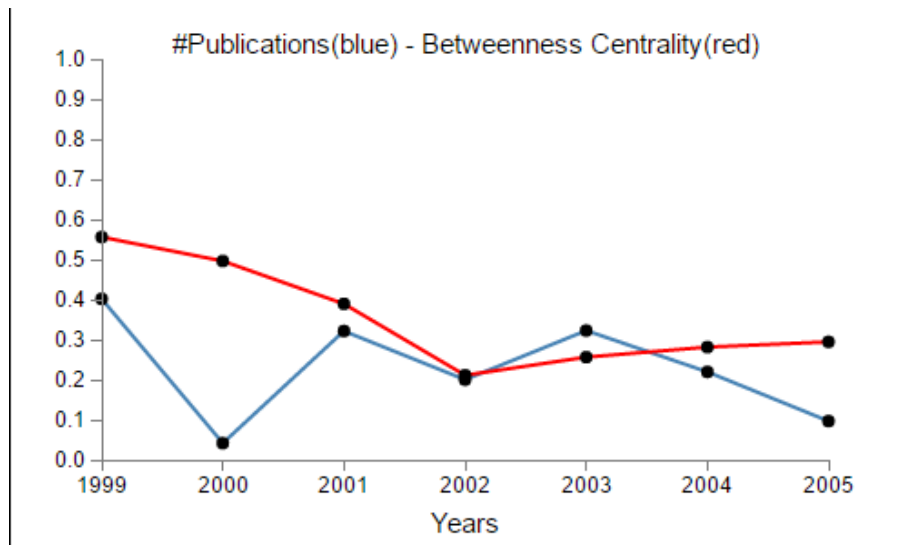
In order to measure the research productivity, we count the total papers of each author using DBLP data set. DBLP is a computer science bibliography website hosted at University of Trier, in Germany. The original data set includes 997,525 active nodes and 5,429,755 edges. We calculate Betweenness centrality and total publications first as a baseline from 1990 to 1999 and then for every additional year till 2005. We used a subset of the data set for the graph visualizations in the following slides.

Unfortunately, our hypothesis that there is a relationship between publication count (i.e., productivity) and collaboration “strength” as measured by Betweenness centrality does not seem to be correct.



However, we find out “important” nodes which meet our proposal. Subsequently, we present two examples.

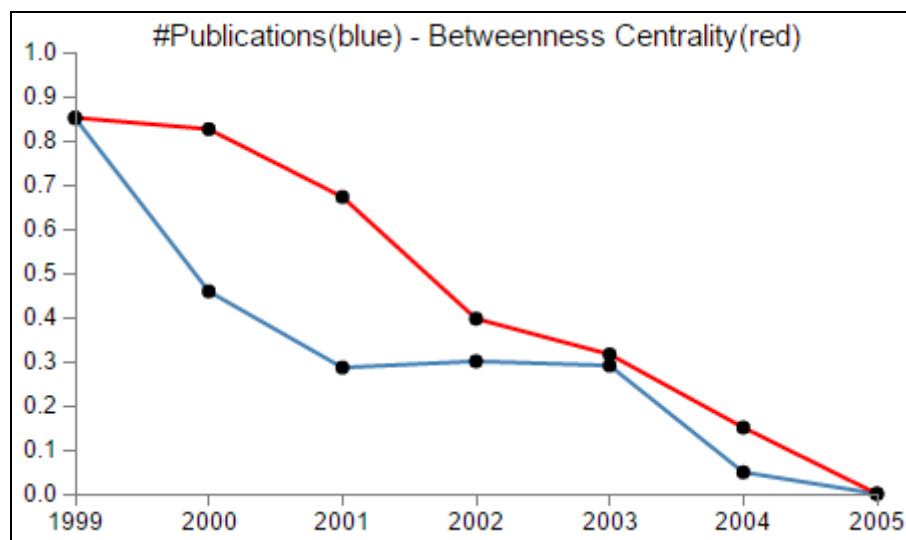
Rama Chellapa is the chair of the department of Electrical and Computer Engineering at University of Maryland.



