

**Κατανεμημένες υλοποιήσεις σε Hadoop μεθόδων  
ανάλυσης σύνθετων δικτύων μεγάλης κλίμακας**

---

**Hadoop-based distributed implementations of analysis  
methods for large-scale complex networks**

**Diploma Thesis**

**of**

**Kosmidou Maria**

**Supervisors:**

**Katsaros Dimitrios | assistant professor**

**Bozanis Panayiotis | associate professor**



**ΤΜΗΜΑ ΜΗΧ. Η/Υ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ & ΔΙΚΤΥΩΝ  
ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ**

---

**DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING  
UNIVERSITY OF THESSALY**

September 2016

Volos, Greece

*“Without Big Data, you are blind and deaf in the middle of a freeway”*

— Geoffrey Moore

*“Data is the new science. Big Data holds the answers”*

— Pat Gelsinger

# Acknowledgments

I would like to thank my supervisor Katsaros Dimitrios for his help from the very early stage of this research and guidance throughout this process of my thesis at the University of Thessaly. I would also like to thank Dr. Bozanis Panayiotis too, for giving me the opportunity to work on this interesting field of research.

# Περίληψη

Στον κλάδο της ανάλυσης δικτύων, οι μετρήσεις κεντρικότητας ενός κόμβου, είναι σημαντικές σε ένα μεγάλο αριθμό εφαρμογών που χρησιμοποιούν δίκτυα, από τον κλάδο της αναζήτησης και της κατάταξης των δικτύων έως στην ανάλυση των κοινωνικών και βιολογικών δικτύων. Σε αυτή την διπλωματική, μελετάμε συγκεκριμένα, την κεντρικότητα ενός κόμβου για σύνθετα δίκτυα και κατ' επέκταση για δίκτυα μεγάλης κλίμακας μέχρι τα δισεκατομμύρια των κόμβων και των ακμών. Διάφοροι ορισμοί για κεντρικότητα έχουν προταθεί, κάποιοι από αυτούς είναι αρκετά απλοί, π.χ., βαθμός του κόμβου, ενώ κάποιοι άλλοι πιο σύνθετοι, π.χ., PageRank. Ωστόσο, η μέτρηση κεντρικότητας σε γραφήματα της κλίμακας δισεκατομμυρίων θέτει πολλές προκλήσεις. Πολλοί από τους συνηθισμένους ορισμούς, όπως εγγύτητα και ενδιαμεσότητα δεν έχουν σχεδιαστεί με δυνατότητα κλιμάκωσης στο μυαλό. Ως εκ τούτου, είναι πολύ δύσκολο, αν όχι αδύνατο, να υπολογίσουμε τόσο με ακρίβεια και αποτελεσματικότητα τέτοιου είδους μετρικές. Τέλος, η ανάπτυξη των αλγορίθμων για τον υπολογισμό των προτεινόμενων μετρικών κεντρικότητας, γίνεται σε περιβάλλον Hadoop / MapReduce, ένα σύγχρονο περιβάλλον για μεγάλης κλίμακας, κατανεμημένη επεξεργασία δεδομένων, που βασίζεται στο Cloud Computing.

**Λέξεις Κλειδιά:** σύνθετα δίκτυα, μεγάλης κλίμακας, κεντρικότητα, Hadoop, MapReduce, Cloud Computing

# Abstract

In the network analysis, the centrality metrics of a node, are important in a large number of graph applications, from search and ranking to social and biological network analysis. In this thesis, we study node centrality for complex networks, by extension large – scale networks up to billions of nodes and edges. Various definitions for centrality have been proposed, ranging from very simple, e.g., degree of the node, to more complex. However, measuring centrality in billion-scale graphs poses several challenges. Many of the usual definitions such as closeness and betweenness were not designed with scalability in mind. Therefore, it is very difficult, if not impossible, to compute them both accurately and efficiently. Finally, the development of the algorithms to compute the proposed centrality measures, is done in Hadoop/MapReduce, a modern environment for large-scale, distributed data processing, that is based on Cloud Computing.

**Keywords:** complex networks, large-scale, centrality, Hadoop, MapReduce, Cloud Computing

# Table of Contents

<b>Chapter 1. Introduction</b> .....	<b>9</b>
1.1 The scope of thesis .....	9
1.2 Organization of thesis .....	9
<b>Chapter 2. Background and Related Work</b> .....	<b>11</b>
2.1 Complex Networks .....	11
2.1.1 <i>Large-scale networks</i> .....	13
2.1.2 <i>Challenges and problems of Large-scale analysis</i> .....	14
2.2 Cloud Computing.....	15
2.2.1 <i>Map/Reduce and Hadoop</i> .....	18
2.3 Graph Theory.....	22
2.3.1 <i>Graph Representation</i> .....	23
2.3.2 <i>Definitions</i> .....	24
2.3.3 <i>Metrics</i> .....	25
2.4 Related Work.....	26
<b>Chapter 3. Implementations</b> .....	<b>27</b>
3.1 Degree Centrality .....	27
3.2 Power Community Index.....	29
3.3 Closeness Centrality .....	31
3.4 Shortest Path Betweenness Centrality .....	35
3.5 Page Rank .....	39
<b>Chapter 4. Experiments and Results</b> .....	<b>42</b>
<b>Chapter 5. Conclusions and Future Work</b> .....	<b>46</b>
<b>Chapter 6. Appendix</b> .....	<b>47</b>
6.1 Degree Centrality's Experiments .....	47
6.2 Power Community Index's Experiments .....	50
6.3 Closeness Centrality's Experiments.....	52
6.4 Shortest Path Betweenness Centrality's Experiments .....	54
6.5 Page Rank's Experiments .....	56
<b>Chapter 7. Bibliography</b> .....	<b>59</b>

# List of Figures

Figure 1 Various types of networks .....	11
Figure 2 Examples of Complex networks .....	12
Figure 3 Visualization of daily Wikipedia edits created by IBM, as an example of big data. ( <a href="https://en.wikipedia.org/wiki/Big_data#/media/File:Viegas-UserActivityonWikipedia.gif">https://en.wikipedia.org/wiki/Big_data#/media/File:Viegas-UserActivityonWikipedia.gif</a> ) .....	14
Figure 4 The pyramid of cloud computing architecture .....	17
Figure 5 HDFS architecture.....	20
Figure 6 A MapReduce job.....	21
Figure 7 Adjacent list and Adjacent matrix of a graph.....	24
Figure 8 Two basic categories of centralities.....	26
Figure 9 Example of a simple undirected graph .....	43
Figure 10 Scalability of metrics.....	45
Figure 11 Output of DC .....	47
Figure 12 Small network experiment for DC .....	48
Figure 13 Medium network experiment for DC.....	48
Figure 14 Large network experiment for DC .....	49
Figure 15 Output of PCI .....	50
Figure 16 Small network experiment for PCI .....	51
Figure 17 Medium network experiment for PCI.....	51
Figure 18 Large network experiment for PCI .....	52
Figure 19 Output of CC .....	52
Figure 20 Small network experiment for CC .....	53
Figure 21 Medium network experiment for CC .....	54
Figure 22 Output of SPBC .....	54
Figure 23 Small network experiment for SPBC .....	55
Figure 24 Medium network experiment for SPBC.....	56
Figure 25 Output of PR.....	56
Figure 26 Small network experiment for PR.....	57
Figure 27 Medium network experiment for PR.....	57
Figure 28 Large network experiment for PR .....	58

# List of Flowcharts

Flowchart 1 Degree Centrality process .....	28
Flowchart 2 Power Community Index process .....	31
Flowchart 3 Closeness Centrality process.....	34
Flowchart 4 Shortest Path Betweenness Centrality process .....	38
Flowchart 5 PageRank process .....	41

# List of Tables

Table 1 Characteristics of computing system .....	42
Table 2 Characteristics of network experiments .....	42
Table 3 Rankings.....	43
Table 4 Kendall tau coefficient ( $\tau$ ).....	44
Table 5 Correlation of metrics.....	44

# Chapter 1. Introduction

## 1.1 The scope of thesis

Nowadays, social network and other complex-network applications become more and more popular in the world. Complex networks are getting larger and the need to analyze datasets with millions of nodes and billions of edges is not uncommon any more. Complex network analysis is always based on large scale raw data, so the performance of complex-network analysis becomes the key of its development and fast algorithms are desirable for the recomputation of key network measures such as centrality. Node centrality measures are important in a large number of graph applications, from search and ranking to social and biological network analysis.

Hadoop, as an implementation of the MapReduce parallel framework, a modern paradigm for large-scale, distributed data processing, is very suitable for large-scale data analysis [1]. Programs running in Hadoop are automatically parallelized and executed on the large cluster. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication.

Generally, this thesis develops Map-Reduce implementations of analysis methods for large-scale complex networks, but more specifically, the scope is to compute different metrics of graph's centrality in Hadoop. Various definitions for centrality have been proposed, some of them are degree centrality, power community index, closeness centrality, betweenness centrality and PageRank. This thesis presents also, experimental results on both synthetic and real datasets, which demonstrate the functionality of these algorithms to very large graphs.

## 1.2 Organization of thesis

This thesis is organized as follows. Chapter 2 describes the background knowledge and all the related work in large scale complex networks and cloud computing. This includes complex networks and more specifically large scale networks (2.1), cloud computing and the Hadoop framework (2.2), graph theory (2.3) and the related work (2.4). Chapter 3 presents detailed the Map/Reduce algorithms of five metrics implemented in this undergraduate thesis. Chapter 4 contains the experiments for every Map/Reduce

algorithm that they conducted in several graphs, along with the results. Finally, Chapter 5 concludes the thesis and presents directions for future work.

# Chapter 2.

# Background and Related Work

## 2.1 Complex Networks

The term 'complex networks' is young. It came to use in the twenty-first century when researchers from very distinct sciences, such as computer scientists, biologists, sociologists, physicists, and mathematicians started to intensively study diverse real-world networks and their models. This notion refers to networks with more complex architectures, for example, a uniformly random graph with given numbers of nodes and links, like internet. This is the quantitative characterization of a system, but on the other hand there is also the quality characterization. As a quality of the system it refers to what makes the system complex, in this case complexity refers to the presence of emergent properties in the system. That is, to the properties which emerge as a consequence of the interactions of the parts in the system. In this sense, the great majority of real-world networks are complex.

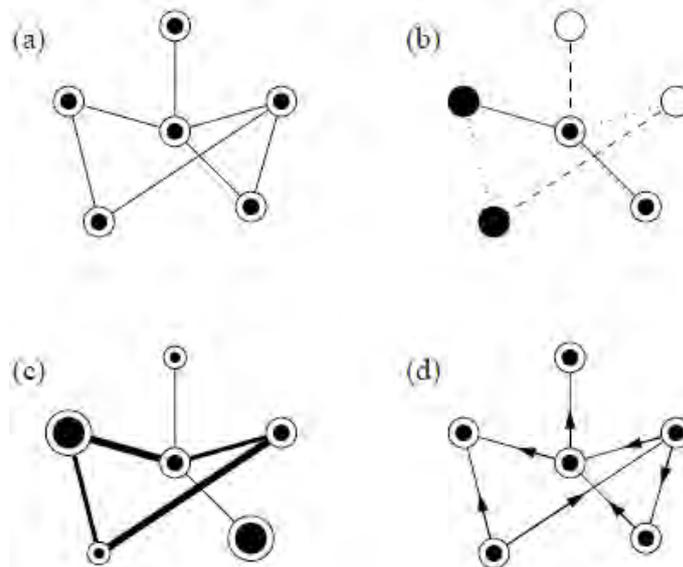
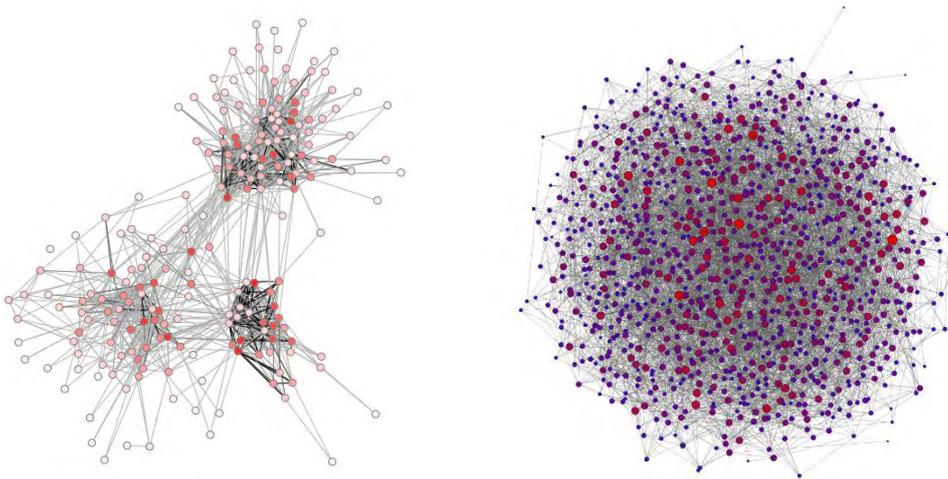


Figure 1 Various types of networks

In more detail, there are many ways in which networks may be more complex than the graphs are illustrated in (Figure 1. a). For instance, there may be more than one different type of vertex in a network, or more than one different type of edge (b). Taking the example of a social network of people, the vertices may represent men or women, people of different nationalities, locations, ages, incomes, or many other things. Edges may

represent friendship, but they could also represent animosity, or professional acquaintance, or geographical proximity. They can carry weights, representing, say, how well two people know each other (c). They can also be directed, pointing in only one direction (d). A graph representing telephone calls or email messages between individuals would be directed, since each message goes in only one direction. Directed graphs can be either cyclic, meaning they contain closed loops of edges, or acyclic meaning they do not. Some networks, such as food webs, are approximately but not perfectly acyclic. A complex network can also have hyper edges that could be used to indicate family ties in a social network for example,  $n$  individuals connected to each other by virtue of belonging to the same immediate family could be represented by an  $n$ -edge joining them. There are more types of graphs, but the study of networks is by no means a complete science yet, and many of the possibilities have yet to be explored in depth.



*Figure 2 Examples of Complex networks*

Furthermore, complex networks can be classified according to the nature of the interactions among the entities forming the nodes of the network. Some examples of these classes are:

- Physical linking: pairs of nodes are physically connected by a tangible link, such as a cable, a road, a vein, etc.  
Examples: Internet, urban street networks, road networks, vascular networks, etc.
- Physical interactions: links between pairs of nodes represents interactions which are determined by a physical force.  
Examples: protein residue networks, protein-protein interaction networks, etc.
- Geographic closeness: nodes represent regions of a surface and their connections are determined by their geographic proximity.  
Examples: countries in a map, landscape networks, etc.

- “Ethereal” connections: links between pairs of nodes are intangible, such that information sent from one node is received at another irrespective of the “physical” trajectory. Examples: WWW, airports network.
- Mass/energy exchange: links connecting pairs of nodes indicate that some energy or mass has been transferred from one node to another.  
Examples: reaction networks, metabolic networks, food webs, trade networks, etc.
- Social connections: links represent any kind of social relationship between nodes.  
Examples: friendship, collaboration, etc.
- Conceptual linking: links indicate conceptual relationships between pairs of nodes. Examples: dictionaries, citation networks, etc.

### 2.1.1 Large-scale networks

Complex networks are essentially large graphs of real life. Large amounts of network data are being produced by various modern applications at an ever-growing speed, ranging from social networks such as Facebook and Twitter, scientific citation networks such as VOSviewer, to biological networks such as gene regulatory networks (DNA–protein interaction networks). Network data analysis is crucial to exploit the wealth of information encoded in these network data. An effective analysis of these data must take into account the complex structure including social, temporal and sometimes spatial dimensions, and an efficient analysis of these data demands scalable solutions. As a result, there has been increasing research in developing scalable solutions for novel large-scale network analytics applications.

Big data analytics is the process of collecting, organizing and analyzing large sets of data to discover patterns and other useful information. Big data analytics can help organizations to better understand the information contained within the data and will also help identify the data that is most important to the business and future business decisions. Analysts working with big data basically want the knowledge that comes from analyzing the data.

Modern informatics applications like web search afford easy parallelization, e.g. the overall index can be partitioned such that even a single query can use multiple processors. Moreover, the peak performance of a machine is less important than the price-performance ratio. In this environment, scalability up to petabyte-sized data often means working in a software framework like MapReduce/Hadoop that supports data-intensive distributed computations running on large clusters of hundreds, thousands, or

even hundreds of thousands of commodity computers. This differs substantially from the scalability issues that arise in traditional applications of interest in scientific computing.

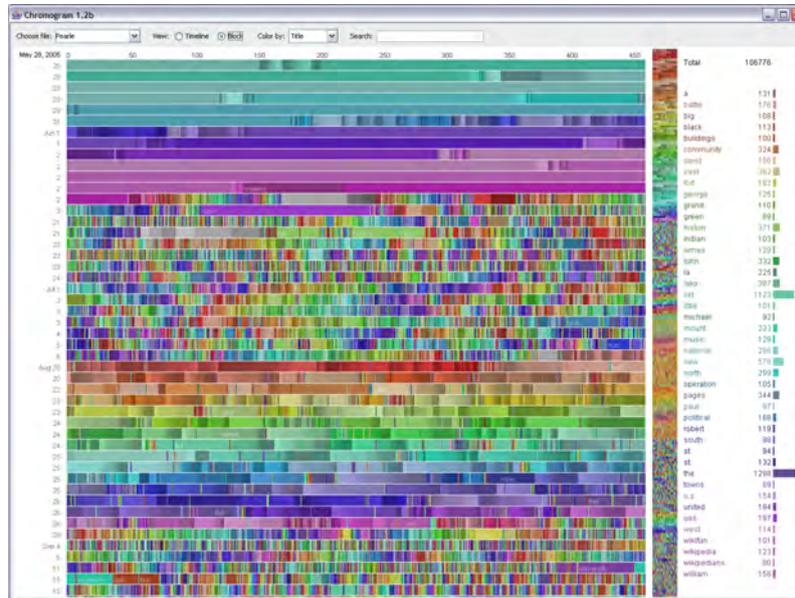


Figure 3 Visualization of daily Wikipedia edits created by IBM, as an example of big data. ([https://en.wikipedia.org/wiki/Big\\_data#/media/File:Viegas-UserActivityonWikipedia.gif](https://en.wikipedia.org/wiki/Big_data#/media/File:Viegas-UserActivityonWikipedia.gif))

## 2.1.2 Challenges and problems of Large-scale analysis

For most organizations, big data analysis is a challenge. Consider the sheer volume of data and the different formats of the data, both structured and unstructured data that is collected across the entire organization and the many different ways of data retrieval can be combined, contrasted and analyzed to find patterns and other useful business information.

The first challenge is in breaking down databases to access all data an organization at stores in different places and often in different systems. A second big data challenge is in creating platforms that can pull in unstructured data as easily as structured data. This massive volume of data is typically so large that it's difficult to process using traditional database and software methods.

Except of these challenges, there are further difficulties in large-scale analysis, but one of the basic problems is the efficiently and effectively measuring centrality for billion-scale networks. More specifically, there are the following problems:

1. Design. Careful design centrality measures that avoid inherent limitations to scalability and parallelization.
2. Algorithms. Fast computations of large scale centralities for billion-scale graphs.

3. Observations. Key patterns and observations on centralities in large, real world networks.

## 2.2 Cloud Computing

With the rapid development of processing, the storage technologies and the success of the Internet, computing resources have become cheaper, more powerful and more available than ever before. This technological trend has enabled the realization of a new computing model called Cloud Computing. Cloud computing divides the role of service provider into two: the *infrastructure providers*, who manage cloud platforms and lease resources according to a usage-based pricing model and *service providers*, who rent resources from one or many infrastructure providers to serve the end users.

The emergence of cloud computing has made a tremendous impact on the Information Technology (IT) industry over the past few years, where large companies such as Google and Microsoft try to provide more powerful, reliable and cost-efficient cloud platforms.

**NIST<sup>1</sup> definition of cloud computing** *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*

Cloud computing is therefore a type of computing that relies on sharing a pool of physical and/or virtual resources, rather than deploying local or personal hardware and software. It is somewhat synonymous with the term 'utility computing' as users are able to tap into a supply of computing resource rather than manage the equipment needed to generate it themselves; much in the same way as a consumer tapping into the national electricity supply, instead of running their own generator.

One of the key characteristics of cloud computing is the flexibility that it offers and one of the ways that flexibility is offered is through scalability. This refers to the ability of a system to adapt and scale to changes in workload. Cloud technology allows for the automatic provision and privation of resource as and when it is necessary, thus ensuring that the level of resource available is as closely matched to current demand as possible.

The basic cloud model is composed of five essential characteristics, four deployment models and three service models.

---

<sup>1</sup> National Institute of Standards Technology

### ESSENTIAL CHARACTERISTICS:

- **On-demand self-service.** A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.
- **Broad network access.** Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).
- **Resource pooling.** The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.
- **Rapid elasticity.** Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.
- **Measured service.** Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service.

### DEPLOYMENT MODELS:

- **Private cloud.** It uses pooled services and infrastructure stored and maintained on a private network – whether physical or virtual – accessible for only one client. The obvious benefits to this are greater levels of security and control. Cost benefits must be sacrificed to some extent though, as the enterprise in question will have to purchase/rent and maintain all the necessary software and hardware
- **Community cloud.** The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.

- **Public cloud.** It is a cloud in which services and infrastructure are hosted off-site by a cloud provider, shared across their client base and accessed by these clients via public networks such as the internet. Public clouds offer great economies of scale and redundancy but are more vulnerable than private cloud setups due their high levels of accessibility.
- **Hybrid cloud.** This as the name suggests, combines both public and private cloud elements. A hybrid cloud allows a company to maximize their efficiencies, by utilizing the public cloud for non-sensitive operations while using a private setup for sensitive or mission critical operations, companies can ensure that their computing setup is ideal without paying any more than is necessary.

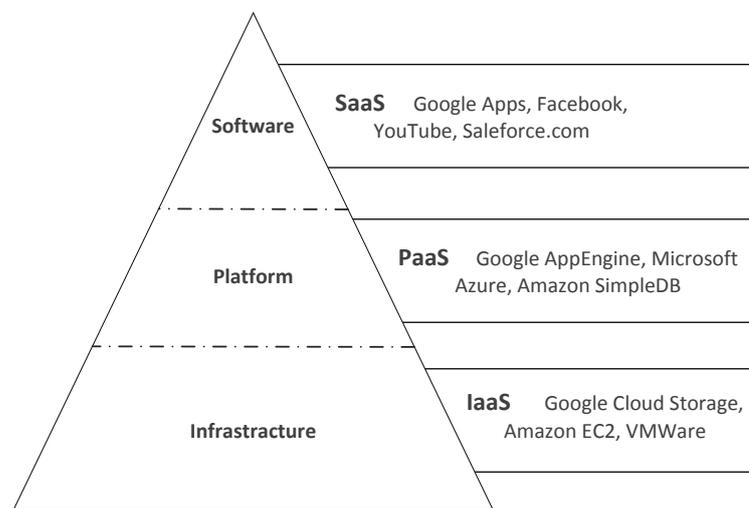


Figure 4 The pyramid of cloud computing architecture

Moving away from deployment models, there are three models of cloud computing which describe the service on offer.

#### **SERVICE MODELS:**

- **Software as a Service (SaaS).** SaaS is arguably the most common of the cloud computing variations; it's the term used to describe a software delivery model in which applications are hosted (usually by a provider) and made available to customers over a network connection. Many people make use of SaaS without realizing it as many web applications are delivered in this way; Gmail, Flickr, Twitter and Facebook are all popular examples of SaaS. Enterprise users also frequently make use of SaaS with many popular accounting, invoicing, sales, communications and CRM systems being delivered this way

- **Platform as a Service (PaaS).** PaaS is an extension of IaaS and describes a category of cloud computing that provides developers with environments in which to build applications, over the internet. In addition to the fundamental computing resource supplied by the hardware in an IaaS offering, PaaS models also include the software and configuration (often known as the solution stack) required to create the platform on which clients can create their applications. PaaS packages can be tailored to meet individual user needs; they can cherry pick the features of the service that are relevant to them while disregarding those that are not. PaaS provides a number of benefits to enterprises, including simplifying the development process for geographically split development teams.
- **Infrastructure as a Service (IaaS).** IaaS refers to the delivery of virtualized computing resource as a service across a network connection. It specifically deals with hardware – or computing infrastructure - delivered as a service. Offerings include virtualized server space, storage space, network connections and IP addresses. The resource is pulled from a pool of servers distributed across data centers under the provider’s control, the user is then granted access to this resource in order to build their own IT platforms. IaaS can provide enterprises with great business benefits.

### 2.2.1 Map/Reduce and Hadoop

Distributed processing on a *cloud*—a large collection of commodity computers, each with its own disk, connected through a network—has the same problem with the study of real world networks. It is that the information is extremely large, extending from hundreds of edges to billions of edges. Obviously, it is difficult to apply sequential algorithms to analyze these graphs. This size has led to the development of parallelization architectures. One recent and effective framework that permits the development of parallelized algorithms is Hadoop. Hadoop provides us with a distributed filesystem and the implementation of the map/reduce programming model, as well as all the necessary libraries that are needed in order for a compute cluster to function. Its main advantage is that it separates the parallelization code from the business logic, thus making easy for anyone to create and execute a parallel algorithm. Additionally, it poses no restrictions regarding the number of computer nodes that the cluster should have, something that has been an issue in older architectures.

In addition, due to Hadoop’s excellent scalability, ease of use, and cost advantage, Hadoop has been used for important graph mining algorithms. Other variants which provide advanced MapReduce, like systems include SCOPE, Sphere, and Sawzall.

Particularly, Hadoop is a project of the Apache Software Foundation that parallelizes data processing across many nodes in a compute cluster, speeding up large computations and hiding I/O latency through increased concurrency. The advantages of this model lie in its ability to deal with the issues of distributing the data, handling failures, load balancing among the cluster, thus separating the business logic from the parallelization code; hence, developers are free to focus on application logic. The Hadoop project includes various subprojects that provide complementary services. These are:

- MapReduce: a distributed data processing model and execution environment that runs on compute clusters.
- HDFS: a distributed filesystem that provides high throughput access to application data.
- Chukwa: a distributed data collection and analysis system.
- Hive: a data warehouse infrastructure that provides a query language based on SQL.
- Pig: a high-level data flow language and execution framework for parallel computation. It is built on top of Common.
- Zookeeper: a high-performance coordination service for distributed applications.

Hadoop implements the MapReduce programming model. The user of this library needs to implement two functions – map and reduce – to perform a computation. Each input record is converted into a key/value pair. A map operation is applied to each input record and produces a set of intermediate key/value pairs. The map outputs are grouped and sorted by key. A reduce operation is applied to all values that share the same key, in order to combine the derived data appropriately.