Until 1999 he showed high Betweenness centrality and he published many scientific papers. In 2001 he published only one paper and his Betweenness centrality steadily decreased until 2002. Until then he was Associate Director of the Center for Automation Research at University of Maryland. At 2002 he became Director of this department and his Betweenness centrality increased.

Another example of important node in the network is Azriel Rosenfeld.

Azriel Rosenfeld was a distinguished university professor. He was a leading researcher in the field of computer image analysis in which he made many fundamental and pioneering contributions.



From 1990 to 1999 he held leading position among the top authors according Betweenness centrality and total publications. After his retirement in 2001 his total publications and the Betweenness centrality score reduced significantly. Finally, the two metrics became zero after his death at 2004.

The graphical illustration of these results is available on an online application, building with Javascript library D3.js, on the below URL.

<http://thesisapp.co.nf/>

## Bibliography

- [1] Alan Mislove, Hema Swetha Koppula, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2008. Growth of the flickr social network. In Proceedings of the first workshop on Online social networks (WOSN '08). ACM, New York, NY, USA, 25-30.
- [2] Meeyoung Cha, Alan Mislove, and Krishna P. Gummadi. 2009. A measurement-driven analysis of information propagation in the flickr social network. In Proceedings of the 18th international conference on World wide web (WWW '09). ACM, New York, NY, USA, 721-730.
- [3] Bimal Viswanath, Alan Mislove, Meeyoung Cha, and Krishna P. Gummadi. 2009. On the evolution of user interaction in Facebook. In Proceedings of the 2nd ACM workshop on Online social networks (WOSN '09). ACM, New York, NY, USA, 37-42.
- [4] Cha, M.; Haddadi, H.; Benevenuto, F. & Gummadi, K. (2010), Measuring User Influence in Twitter: The Million Follower Fallacy, in '4th International AAAI Conference on Weblogs and Social Media (ICWSM)'.
- [5] Julian McAuley and Jure Leskovec. 2014. Discovering social circles in ego networks. ACM Trans. Knowl. Discov. Data 8, 1, Article 4 (February 2014), 28 pages.
- [6] Bader, D. & Madduri, K. (2006), Parallel Algorithms for Evaluating Centrality Indices in Real-world Networks, in 'Proc. The 35th International Conference on Parallel Processing (ICPP)'.
- [7] Brandes, Ulrik, and Christian Pich. "Centrality estimation in large networks." International Journal of Bifurcation and Chaos 17.07 (2007): 2303-2318.
- [8] U. Brandes, A faster algorithm for betweenness centrality. Journal of Mathematical Sociology 25, 163–177 (2001).
- [10] Edmonds, Nick, Torsten Hoefler, and Andrew Lumsdaine. "A space-efficient parallel algorithm for computing betweenness centrality in distributed memory." 2010 International Conference on High Performance Computing. IEEE, 2010.
- [11] Minyang Han, Khuzaima Daudjee, Khaled Ammar, M. Tamer Özsu, Xingfang Wang, and Tianqi Jin. 2014. An experimental comparison of pregel-like graph processing systems. Proc. VLDB Endow. 7, 12 (August 2014), 1047-1058.
- [12] Salim Jouili and Valentin Vansteenberghe. 2013. An Empirical Comparison of Graph Databases. In Proceedings of the 2013 International Conference on Social Computing (SOCIALCOM '13). IEEE Computer Society, Washington, DC, USA, 708-715.
- [13] J. Koch, C.L. Staudt, M. Vogel, H. Meyerhenke: An Empirical Comparison of Big Graph Frameworks in the Context of Network Analysis. In revision.
- [14] Stanford Large Network Dataset Collection. <http://snap.stanford.edu/data/index.html>
- [15] Arizona State University Social Computing Data Repository. <http://socialcomputing.asu.edu/>
- [16] Apache Giraph. <http://giraph.apache.org/>
- [17] Neo4j. <https://neo4j.com/>
- [18] Claudio Martella, Dionysios Logothetis, Roman Shaposhnik. 2015. Practical Graph Analytics with Apache Giraph
- [19] DBLP. <http://dblp.uni-trier.de/>
- [20] Miriam Baglioni, Filippo Geraci, Marco Pellegrini, and Ernesto Lastres. 2012. Fast Exact Computation of betweenness Centrality in Social Networks. In Proceedings of the 2012 International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2012) (ASONAM '12). IEEE Computer Society, Washington, DC, USA, 450-456.