HDFS is a file system designed to store large files across multiple machines. Storage reliability is achieved with the data replication on several nodes. Three processes control the HDFS services. *Namenode* manages the filesystem namespace and regulates access to files by clients. It is a single point of failure for an HDFS installation, as if it goes down the system is offline. It is responsible for operations like opening, closing and renaming of files and directories available via an RPC interface. Also, it determines the mapping of blocks to Datanodes. *Secondary Namenode* is a process that regularly connects to the Namenode and downloads a snapshot of its directory information, which is then saved to a directory. The Secondary Namenode is used together with the edit log of the Namenode to create an up-to-date directory structure. *Datanode* is a process that provides block storage and retrieval services like serving read/write requests from

clients and performing block creation, deletion and replication upon instruction from the Namenode.

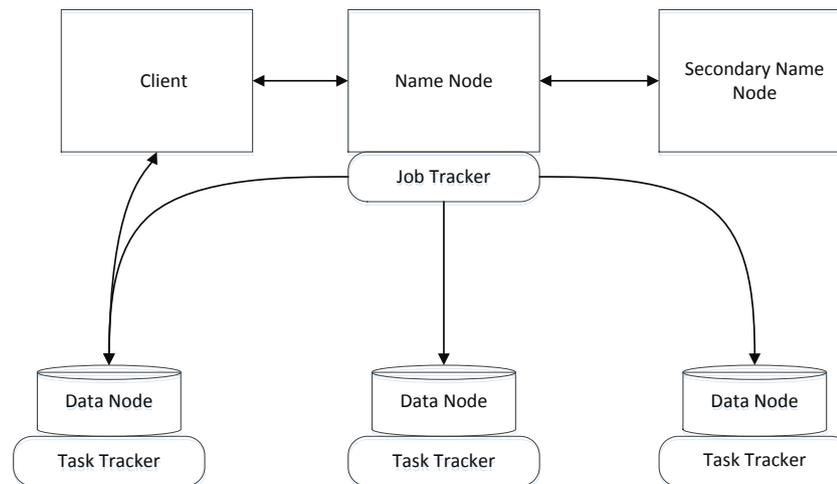


Figure 5 HDFS architecture

The Hadoop framework provides two processes that handle the execution of MapReduce jobs. *TaskTracker* manages the execution of individual map and reduce tasks on a compute node in the cluster and *JobTracker* accepts job submissions, provides job monitoring and control and manages the distribution of tasks to the TaskTracker nodes. When a MapReduce job is submitted by the user, it is decomposed into a number of tasks. The user is responsible for submitting the job configuration in order to provide the framework with a series of necessary parameters regarding the job, like the input and output destination in HDFS, the input and output format, the classes that contain the map and reduce functions and the JAR file(s) that contain the map and reduce functions and any support classes. Then, the input is split according to the HDFS block size (typically 64 MB) and distributed across the map tasks. If the input is N files, then at least there will be N map tasks. The map tasks are executed and produce the intermediate key/value pairs according to the map function that is specified by the user. Each map function receives one record (line) from the split and process it accordingly. Then, follows the shuffle phase where the map outputs are partitioned and sorted. The shuffle output for each partition is sorted. Afterwards, the reduce tasks start with input the data that correspond to their partition. Each reduce function is called once for each input unique key with all the values that share that key. The reduce tasks emit key/value pairs, which are written to output directory. The number of output files in the directory will be as many as the number of reduce tasks that were executed.

MapReduce has two benefits: (a) The data distribution, replication, fault-tolerance, and load balancing is handled automatically; and furthermore (b) it uses the

familiar concept of functional programming. The programmer needs to define only two functions, a map and a reduce. The general framework is as shows in Figure 6: (a) the map stage reads the input file and emits (key, value) pairs; (b) the shuffling stage sorts the output and distributes them to reducers; (c) the reduce stage processes the values with the same key and emits another (key, value) pairs which become the final result.

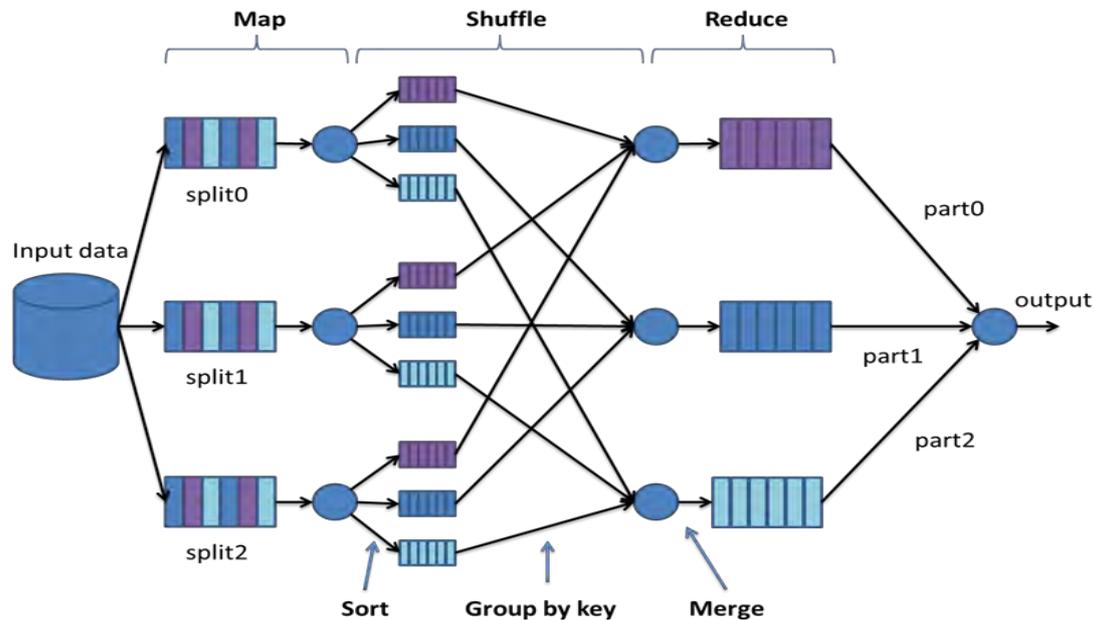


Figure 6 A MapReduce job

The map and the reduce functions have 4 parameters. The *key*, the *value*, the *output collector* and the *reporter*. The output collector is the object used to emit the key/value pairs. The reporter object provides the mechanism for informing the framework of the current status of the job. If a job takes too long to complete, it is useful to inform the framework that it is still working through the reporter, so that the framework will not kill it.

As it is appreciated, a large variety of input formats are supplied by the framework. The major distinctions are between textual and binary input formats. The available formats are:

**FORMATS OF HADOOP:**

- KeyValueTextInputFormat: key/value pairs, one per line.
- TextInputFormat: the key is the byte offset of the line and the value is the line.
- NLineInputFormat: similar to KeyValueTextInputFormat, but the splits are based on N lines of input rather than Y bytes of input.
- MultiFileInputFormat: an abstract class that lets user implement an input format that aggregates multiple files into one split.

- SequenceFileInputFormat: the input file is a Hadoop sequence file, containing serialized key/value pairs.

Hadoop provides its own set of data types that are optimized for network serialization and correspond to the known Java built-in data types. Of course, the user can define custom data types if necessary. The data types that are used as keys need to implement the WritableComparable and the data types that are used as values need to implement the Writable interface, which is a subset of WritableComparable. The Writable interface implements the methods that are used for serialization and deserialization of the objects and the WritableComparable implements additionally the methods that are used for the comparison of the keys.

The most common Hadoop data types are:

#### **DATA TYPES OF HADOOP:**

- Text: equivalent to String.
- IntWritable: equivalent to Integer.
- VIntWritable: used for integer values stored in variable-length format. Such values take between 1-5 bytes. Smaller values take fewer bytes.
- LongWritable: equivalent to Long.
- VLongWritable: used for long values stored in variable-length format. Such values take between 1-5 bytes. Smaller values take fewer bytes.
- FloatWritable: equivalent to Float.
- DoubleWritable: equivalent to Double.
- ByteWritable: equivalent to Byte.
- BytesWritable: used for byte arrays.
- BooleanWritable: equivalent to Boolean.
- NullWritable: equivalent to Null.

## **2.3 Graph Theory**

**A**s it was defined above network science studies representations of physical, biological and social phenomena and seeks to discover common principles that govern network behavior. A network is a set of entities, which are pairwise connected with links. In computer science, networks are represented as graphs, where the entities correspond to vertices which are connected with edges. Examples include the World Wide Web, where the vertices are the web pages and links from one page to another form edges; social networks, where the vertices are people and edges express some sort of acquaintance like friendship or relativity; co-author ship networks, where

the vertices are scientists in a particular discipline and edges connect those who have co-authored a paper; collaboration networks, where the vertices are employees and edges are formed between those who have worked in common projects; biological networks that express relations among proteins or neurons.

In more detail, graphs are mathematical structures used to model pairwise relations between objects from a certain collection. A graph is a collection of objects, where some pairs of the objects are connected by links. The objects are called vertices or nodes and the links are called edges or arcs. The edges may be *directed* – asymmetric or *undirected* – symmetric. The corresponding graphs are called directed or digraphs and undirected graphs. Of course, we can represent an undirected graph as directed if we have two edges between every pair of nodes, one for each direction. The edges may carry weights, that could represent costs, length, capacities or other quantities depending on the problem. These edges define a graph as *weighted*. Graphs can be either *cyclic*, meaning they contain closed loops of edges or *acyclic* meaning they do not. Also, there is a *subgraph* of a graph  $G$ , which is a graph whose vertex set is a subset of  $G$ , and whose adjacency relation is a subset of  $G$  restricted to the new vertex subset. In the other direction, a *super graph* of a graph  $G$  is a graph of which  $G$  is a subgraph. *Bipartite* graphs are graphs whose vertices are divided into two disjoint sets  $U$  and  $V$ , such that every edge connects a vertex from  $U$  to one in  $V$ . Furthermore, a graph may have hyper edges. Hyper edges join more than two vertices together. Graphs containing such edges are called *hypergraphs*.

The goal of network analysis is to model the interactions among the entities and discover interesting patterns, by focusing on the properties of real world networks. Patterns that have been discovered include the small world effect, the shrinking diameter and many others. One important property of networks is that they evolve over time with edges appearing or disappearing.

### 2.3.1 Graph Representation

There are different ways to represent a network mathematically, the two most commonly used data structures for representing a graph  $G = (V, E)$  are the adjacency list and the adjacency matrix.

The adjacency list is implemented as an array of  $|V|$  lists, with one list of destination nodes for each source node. The vertices in each adjacency list are typically stored in an arbitrary order. In directed graphs the sum of the lengths of all adjacency lists is  $|E|$ , while in undirected graphs it is  $2|E|$ . This happens because in undirected graphs, each edge  $(u, v)$ , is stored in the list of node  $u$ , as well as in the list of node  $v$ .

The adjacency matrix is a two-dimensional Boolean matrix, of length  $|V| \times |V|$ . It is assumed that the identities of vertices vary from  $0 \dots |V|$ . A matrix entry  $(i, j)$  indicates if there is an edge from vertex  $i$  to  $j$ . Formally, the adjacency matrix  $A$  of a simple graph is the matrix with elements  $A_{ij}$ , such that

$$A_{ij} = \begin{cases} 1, & \text{if } (i, j) \in E \\ 0, & \text{otherwise} \end{cases}$$

Adjacency matrices of undirected graphs are symmetric, as for every edge  $(i, j)$ , there also exists an edge  $(j, i)$ . The transposed adjacency matrix of  $A = (a_{ij})$  is the matrix  $A^T = (a^T_{ij}) = (a_{ji})$ .

Adjacency lists are usually preferable for sparse matrices, where  $|E| \ll |V|^2$  [8], because they occupy less space, as they do not use any space for edges that are not present. Respectively, adjacency matrices are preferred when the graph is dense and  $|E| \simeq |V|^2$ . Because each entry of matrix requires one bit, they can be represented in a compact way occupying  $|V|^2/8$  bytes. The adjacency matrix requires  $\Theta(V^2)$  memory, independent of the number of the edges in the graph, while the adjacency list requires  $\Theta(V + E)$  memory. Although the adjacency list representation is asymptotically at least as efficient as the adjacency matrix representation, the simplicity of an adjacency matrix may make it preferable when graphs are reasonably small.

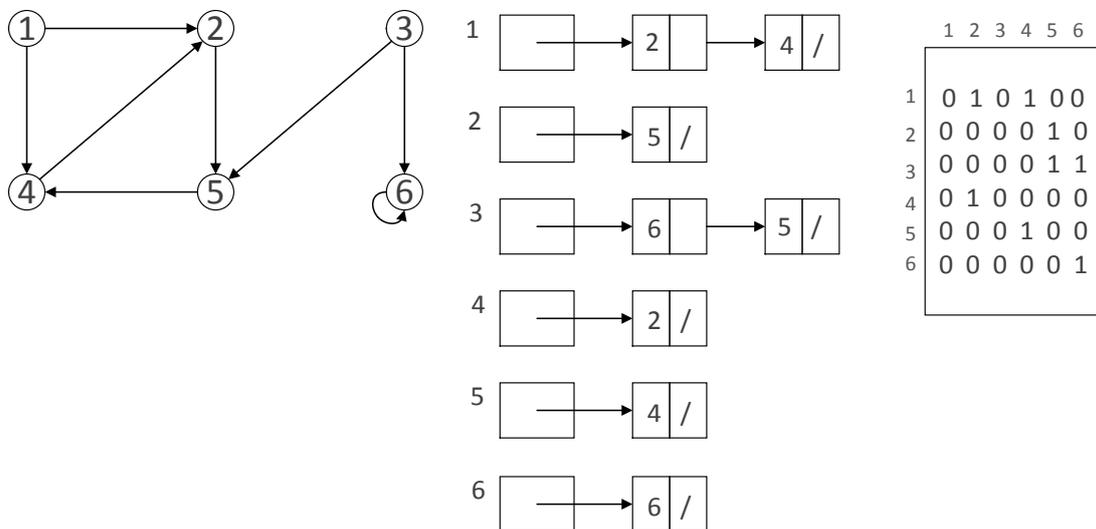


Figure 7 Adjacent list and Adjacent matrix of a graph

### 2.3.2 Definitions

Some important definitions and metrics that are used in network analysis are:

- path: an alternating sequence of vertices and edges, beginning and ending with a vertex, where each vertex is incident to both the edge that precedes it and the edge

that follows it in the sequence. The length of a path is the number of edges traversed.

- degree: number of edges incident to the vertex. The degree is not necessarily equal to the number of vertices adjacent to a vertex, since there may be more than one edge between any two vertices. Such a graph is called a multigraph. In directed graphs, there is in-degree and out-degree for every vertex, which are the numbers of incoming and outgoing edges respectively.
- geodesic distance: the distance between two vertices in a graph is the number of edges in a shortest path (also called a graph geodesic) connecting them.
- diameter: the greatest distance between any pair of vertices; it is equal to the length of the longest shortest path between any two vertices.
- clustering coefficient: the probability that a connected triple of nodes is actually a triangle. It describes the tendency to form clusters - fully connected subgraphs in a graph and is a measure of the likelihood that two associates of a node are associates themselves.

### 2.3.3 Metrics

If the structure of a network is known, we can calculate from it a variety of useful quantities or measures that capture particular features of the network topology. In this thesis are analyzed some of these measures.

Centrality is widely-used for measuring the importance of nodes within a graph. For instance, who are the most well-connected people in a social network. In general, the concept of centrality has helped in the understanding of various kinds of networks by researchers from computer science, network science etc. In addition, centrality has typically been studied for small graphs. However, in the past few years, centralities have played important role in the very large graphs, too. Many of these networks reach billions of nodes and edges requiring terabytes of storage.

Moreover, there are some challenges for measuring centrality in very large graphs. First, some definitions of centrality have inherently high computational complexity. For example, shortest-path or random walk betweenness have complexity at least  $O(n^3)$  where  $n$  is the number of nodes in a graph. Furthermore, some of the faster estimation algorithms require operations, which are not classified to parallelization, such as all sources breadth-first search. Second, even if a centrality measure is designed in a way that avoids expensive or non-parallelizable operations, developing algorithms that

are both efficient, scalable and accurate is not straightforward. Clever approximations may need to be employed, in order to achieve these goals.

Finally, there are two types of centralities: Geodesic-based – Closeness, Shortest Path Betweenness and Degree-based – Degree, Power Community Index, PageRank.

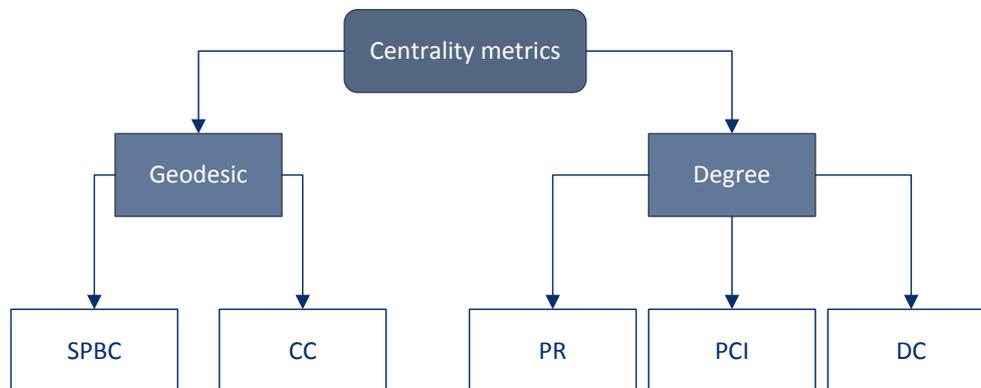


Figure 8 Two basic categories of centralities

## 2.4 Related Work

Some of the related works on processing large-scale, complex networks on Hadoop are computing social measures in large graphs. In more detail, processing large graphs for cliques with limited memory, k-core maintenance in large dynamic graphs, k-core decomposition, truss decomposition. Moreover, the centrality of a node in a network is interpreted as the importance of the node. Many centrality measures have been proposed based on how the importance is defined. For instance, influence-based centralities. The influence-based community partition for social networks is another issue of the network analysis, too. All these combined with the parallel graph mining using Hadoop and the distributed programming framework for processing web-scale data.

Moreover, another branch that depends on large scale data analysis is bioinformatics. In bioinformatics, dealing with neural networks, with purpose of obtaining genetic information for the human organism.

# Chapter 3. Implementations

## 3.1 Degree Centrality

**D**egree centrality has a very simple and intuitive definition: it is the number of neighbors of a node. Despite, or perhaps because of its simplicity, it is very popular and used extensively. Not surprisingly, it is also the easiest to compute.

The major limitation of degree based centrality is that it only captures the local information of a node. In many applications, we need more informative measures that can further distinguish among nodes that have almost equally low degrees, or almost equally high degrees.

In more particular, for undirected graphs degree centrality of a node is the number of its links, but for directed graphs degree centrality is separated to the number of in links and the number of out links.

The general implementation process of this algorithm with the help of MapReduce jobs is as follows:

- First, as input is given a txt file with the following format:

```
*VerticeID:neighbor1, . . . ,neighborN  
1:2,3  
2:1,3
```

- the Mapper takes as input the file and separates the indexes of each node with the nodes to which it is affiliated, by the separator ':'. The nodes to which is connected each node are its neighbors. Finally, mapper sets the output, as a key each id and as a value its neighbors.
- Subsequently, the Reducer receives as input the output of Mapper and the essential function of reducer is to separate the neighbors by the separator ',' and set up for each different id, a list of its neighbors. Finally, as output gives key for the id of each node and value for the size of the list that it is the number of links of the current node, hence the DC of each node, for an undirected graph.

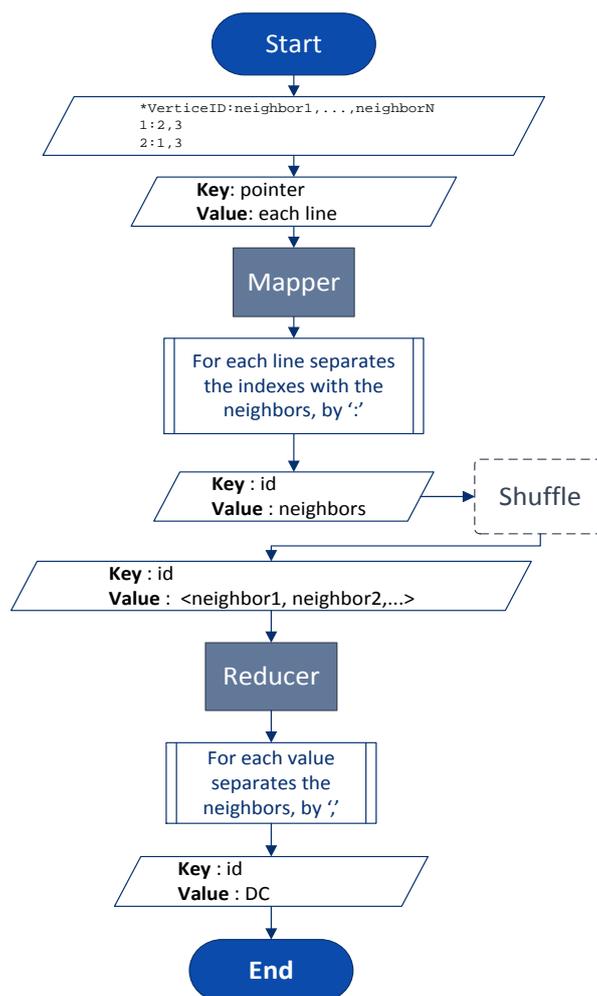
PSEUDOCODE:

## Class Mapper

```
public void map(LongWritable key, Text value, Context context){  
  
    String [] inputLine = value.split(":");  
    nodeID.set(inputLine[0]);  
    neighbours.set(inputLine[1]);  
    context.write(nodeID, neighbours);  
  
}
```

## Class Reducer

```
public void reduce(Text key, Iterable<Text> values, Context context){  
  
    for each v in values {  
        String [] tokens = v.split(",");  
        List<> adjacencies.add(tokens);  
    }  
    context.write(key, adjacencies.size());  
  
}
```



Flowchart 1 Degree Centrality process

## 3.2 Power Community Index

This is a new centrality metric of the nodes that specifically applies to sensor networks where there are more efficient algorithms, and not affected by individual nodes and considerably more informative than computing the DC degree centrality in a graph.

The  $\mu$ -Power Community Index of a node  $V$  is equal to  $k$ , where there are any more nodes  $\mu * k$  of the neighbors of  $V$ , with degree (DC) greater than or equal to  $k$ . In more particular, the other neighbors of the node should have a degree of less than or equal to  $k$ . For convenience, we consider  $\mu = 1$ .

The general implementation process of this algorithm with the help of MapReduce jobs is as follows:

- First, as input is given a txt file with the following format:

```
*VerticeID:neighbor1, . . . ,neighborN
1:2,3
2:1,3
```

Precondition of this algorithm is find first the DC of each node, therefore we follow the same procedure as described earlier regarding the metric Degree Centrality. Then, having successfully completed the first job, starts the second entirely on the `pciMapper ()` and `pciReducer ()`.

- the `pciMapper` takes as input the output file of `dcReducer`. Having already calculated the DC for each node and created a `HashMap` with key the id and value the DC and having for each node a list of its neighbors too, `pciMapper` outputs for each id a list consisting of DC, of its neighbors. As the key is the id and the value the DC list of its neighbors.
- the `pciReducer` takes as input the output file of `pciMapper`. Having the DC list of neighboring nodes for each node of the graph, we check if the number of neighboring nodes with  $DC > k$ , is less than or equal to  $k$ , then we set the PCI of the current node, the  $k$ . Giving as a final output for the key the id of the node and for value the PCI.

PSEUDOCODE:

### Class `DCMapper`

```
public void map(LongWritable key, Text value, Context context){
    String [] inputLine = value.split(":");
    nodeID.set(inputLine[0]);
```

```

        neighbours.set(inputLine[1]);
        context.write(nodeID, neighbours);
    }

```

### Class DCReducer

```

public void reduce(Text key, Iterable<Text> values, Context context){
    for each v in values.split(",") {
        List<> adjacencies.add(v);
    }
    Node nodeInfo = new Node(key, adjacencies.size);
    context.write(nodeInfo, adjacencies);
}

```

### Class PCIMapper

```

public void map(Node node, ArrayListWritable<> value, Context context){
    MapWritable dcList;
    //toReturn is the list of neighbor's DC
    StringBuilder toReturn;
    //value is a list with neighbors
    for (int i = 0; i<value.size(); i++){
        Writable list = dcList.get(value.get(i))
        Append(toReturn,list.toString());
        if (i + 1 < value.size)
            Append(toReturn, ",");
    }
    context.write(nodeID, toReturn);
}

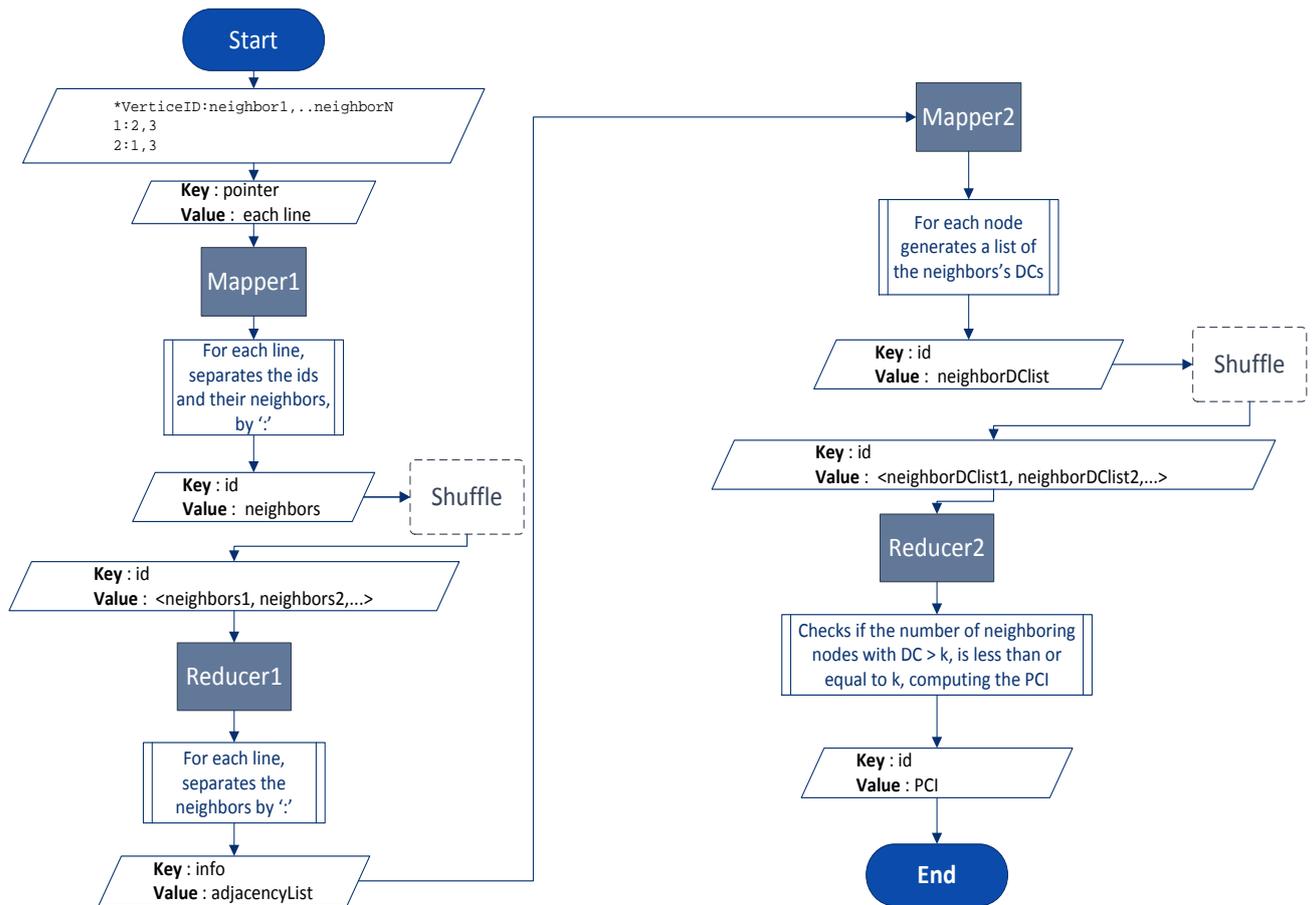
```