- [21] Eytan Bakshy, Jake M. Hofman, Winter A. Mason, and Duncan J. Watts. 2011. Everyone's an influencer: quantifying influence on twitter. In Proceedings of the fourth ACM international conference on Web search and data mining (WSDM '11). ACM, New York, NY, USA, 65-74.
- [22] Kamesh Madduri, David Ediger, Karl Jiang, David A. Bader, and Daniel Chavarria-Miranda. 2009. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing (IPDPS '09). IEEE Computer Society, Washington, DC, USA, 1-8.
- [23] Email Enron dataset. <https://www.cs.cmu.edu/~enron/>
- [24] Ghosh, Saptarshi, et al. "Understanding and combating link farming in the twitter social network." Proceedings of the 21st international conference on World Wide Web. ACM, 2012.
- [25] Day, Min-Yuh, Sheng-Pao Shih, and Weide Chang. "Understanding scientific collaboration with social network analysis." The 17th Cross-Strait Conference on Information and Management. 2011.
- [26] Abbasi, Alireza, and Jorn Altmann. "On the correlation between research performance and social network analysis measures applied to research collaboration networks." System Sciences (HICSS), 2011 44th Hawaii International Conference on. IEEE, 2011.
- [27] Apache Hadoop. <http://hadoop.apache.org/>
- [28] D3.js. <https://d3js.org/>

## APPENDIX A

In this section, we present the necessary code for the Closeness centrality algorithm in Apache Giraph

### A.1

```
public class VertexData implements Writable {
    private Map<Long,Double> closeness = new HashMap<Long,Double>();
    public VertexData() {}

    public VertexData(Map<Long,Double> closeness) {
        this.closeness = closeness;
    }

    public Map<Long,Double> getCloseness() {
        return closeness;
    }

    public void setCloseness(Map<Long,Double> closeness) {
        this.closeness = closeness;
    }

    @Override
    public void readFields(DataInput input) throws IOException {
        long size = input.readLong();
        this.closeness.clear();
        for(int i = 0; i < size; i++){
            this.closeness.put(input.readLong(),input.readDouble());
        }
    }

    @Override
    public void write(DataOutput output) throws IOException {
        output.writeLong(closeness.size());
        for(Entry<Long,Double> entry : closeness.entrySet()){
            output.writeLong(entry.getKey());
            output.writeDouble(entry.getValue());
        }
    }
}
```

## A.2

```
public class Message implements Writable {
    private Double value;
    private ArrayList<Long> ids = new ArrayList<Long>();
    public Message() {}

    public Message(Double value, ArrayList<Long> ids) {
        this.value = value;
        this.ids = ids;
    }

    public Double getValue() {
        return value;
    }

    public void setValue(Double value) {
        this.value = value;
    }

    public ArrayList<Long> getIds() {
        return ids;
    }

    public void setIds(ArrayList<Long> ids) {
        this.ids = ids;
    }

    @Override
    public void readFields(DataInput input) throws IOException {
        // TODO Auto-generated method stub
        this.value = input.readDouble();
        int size = input.readInt();
        this.ids.clear();
        for(int i = 0; i < size; i++){
            this.ids.add(input.readLong());
        }
    }

    @Override
    public void write(DataOutput output) throws IOException {
        // TODO Auto-generated method stub
        output.writeDouble(this.value);
        output.writeInt(ids.size());
        for(int i = 0; i < ids.size(); i++){
            output.writeLong(ids.get(i));
        }
    }
}
```