### Class PCIReducer

```

public void reduce(Text key, Iterable<Text> value, Context context){
    List<> neighbourDcList;
    int PCI = 0;
    for (k=1 to neighbourDcList.size()) {
        for each m from neighbourDcList{
            if (m > k)
                count++;
        }
        if (count <= k) {
            PCI = k;
            break;
        }
    }
    context.write(key, PCI);
}

```



Flowchart 2 Power Community Index process

### 3.3 Closeness Centrality

Closeness centrality is the most popular diameter-based centrality measure. While degree centrality considers only one-step neighbors, closeness centrality considers all nodes in the graph, and gives high score to nodes which have short average distances to all the other nodes.

Closeness of a node is typically defined as the inverse of the average over the shortest distances to all other nodes; to simplify formulas we omit the inverse. Exact computation requires an all-pairs shortest paths algorithm. Unfortunately, this operation requires  $O(n^3)$  time. For the billion-scale graphs we consider in this work, computing closeness centrality is prohibitively expensive.

To implement the Closeness centrality, BFS algorithm was used for the pervasion of the graph and Dijkstra algorithm for finding the shortest path between any two nodes in the graph. The fact that we have to find the minimum distances from each node to source, forcing us to use iteration and thus counters of mapreduce library. Each iteration also implies a different node source; therefore, the same process is performed as many times as the  $n$ , number of nodes of the graph.

The general implementation process of this algorithm with the help of MapReduce jobs is as follows:

- First, as input is given a txt file with the following format:

```
*VerticeID:neighbor1,...,neighborN
1:2,3
2:1,3
```

- This file is taken as input by InitMapper which aims to give as output a txt file with the following format:

```
*VerticeID neighbor1,...,neighborN|distance|status|cc
1 5|2147483647|UNVISITED|0
2 5|2147483647|UNVISITED|0
3 4,2|2147483647|UNVISITED|0
```

So as to start the BFS algorithm process. Specifically, the BFS algorithm assumes that the source node is initialized as VISITED, while all adjacent nodes can now be processed. At the end of processing, they become VISITED, while the source node is now become PROCESSED. When there is no one else VISITED node, then consider that the pervasion of the graph is over and we have reached to the final node of each current iteration.

- the InitMapperTwo takes as input the txt file produced by the InitMapper and aims to set the start BFS algorithm, i.e., to set source node as VISITED

```
3 5|2147483647|UNVISITED|0
2 5|2147483647|UNVISITED|0
1 4,2|0|VISITED|0
```

τον source κόμβο ως VISITED and determine the distance 0, because obviously the distance from starting node is constant and equal to 0. The key is the id of each node and the value is all of the remaining information, that is properly processed.

- the SearchMapper takes as input the output of InitMappertwo and aims to verify if it is accessible by the current node, i.e., if it is VISITED, then access all of its neighbors and set them as VISITED, add the distance, in more particular, the weight of each node if we refer to weighted graphs or 1 if we refer to unweighted graphs and in the end, as long as we finish the access of neighbors, we make current node PROCESSED. Giving as an expense as key the id of each node and as value the rest information.
- SearchReducer takes as input the output of SearchMapper and aims to keep always, a minimum distance value; giving as output the same format with SearchMapper.

- the FinalMapper takes as input the output of SearchReducer and aims to keep through all the information, only the id of each node and the minimum distance. As a key set the id of each node and as a value the minimum distance from the current source node.
- the FinalReducer takes as input the output of FinalMapper and intends to calculate the sum of all minimum distances of all other nodes for each node and finally, the inverse of this number is the CC. Giving key as the id of each node and the value as CC.

PSEUDOCODE:

### Class SearchMapper

```
public void map(LongWritable key, Text value, Context context, Node inNode){
    if (inNode.getStatus() == Node.State.VISITED) {
        // for all the adjacent nodes of the VISITED node
        for (neighbor in inNode.getEdges()) {
            Node adjacentNode = new Node();
            adjacentNode.setId(neighbor);
            // set the id of the node
            // for weighted graph or for unweighted the weight is 1
            adjacentNode.setDistance(inNode.getDistance() +
            inNode.getWeight());
            // set the status of the current node to be VISITED
            adjacentNode.setStatus(Node.State.VISITED);
            context.write(adjacentNode.getId(),
            adjacentNode.getNodeInfo());
        }
        // this node is done, set status to be PROCESSED
        inNode.setStatus(Node.State.PROCESSED);
    }
    context.write(inNode.getId(), inNode.getNodeInfo());
}
```

### Class SearchReducer

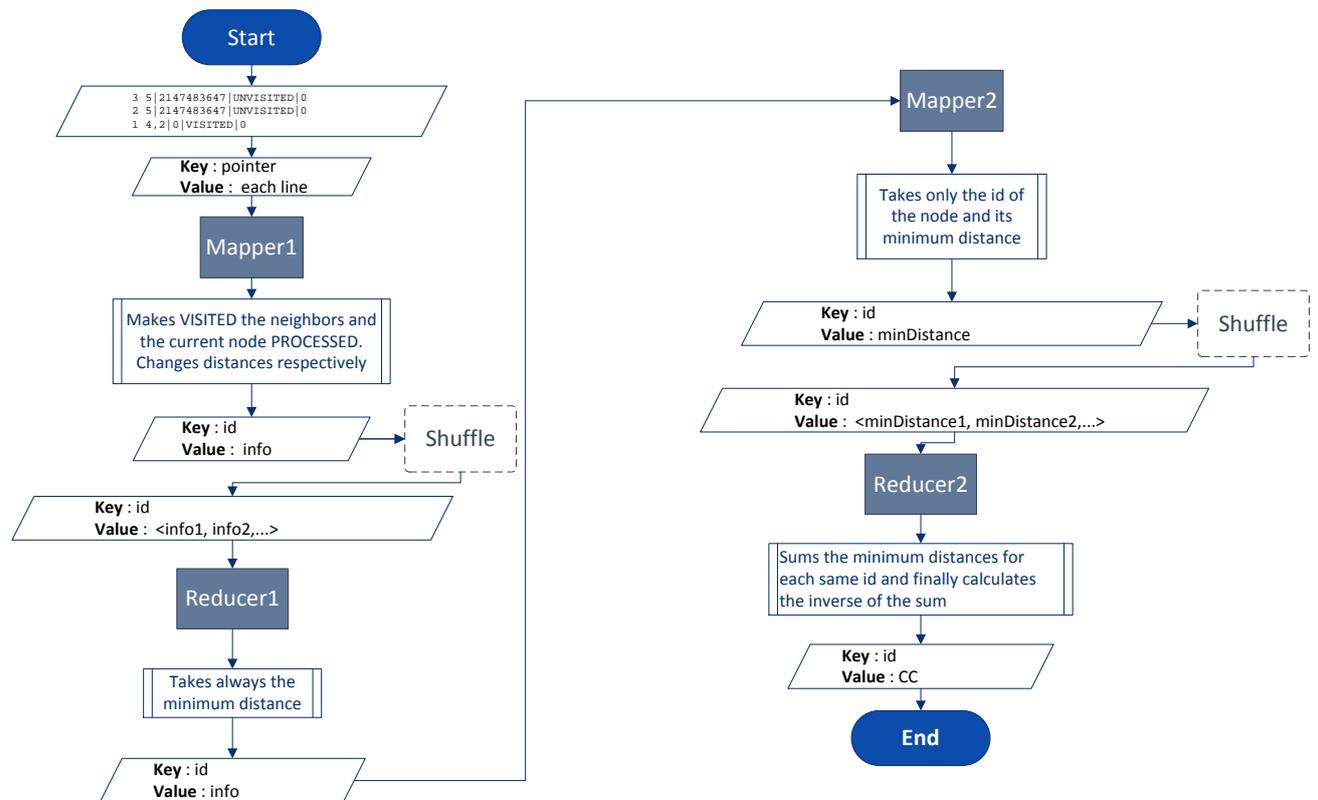
```
public Node reduce(Text key, Iterable<Text> values, Context context, Node
outNode){
    int sum = 0;
    //set the node id as the key
    outNode.setId(key);
    //for all the values corresponding to a particular node id
    for each v in values {
        Node inNode = new Node(key + "\t" + v);
        if (inNode.getEdges().size() > 0)
            outNode.setEdges(inNode.getEdges());
        // Save the minimum distance
        if (inNode.getDistance() < outNode.getDistance())
            outNode.setDistance(inNode.getDistance());
        // Save the VISITED between UNVISITED and VISITED
        // Save the PROCESSED between VISITED and PROCESSED
        if (inNode.getStatus().ordinal() >
        outNode.getStatus().ordinal())
            outNode.setStatus(inNode.getStatus());
    }
    context.write(key, outNode.getNodeInfo());
    return outNode;
}
```

## Class FinalMapper

```
public void map(LongWritable key, Text value, Context context){
    String[] inputLine = value.split("\\t");
    String[] tokens = inputLine[1].split("\\|");
    // key is id and value is minimum distance
    context.write(inputLine[0], tokens[1]);
}
}
```

## Class FinalReducer

```
public void reduce(Text key, Iterable<Text> values, Context context){
    for each v in values{
        sum = sum + Integer.parseInt(v.toString());
        //if it's the source node, get the id
        if (Integer.parseInt(v.toString()) == 0)
            id.set(key);
        //if it's the last node of the graph, set the closeness
        if (N == Integer.parseInt(key.toString())) {
            closeness.set(String.valueOf(1/sum));
            list.add(closeness);
        }
    }
    //if we are at the end of the list, set the final output key & value
    if (N == list.size()) {
        for (int i = 0; i<list.size(); i++) {
            context.write((i+1), list.get(i));
        }
    }
}
}
```



Flowchart 3 Closeness Centrality process

### 3.4 Shortest Path Betweenness Centrality

**B**etweenness centrality is the most common and representative flow-based measure. In general, the betweenness centrality of a node  $V$  is the number of times a walker visits node  $V$ , averaged over all possible starting and ending nodes. Different types of walks lead to different definitions for betweenness centrality. In Freeman betweenness, the walks always follow the shortest path from starting to ending node. In Newman's betweenness, the walks are absorbing random walks. Both of these popular definitions require prohibitively expensive computations: the best algorithm for shortest-path betweenness has  $O(n^2 \log n)$  complexity, while the best for Newman's betweenness has  $O(m \cdot n^2)$  complexity.

To implement this algorithm BFS algorithm was used for the permeation of the graph and Dijkstra algorithm for finding the shortest path between any two nodes in the graph. The fact that we have to find the minimum distance of 2 any node couples to source all the graph nodes each time while store Mr. different equally spaced possible minimum paths, forcing us to use repetition and thus counters the implementation of algorithm. Each iteration also implies a different node source; therefore, the same process is performed as many times as there are the  $n$  nodes of the graph.

The general implementation process of this algorithm with the help of MapReduce jobs is as follows:

- First, as input is given a txt file with the following format:

```
*VerticeID:neighbor1,...,neighborN
1:2,3
2:1,3
```

- This file is taken as input by InitMapper which aims to give as output a txt file

```
*VerticeID neighbor1,...,neighborN|distance|status|cc
1 5|2147483647|UNVISITED|0
2 5|2147483647|UNVISITED|0
3 4,2|2147483647|UNVISITED|0
```

with the following format:

So as to start the BFS algorithm process. Specifically, the BFS algorithm assumes that the source node is initialized as VISITED, while all adjacent nodes can now be processed. At the end of processing, they become VISITED, while the source node is now become PROCESSED. When there is no one else VISITED node, then consider that

the pervasion of the graph is over and we have reached to the final node of each current iteration.

- the `InitMapperTwo` takes as input the txt file produced by the `InitMapper` and aims to set the start BFS algorithm, i.e., to set source node as VISITED

3	5		2147483647		UNVISITED		0
2	5		2147483647		UNVISITED		0
1	4,2		0		VISITED		0

τον source κόμβο ως VISITED and determine the distance 0, because obviously the distance from starting node is constant and equal to 0. The key is the id of each node and the value is all of the remaining information, that is properly processed.

- the `SearchMapper` takes as input the output of `InitMappertwo` and aims to verify if it is accessible by the current node, i.e. if VISITED, to access all the neighbors, put and them VISITED, add the distance the weight of each node if refer to weighted graphs or 1 if we refer to unweighted graphs and end as long as you finish the access of neighbors to appoint current node as PROCESSED. Here are perceived as necessary to the store somehow all possible paths soon because SPBC varies and depends directly. More specifically, in cases with equivalent minimum paths added to each if located intermediate node we ask each time, 1 for the number of equivalent paths rather than one as in the other cases. Giving as an expense for the key id of each node and for value the rest information.
- the `SearchReducer` takes as input the output of `SearchMapper` and aims to always keep a minimum distance value giving as output the same format with `SearchMapper`.
- the `FinalMapper` takes as input the output of `SearchReducer` and aims if we have all the possible paths for each pair to add together the times we meet in between the current node in these lists are all possible short paths between the source and destination, allowing as the key id of each node and value that sum.
- the `FinalReducer` receives as input the output of `FinalMapper` and intends to summation of all the individual impressions of each node for all replicates i.e. each node that becomes source. This sum is in fact the SPBC node. Giving key as the id of each node and the value SPBC.

PSEUDOCODE:

### Class `SearchMapper`

```
public void map(LongWritable key, Text value, Context context, Node inNode){
    if (inNode.getStatus() == Node.State.VISITED) {
        //for all the adjacent nodes of the VISIED node
        for each neighbor in inNode.getEdges(){
```

```

        adjacentNode.set(all the info);
        //id, VISITED, distance
        ArrayList<> list;
        //list with the nodes of the path
        adjacentNode.setShortest_Path(list);
//if the parent is not exist in the list from adjacents and add the parent
        check(adjacentNode);
        context.write(adjacentNode.getId(),
        adjacentNode.getNodeInfo());
        inNode.setShortest_Path(list);
// if the parent is not exist in the list from current node and add the
parent
        check(inNode);
    }
    // this node is done, set it PROCESSED
    inNode.setStatus(Node.State.PROCESSED);
}
context.write(inNode.getId(), inNode.getNodeInfo());
}
}

```

## Class SearchReducer

```

public Node reduce(Text key, Iterable<Text> values, Context context, Node
outNode){
    int sum = 0;
    //set the node id as the key
    outNode.setId(key);
    //for all the values corresponding to a particular node id
    for each v in values {
        Node inNode = new Node(key + "\t" + v);
        if (inNode.getEdges().size() > 0)
            outNode.setEdges(inNode.getEdges());
        // Save the minimum distance
        if (inNode.getDistance() <= outNode.getDistance()){
            if (inNode.getDistance == outNode.getDistance)
                outNode.setShortest_Path(inNodeSP,
outNodeSP);
            outNode.setDistance(inNode.getDistance());
        }
        // Save the VISITED between UNVISITED and VISITED
        // Save the PROCESSED between VISITED and PROCESSED
        if (inNode.getStatus().ordinal() >
outNode.getStatus().ordinal())
            outNode.setStatus(inNode.getStatus());
        //same source but destination?
        for each inNodeSP
            checkDestination();
        checkDuplicates();
    }
    context.write(key, outNode.getNodeInfo());
    return outNode;
}
}

```

## Class FinalMapper

```

public void map(LongWritable key, Text value, Context context){
    for (int k = 1; k<=N;k++) {
        String element = String.valueOf(k);
        //the rest info from the inputLine
        //distance,Status etc
        String[] tokens;
        //if there is one shortest path
        if (tokens.length == 4) {
            ArrayList<String> ar; //list with the nodes of the path
            //check if the element is in the list
            //but not at the start and in the end of the list
            //and if it's true sum++

```

```

        checkIfContainsK(ar,element);
    } //if there are more than one shortest paths
    else if (tokens.length > 4) {
        double pathNum;
        for each path from paths {
            ArrayList<String> ar2 = new ArrayList<>();
            //check if the element is in the list
            //but not at the start and in the end of the list
            //and if it's true sum=sum + 1/pathNum
            checkIfContainsK(ar2,element)
        }
    }
    context.write(k, sum);
    sum = 0;
}
}
}

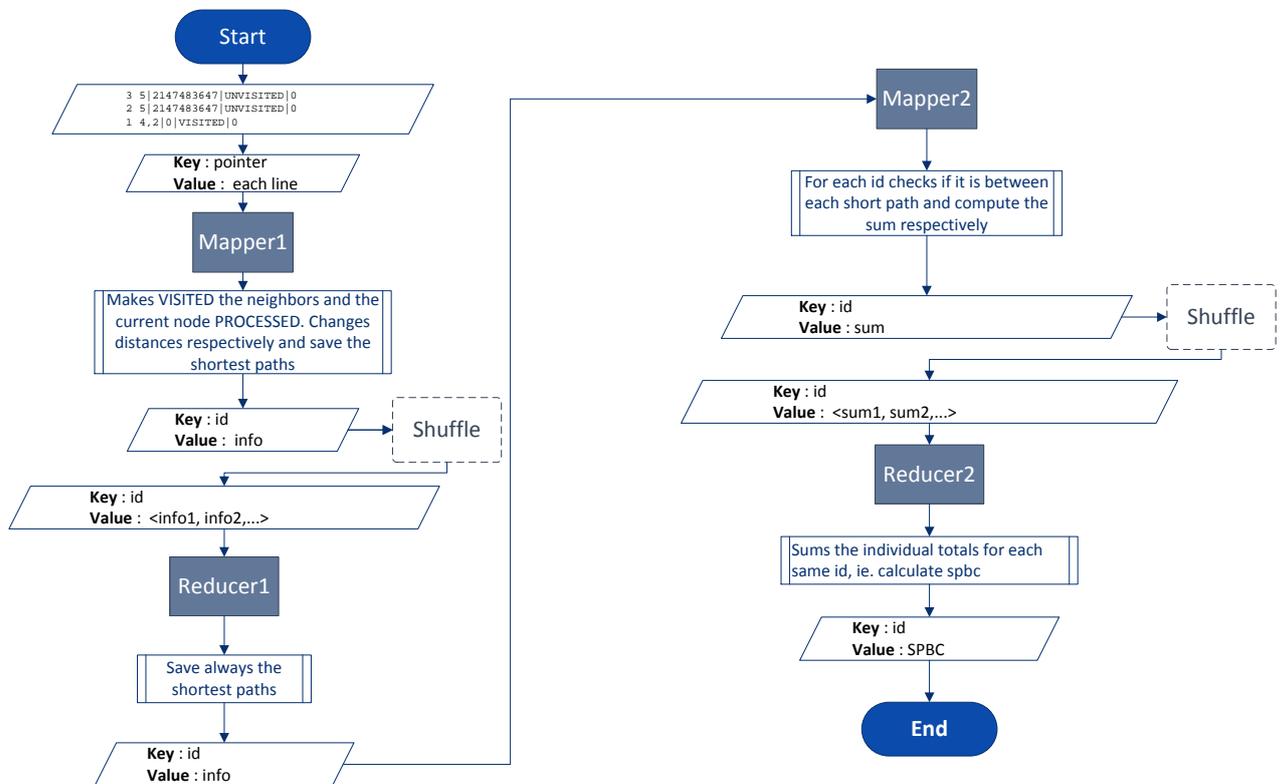
```

### Class FinalReducer

```

public void reduce(Text key, Iterable<Text> values, Context context){
    double sum = 0;
    Text spbc = new Text();
    // for directed graphs
    Text spbc2 = new Text();
    // for undirected graphs
    for each v in values {
        //id is initialized to 1
        //Sum all the spbc from all the iterations
        //first for id = 1, then for id = 2, etc.
        if (key.equals(id))
            sum = sum + (double) v;
    }
    context.write(key, sum + " " + sum/2); id++;
    // for the next iteration
}
}

```



Flowchart 4 Shortest Path Betweenness Centrality process

### 3.5 Page Rank

This algorithm is perhaps the most popular centrality metric of a node in a graph. More specifically, the PageRank of a node is the sum of the quotient of incoming PageRank nodes, to the plurality of outgoing nodes. The formula is:

$$PR(i) = 1 - d + d * (\sum_{j \rightarrow i} PR(j) / |j|)$$

Such as,  $d = 0.85$

j: number of outgoing links

The general implementation process of this algorithm with the help of MapReduce jobs is as follows:

- First, as input is given a txt file with the following format :

```
*VerticeID:neighbor1, ..., neighborN
1:2,3
2:1,3
```

- Subsequently, IniMapper simply outputs a txt file in format:

```
*VerticeID neighbor1 ... neighborN PR
1 2 3 0.166667
2 1 3 0.166667
```

- We insert the initial value of PageRank of each node as 0.166667 arbitrarily.
- the PageRankMapper takes as input the output of InitMapper and aims to separate the current node to the other adjacent to each line of the input file for each outgoing node of current starting with the initial value of PageRank divide by the number of neighboring nodes having each node. Giving as an expense for the key id of each adjacent node for value and a current value that will be the end of the PageRank value and end of each file key as the parent node and value nodes children that their outgoing links.
- the PageRankReducer receives as input the output of PageRankMapper and aims to add all of the individual values from the input file, and finally adding the term  $1-d$  and multiply the sum by  $d$  agent giving as output the same format with the PageRankMapper, i.e., the key id of each node and value as their neighboring nodes and the renewed PageRank value.

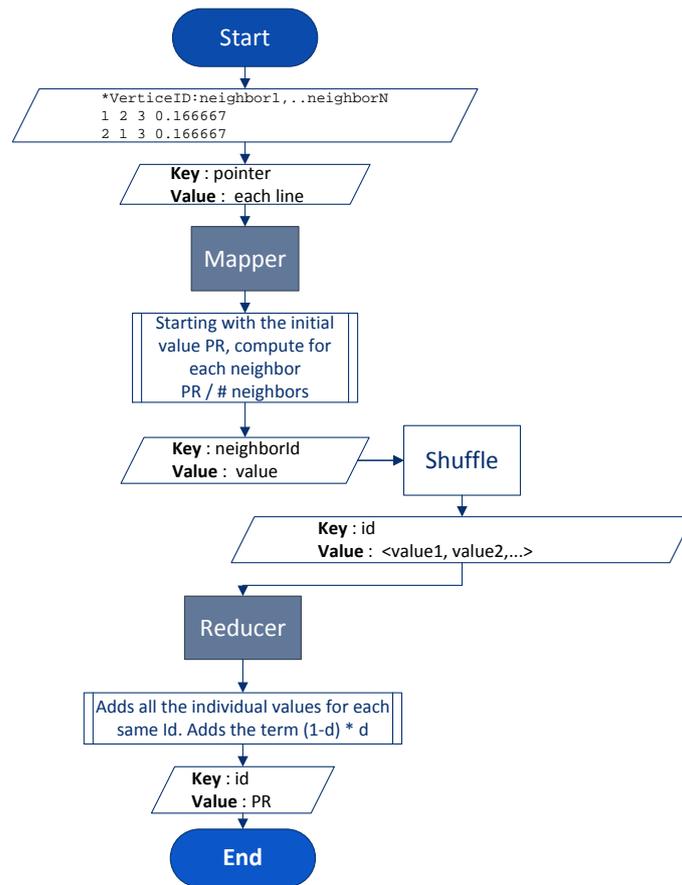
PSEUDOCODE:

### Class PageRankMapper

```
public void map(LongWritable key, Text value, Context context){
    //val = PR / numOfChildren or outlinks
    String valString;
    String outgoing = "";
    for each c in children {
        page.set(c);
        context.write(page, valString);
    }
    for each c in children
        outgoing += (c + " ");
    page.set(parent);
    context.write(page, outgoing);
}
```

### Class PageRankReducer

```
public void reduce(Text key, Iterable<Text> values, Context context){
    String outgoing = "";
    double trans;
    double sum = 0.0;
    for each v in values{
        if(v.startsWith("0")) {
            //trans = parseDouble(line)
            sum += trans;
        } else outgoing = v;
    }
    sum = ((1-0.85) + 0.85*(sum));
    outgoing += Double.toString(sum);
    result = new Text(outgoing);
    context.write(key, result);
}
```



*Flowchart 5 PageRank process*

## Chapter 4. Experiments and Results

In this section, it is present the experimental evaluation of each algorithm. The experiments were run on a computer with the following characteristics:

Core i7 4770K 3.9 GHz
24 GB DDR3 RAM
128GB SSD
OS Ubuntu 16.04 LTS

*Table 1 Characteristics of computing system*

For each algorithm, it is used as input 3 graphs of different sizes. These are described in Table 2. There is a wide collection of real graphs in [7]. The scope of this thesis is to show the implementation of algorithms in an environment that supports parallel processing, like Hadoop. For this reason, we will not deal with performance and accuracy of algorithms.

NETWORK 1	NETWORK 2	NETWORK 3
100 nodes	500 nodes	1000 nodes
200 edges	1500 edges	5000 edges
Undirected	Undirected	Undirected

*Table 2 Characteristics of network experiments*

Moreover, in this section there are analyzed two different approaches. On the one hand, the relationship of the algorithms with the size of complex networks, in other words, scalability w.r.t. complex net size, and on the other, the relationship between them, based on the correlation of the results, i.e., ranking correlation

We present some simple experiments, that are based on the graph that is shown in Figure 9. The outputs of the simple examples and the screenshots of the real network experiments are shown in Appendix. Subsequently, we explain the Kendall distance in which the second approach is based on and finally, we show some graphs on real networks that we have ran.

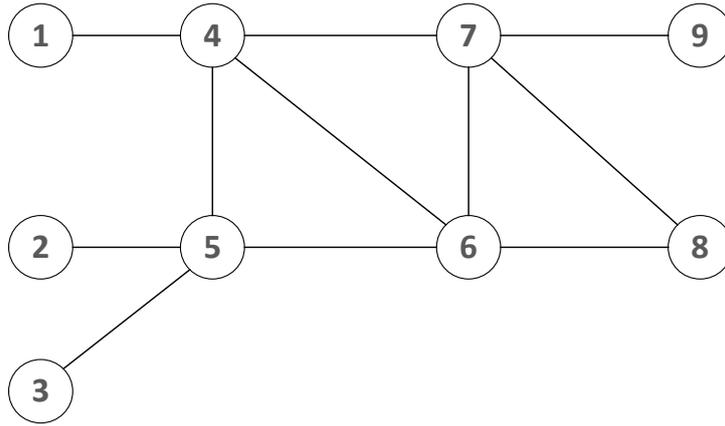


Figure 9 Example of a simple undirected graph

The Kendall tau rank distance is a metric that counts the number of pairwise disagreements between two ranking lists. The larger the distance, the more dissimilar the two lists are. Kendall tau distance is also called bubble-sort distance since it is equivalent to the number of swaps that the bubble sort algorithm would make to place one list in the same order as the other list. The Kendall tau distance was created by Maurice Kendall.

First of all, for the graph on Figure 9, we create the table 3, in other words, we run and compute each algorithm so as to find out the rankings/outputs.

	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>
<b>DC</b>	3	3	3	1	1	1	1	2	3
<b>PCI</b>	3	3	3	1	2	1	2	2	3
<b>CC</b>	5	6	6	1	2	1	3	4	7
<b>SPBC</b>	5	5	5	2	1	4	3	5	5
<b>PR</b>	7	6	6	3	1	1	2	5	8

Table 3 Rankings

In statistics, the Kendall rank correlation coefficient, commonly referred to as Kendall's tau coefficient (after the Greek letter  $\tau$ ), is a statistic used to measure the ordinal association between two measured quantities. It is a measure of rank correlation, too, i.e., the similarity of the orderings of the data when ranked by each of the quantities. Intuitively, the Kendall correlation between two variables will be high when observations have a similar (or identical for a correlation of 1) rank between the two variables, and low when observations have a dissimilar (or fully different for a correlation of -1) rank between the two variables.

$$\tau = \frac{(\text{numOfConcordant}) - (\text{numOfDiscordant})}{n(n - 1)/2}$$

In accordance with the above formula, we can now calculate on the following table the  $\tau$  coefficient.

	<b>DC</b>	<b>PCI</b>	<b>CC</b>	<b>SPBC</b>	<b>PR</b>
<b>DC</b>	1	1	1	11	
<b>PCI</b>	1	1	1	0.833	0.777
<b>CC</b>	1	1	1	0.833	0.666
<b>SPBC</b>	1	0.833	0.833	1	0.944
<b>PR</b>	1	0.777	0.666	0.944	1

*Table 4 Kendall tau coefficient ( $\tau$ )*

The correlation of the algorithms from max to min, is shown in the following table:

<b>Metrics</b>	<b>Kendall's tau coefficient (<math>\tau</math>)</b>
{DC, #}, {CC, PCI}	1
{PR, SPBC}	0.944
{SPBC, PCI}, {SPBC, CC}	0.833
{PR, PCI}	0.777
{PR, CC}	0.666

*Table 5 Correlation of metrics*

We observe that DC algorithm is correlated with each other algorithm. This phenomenon applies because of the small information that offers, so that all the rest algorithms have fully correlation with DC. This contrasts with efficiency at runtime, as we will see later. In addition, the maximum correlation, except DC, appears between CC and PCI, and the minimum one appears between CC and PR.

Furthermore, on Figure 10, we see that DC, PCI, PR are not affected by the size of the networks. Also, we can clearly see that SPBC and CC algorithms, because of their big complexity ( $O|n^3|$ ), όπου  $n$  = number of nodes of the graph, differ from the others at

runtime efficiency. As we expect, they are slower than the others and as the size is increasing, so the execution time get higher.

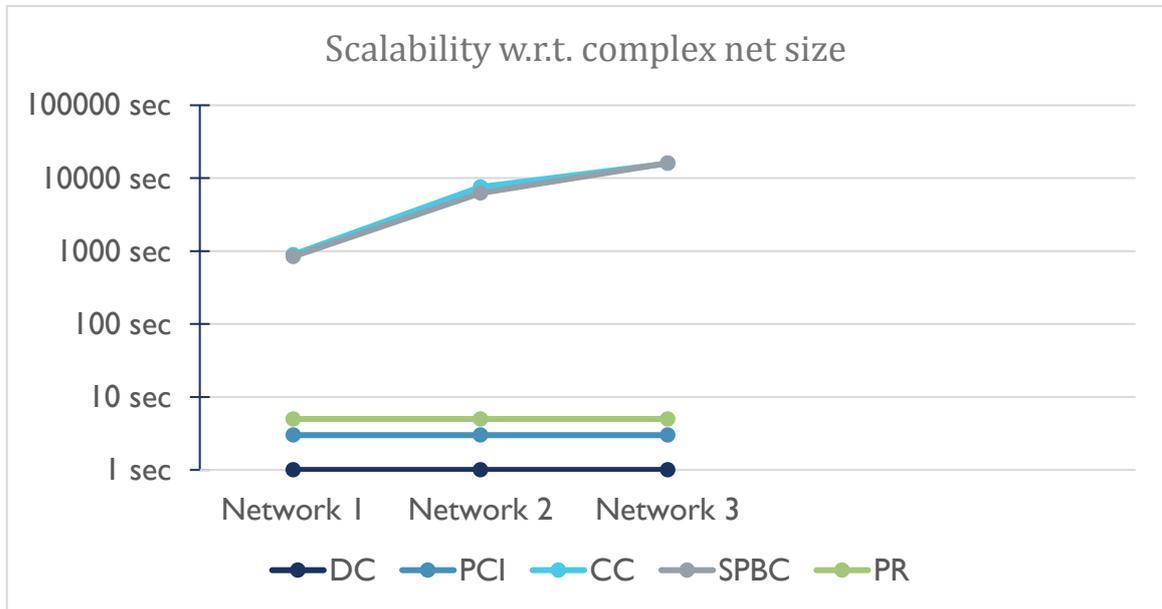


Figure 10 Scalability of metrics

The motivation of this thesis has been to develop parallel Map/Reduce algorithms in Hadoop to handle a graph mining task. This task was chosen to be node centrality, which shows the significance of each node in a given graph. Centrality has many applications in network theory according to the relations that a network expresses. Such include recommendation systems, social network analysis and the discovery of patterns in a computer network traffic.

However, sequential algorithms cannot address the problem of data that occurs in real world networks. Hadoop is a tool that offers us the possibility to easily write parallel algorithms without caring about parallelization details like the communication of machines, the distribution of data, the replication and fault tolerance. All that is needed for a programmer, is to supply the implementation of a map and a reduce function. Hadoop is a powerful tool and has already been used in graph mining algorithms like counting triangles, detecting components, finding the diameter, link prediction etc.

As a subject of future study could be considered to optimize some algorithms, so as to succeed better efficiency at runtime and accuracy. There are many techniques in parallel computing that can help us to do this., some are based on hardware and some others on software. A good technique, for example is to reduce the size of the intermediate data by adopting techniques of data compression or implementations that compute matrix approximations.

Furthermore, another issue of the future study is, if it is possible, the merge of two quite correlated metrics, so as to create a tool that will provide us more information about the graphs.

# Chapter 6.

# Appendix

In this section, we present the outputs/screenshots of the experiments, for each metric.

## 6.1 Degree Centrality's Experiments

<u>INPUT FILE</u>	<u>OUTPUT FILE</u>
9:7	part-r-00000
8:6,7	2 1
7:4,6,8,9	5 4
6:4,5,7,8	8 2
5:2,3,4,6	part-r-00001
4:1,5,6,7	3 1
3:5	6 4
2:5	9 1
1:4	part-r-00002
	1 1
	4 4
	7 4

Figure 11 Output of DC

### File System Counters

```
FILE: Number of bytes read=35556
FILE: Number of bytes written=1181325
FILE: Number of read operations=0
FILE: Number of large read operations=0
FILE: Number of write operations=0
HDFS: Number of bytes read=6232
HDFS: Number of bytes written=978
HDFS: Number of read operations=42
HDFS: Number of large read operations=0
HDFS: Number of write operations=16
```

### Map-Reduce Framework

```
Map input records=100
Map output records=100
Map output bytes=1460
Map output materialized bytes=1678
Input split bytes=100
Combine input records=0
Combine output records=0
Reduce input groups=100
Reduce shuffle bytes=1678
Reduce input records=100
Reduce output records=100
Spilled Records=200
Shuffled Maps =3
Failed Shuffles=0
Merged Map outputs=3
```

```

GC time elapsed (ms)=3
Total committed heap usage (bytes)=1107296256
Shuffle Errors
  BAD_ID=0
  CONNECTION=0
  IO_ERROR=0
  WRONG_LENGTH=0
  WRONG_MAP=0
  WRONG_REDUCE=0
File Input Format Counters
  Bytes Read=1558
File Output Format Counters
  Bytes Written=492
Elapsed time is: 1s.

```

*Figure 12 Small network experiment for DC*

```

File System Counters
  FILE: Number of bytes read=101846
  FILE: Number of bytes written=1262937
  FILE: Number of read operations=0
  FILE: Number of large read operations=0
  FILE: Number of write operations=0
  HDFS: Number of bytes read=54968
  HDFS: Number of bytes written=5784
  HDFS: Number of read operations=42
  HDFS: Number of large read operations=0
  HDFS: Number of write operations=16
Map-Reduce Framework
  Map input records=500
  Map output records=500
  Map output bytes=13244
  Map output materialized bytes=14262
  Input split bytes=100
  Combine input records=0
  Combine output records=0
  Reduce input groups=500
  Reduce shuffle bytes=14262
  Reduce input records=500
  Reduce output records=500
  Spilled Records=1000
  Shuffled Maps =3
  Failed Shuffles=0
  Merged Map outputs=3
  GC time elapsed (ms)=3
  Total committed heap usage (bytes)=1105199104
Shuffle Errors
  BAD_ID=0
  CONNECTION=0
  IO_ERROR=0
  WRONG_LENGTH=0
  WRONG_MAP=0
  WRONG_REDUCE=0
File Input Format Counters
  Bytes Read=13742
File Output Format Counters
  Bytes Written=2892
Elapsed time is: 1s.

```

*Figure 13 Medium network experiment for DC*

```
File System Counters
  FILE: Number of bytes read=224092
  FILE: Number of bytes written=1446386
  FILE: Number of read operations=0
  FILE: Number of large read operations=0
  FILE: Number of write operations=0
  HDFS: Number of bytes read=175284
  HDFS: Number of bytes written=13778
  HDFS: Number of read operations=42
  HDFS: Number of large read operations=0
  HDFS: Number of write operations=16
Map-Reduce Framework
  Map input records=1000
  Map output records=1000
  Map output bytes=42823
  Map output materialized bytes=44841
  Input split bytes=101
  Combine input records=0
  Combine output records=0
  Reduce input groups=1000
  Reduce shuffle bytes=44841
  Reduce input records=1000
  Reduce output records=1000
  Spilled Records=2000
  Shuffled Maps =3
  Failed Shuffles=0
  Merged Map outputs=3
  GC time elapsed (ms)=3
  Total committed heap usage (bytes)=1119879168
Shuffle Errors
  BAD_ID=0
  CONNECTION=0
  IO_ERROR=0
  WRONG_LENGTH=0
  WRONG_MAP=0
  WRONG_REDUCE=0
File Input Format Counters
  Bytes Read=43821
File Output Format Counters
  Bytes Written=6893
Elapsed time is: 1s.
```

*Figure 14 Large network experiment for DC*

## 6.2 Power Community Index's Experiments

<u>INPUT FILE</u>	<u>OUTPUT FILE</u>
1:4	part-r-00000
2:5	2 1
3:5	5 2
4:1, 7, 5, 6	8 2
5:6, 3, 4, 2	part-r-00001
6:8, 5, 4, 7	3 1
7:8, 9, 4, 6	6 3
8:6, 7	9 1
9:7	part-r-00002
	1 1
	4 3
	7 2

*Figure 15 Output of PCI*

### File System Counters

```
FILE: Number of bytes read=230945
FILE: Number of bytes written=3685514
FILE: Number of read operations=0
FILE: Number of large read operations=0
FILE: Number of write operations=0
HDFS: Number of bytes read=156558
HDFS: Number of bytes written=45120
HDFS: Number of read operations=342
HDFS: Number of large read operations=0
HDFS: Number of write operations=90
```

### Map-Reduce Framework

```
Map input records=100
Map output records=100
Map output bytes=1092
Map output materialized bytes=1346
Input split bytes=312
Combine input records=0
Combine output records=0
Reduce input groups=100
Reduce shuffle bytes=1346
Reduce input records=100
Reduce output records=100
Spilled Records=200
Shuffled Maps =9
Failed Shuffles=0
Merged Map outputs=9
GC time elapsed (ms)=5
Total committed heap usage (bytes)=2884108288
```

### Shuffle Errors

```
BAD_ID=0
CONNECTION=0
IO_ERROR=0
WRONG_LENGTH=0
WRONG_MAP=0
WRONG_REDUCE=0
```

```
File Input Format Counters
  Bytes Read=7357
File Output Format Counters
  Bytes Written=492
Elapsed time is: 3s.
```

*Figure 16 Small network experiment for PCI*

```
File System Counters
  FILE: Number of bytes read=471552
  FILE: Number of bytes written=3898986
  FILE: Number of read operations=0
  FILE: Number of large read operations=0
  FILE: Number of write operations=0
  HDFS: Number of bytes read=875670
  HDFS: Number of bytes written=243726
  HDFS: Number of read operations=342
  HDFS: Number of large read operations=0
  HDFS: Number of write operations=96
Map-Reduce Framework
  Map input records=500
  Map output records=500
  Map output bytes=7892
  Map output materialized bytes=8946
  Input split bytes=315
  Combine input records=0
  Combine output records=0
  Reduce input groups=500
  Reduce shuffle bytes=8946
  Reduce input records=500
  Reduce output records=500
  Spilled Records=1000
  Shuffled Maps =9
  Failed Shuffles=0
  Merged Map outputs=9
  GC time elapsed (ms)=4
  Total committed heap usage (bytes)=2912419840
Shuffle Errors
  BAD_ID=0
  CONNECTION=0
  IO_ERROR=0
  WRONG_LENGTH=0
  WRONG_MAP=0
  WRONG_REDUCE=0
File Input Format Counters
  Bytes Read=39657
File Output Format Counters
  Bytes Written=2892
Elapsed time is: 3s.
```

*Figure 17 Medium network experiment for PCI*

```
File System Counters
  FILE: Number of bytes read=955321
  FILE: Number of bytes written=4454899
  FILE: Number of read operations=0
  FILE: Number of large read operations=0
  FILE: Number of write operations=0
  HDFS: Number of bytes read=2166960
  HDFS: Number of bytes written=584960
```

```

HDFS: Number of read operations=342
HDFS: Number of large read operations=0
HDFS: Number of write operations=90
Map-Reduce Framework
Map input records=1000
Map output records=1000
Map output bytes=33893
Map output materialized bytes=35947
Input split bytes=312
Combine input records=0
Combine output records=0
Reduce input groups=1000
Reduce shuffle bytes=35947
Reduce input records=1000
Reduce output records=1000
Spilled Records=2000
Shuffled Maps =9
Failed Shuffles=0
Merged Map outputs=9
GC time elapsed (ms)=5
Total committed heap usage (bytes)=3318218752
Shuffle Errors
BAD_ID=0
CONNECTION=0
IO_ERROR=0
WRONG_LENGTH=0
WRONG_MAP=0
WRONG_REDUCE=0
File Input Format Counters
Bytes Read=95197
File Output Format Counters
Bytes Written=6893
Elapsed time is: 3s.

```

*Figure 18 Large network experiment for PCI*

## 6.3 Closeness Centrality's Experiments

<u>INPUT FILE</u>	<u>OUTPUT FILE</u>
9:7	1 0.05263157894736842
8:6,7	2 0.05
7:4,6,8,9	3 0.05
6:4,5,7,8	4 0.08333333333333333
5:2,3,4,6	5 0.07692307692307693
4:1,5,6,7	6 0.08333333333333333
3:5	7 0.07142857142857142
2:5	8 0.058823529411764705
1:4	9 0.047619047619047616

*Figure 19 Output of CC*

```

File System Counters
FILE: Number of bytes read=7513684531
FILE: Number of bytes written=70328444360
FILE: Number of read operations=0
FILE: Number of large read operations=0

```

```

FILE: Number of write operations=0
HDFS: Number of bytes read=730618114
HDFS: Number of bytes written=725350494
HDFS: Number of read operations=5935170
HDFS: Number of large read operations=0
HDFS: Number of write operations=2635203
Map-Reduce Framework
Map input records=10000
Map output records=10000
Map output bytes=49200
Map output materialized bytes=74600
Input split bytes=32328
Combine input records=0
Combine output records=0
Reduce input groups=100
Reduce shuffle bytes=74600
Reduce input records=10000
Reduce output records=100
Spilled Records=20000
Shuffled Maps =900
Failed Shuffles=0
Merged Map outputs=900
GC time elapsed (ms)=3864
Total committed heap usage (bytes)=161719779328
Shuffle Errors
BAD_ID=0
CONNECTION=0
IO_ERROR=0
WRONG_LENGTH=0
WRONG_MAP=0
WRONG_REDUCE=0
File Input Format Counters
Bytes Read=286000
File Output Format Counters
Bytes Written=2436
Elapsed time is: 897s.

```

*Figure 20 Small network experiment for CC*

```

File System Counters
FILE: Number of bytes read=2433954365879
FILE: Number of bytes written=5181243880516
FILE: Number of read operations=0
FILE: Number of large read operations=0
FILE: Number of write operations=0
HDFS: Number of bytes read=325853542792
HDFS: Number of bytes written=318138992353
HDFS: Number of read operations=850481604
HDFS: Number of large read operations=0
HDFS: Number of write operations=348627936
Map-Reduce Framework
Map input records=250000
Map output records=250000
Map output bytes=1446000
Map output materialized bytes=2138000
Input split bytes=431648
Combine input records=0
Combine output records=0
Reduce input groups=500
Reduce shuffle bytes=2138000

```

```

Reduce input records=250000
Reduce output records=500
Spilled Records=500000
Shuffled Maps =32000
Failed Shuffles=0
Merged Map outputs=32000
GC time elapsed (ms)=735099
Total committed heap usage (bytes)=2088738226176
Shuffle Errors
BAD_ID=0
CONNECTION=0
IO_ERROR=0
WRONG_LENGTH=0
WRONG_MAP=0
WRONG_REDUCE=0
File Input Format Counters
Bytes Read=10122000
File Output Format Counters
Bytes Written=12374
Elapsed time is: 7542s.

```

*Figure 21 Medium network experiment for CC*

## 6.4 Shortest Path Betweenness Centrality's Experiments

<u>INPUT FILE</u>	<u>OUTPUT FILE</u>
9:7	
8:6,7	1 0.0 0.0
7:4,6,8,9	2 0.0 0.0
6:4,5,7,8	3 0.0 0.0
5:2,3,4,6	4 21.166666666 10.5833333
4:1,5,6,7	5 26.0 13.0
3:5	6 12.66666666 6.33333333
2:5	7 16.166666666 8.0833333
1:4	8 0.0 0.0
	9 0.0 0.0

*Figure 22 Output of SPBC*

```

File System Counters
FILE: Number of bytes read=11545071989
FILE: Number of bytes written=75065433660
FILE: Number of read operations=0
FILE: Number of large read operations=0
FILE: Number of write operations=0
HDFS: Number of bytes read=1061388179
HDFS: Number of bytes written=1106338165
HDFS: Number of read operations=5935170
HDFS: Number of large read operations=0
HDFS: Number of write operations=2635203
Map-Reduce Framework
Map input records=10000
Map output records=1000000
Map output bytes=7130811
Map output materialized bytes=9136211

```

```

Input split bytes=32328
Combine input records=0
Combine output records=0
Reduce input groups=100
Reduce shuffle bytes=9136211
Reduce input records=1000000
Reduce output records=100
Spilled Records=2000000
Shuffled Maps =900
Failed Shuffles=0
Merged Map outputs=900
GC time elapsed (ms)=4004
Total committed heap usage (bytes)=161877590016
Shuffle Errors
  BAD_ID=0
  CONNECTION=0
  IO_ERROR=0
  WRONG_LENGTH=0
  WRONG_MAP=0
  WRONG_REDUCE=0
File Input Format Counters
  Bytes Read=662677
File Output Format Counters
  Bytes Written=4117
Elapsed time is: 845s.

```

*Figure 23 Small network experiment for SPBC*

```

File System Counters
  FILE: Number of bytes read=1807745748637
  FILE: Number of bytes written=4186599361080
  FILE: Number of read operations=0
  FILE: Number of large read operations=0
  FILE: Number of write operations=0
  HDFS: Number of bytes read=191044761418
  HDFS: Number of bytes written=202383347078
  HDFS: Number of read operations=141298539
  HDFS: Number of large read operations=0
  HDFS: Number of write operations=62717196
Map-Reduce Framework
  Map input records=250000
  Map output records=125000000
  Map output bytes=989216868
  Map output materialized bytes=1239243868
  Input split bytes=161868
  Combine input records=0
  Combine output records=0
  Reduce input groups=500
  Reduce shuffle bytes=1239243868
  Reduce input records=125000000
  Reduce output records=500
  Spilled Records=250000000
  Shuffled Maps =4500
  Failed Shuffles=0
  Merged Map outputs=4500
  GC time elapsed (ms)=169717
  Total committed heap usage (bytes)=1573436522496
Shuffle Errors
  BAD_ID=0
  CONNECTION=0

```

```

IO_ERROR=0
WRONG_LENGTH=0
WRONG_MAP=0
WRONG_REDUCE=0
File Input Format Counters
  Bytes Read=30712316
File Output Format Counters
  Bytes Written=20682
Elapsed time is: 6263s.

```

*Figure 24 Medium network experiment for SPBC*

## 6.5 Page Rank's Experiments

<u>INPUT FILE</u>	<u>OUTPUT FILE</u>
9 7 0.166667	1 4 0.27246981292382816
8 6 7 0.166667	2 5 0.28159511605468757
7 4 6 8 9 0.166667	3 5 0.28159511605468757
6 4 5 7 8 0.166667	4 1 5 6 7 0.71113865661523
5 2 3 4 6 0.166667	5 2 3 4 6 0.82172178917187
4 1 5 6 7 0.166667	6 4 5 7 8 0.65453838340429
3 5 0.166667	7 4 6 8 9 0.72278747401171
2 5 0.166667	8 6 7 0.3801697530839844
1 4 0.166667	9 7 0.26804824105468

*Figure 25 Output of PR*

```

File System Counters
  FILE: Number of bytes read=105120
  FILE: Number of bytes written=2410024
  FILE: Number of read operations=0
  FILE: Number of large read operations=0
  FILE: Number of write operations=0
  HDFS: Number of bytes read=18274
  HDFS: Number of bytes written=17735
  HDFS: Number of read operations=91
  HDFS: Number of large read operations=0
  HDFS: Number of write operations=40
Map-Reduce Framework
  Map input records=100
  Map output records=196
  Map output bytes=1346
  Map output materialized bytes=1744
  Input split bytes=116
  Combine input records=0
  Combine output records=0
  Reduce input groups=100
  Reduce shuffle bytes=1744
  Reduce input records=196
  Reduce output records=100
  Spilled Records=392
  Shuffled Maps =1
  Failed Shuffles=0
  Merged Map outputs=1
  GC time elapsed (ms)=0
  Total committed heap usage (bytes)=608174080

```

```
Shuffle Errors
  BAD_ID=0
  CONNECTION=0
  IO_ERROR=0
  WRONG_LENGTH=0
  WRONG_MAP=0
  WRONG_REDUCE=0
File Input Format Counters
  Bytes Read=2577
File Output Format Counters
  Bytes Written=2577
Elapsed time is: 5s.
```

*Figure 26 Small network experiment for PR*

```
File System Counters
  FILE: Number of bytes read=297236
  FILE: Number of bytes written=2609404
  FILE: Number of read operations=0
  FILE: Number of large read operations=0
  FILE: Number of write operations=0
  HDFS: Number of bytes read=115148
  HDFS: Number of bytes written=101973
  HDFS: Number of read operations=91
  HDFS: Number of large read operations=0
  HDFS: Number of write operations=48
Map-Reduce Framework
  Map input records=500
  Map output records=1140
  Map output bytes=9786
  Map output materialized bytes=12072
  Input split bytes=116
  Combine input records=0
  Combine output records=0
  Reduce input groups=500
  Reduce shuffle bytes=12072
  Reduce input records=1140
  Reduce output records=500
  Spilled Records=2280
  Shuffled Maps =1
  Failed Shuffles=0
  Merged Map outputs=1
  GC time elapsed (ms)=0
  Total committed heap usage (bytes)=840957952
Shuffle Errors
  BAD_ID=0
  CONNECTION=0
  IO_ERROR=0
  WRONG_LENGTH=0
  WRONG_MAP=0
  WRONG_REDUCE=0
File Input Format Counters
  Bytes Read=14309
File Output Format Counters
  Bytes Written=14309
Elapsed time is: 5s.
```

*Figure 27 Medium network experiment for PR*

```
File System Counters
  FILE: Number of bytes read=1174720
```

```
FILE: Number of bytes written=3491829
FILE: Number of read operations=0
FILE: Number of large read operations=0
FILE: Number of write operations=0
HDFS: Number of bytes read=314496
HDFS: Number of bytes written=254472
HDFS: Number of read operations=91
HDFS: Number of large read operations=0
HDFS: Number of write operations=40
Map-Reduce Framework
Map input records=1000
Map output records=1954
Map output bytes=16159
Map output materialized bytes=20073
Input split bytes=116
Combine input records=0
Combine output records=0
Reduce input groups=1000
Reduce shuffle bytes=20073
Reduce input records=1954
Reduce output records=1000
Spilled Records=3908
Shuffled Maps =1
Failed Shuffles=0
Merged Map outputs=1
GC time elapsed (ms)=0
Total committed heap usage (bytes)=1034944512
Shuffle Errors
BAD_ID=0
CONNECTION=0
IO_ERROR=0
WRONG_LENGTH=0
WRONG_MAP=0
WRONG_REDUCE=0
File Input Format Counters
Bytes Read=27618
File Output Format Counters
Bytes Written=27618
Elapsed time is: 5s.
```

*Figure 28 Large network experiment for PR*

## Chapter 7. Bibliography

- [1] <http://hadoop.apache.org>
- [2] <https://en.wikipedia.org/wiki/PageRank>
- [3] [https://en.wikipedia.org/wiki/Kendall\\_rank\\_correlation\\_coefficient](https://en.wikipedia.org/wiki/Kendall_rank_correlation_coefficient)
- [4] [https://en.wikipedia.org/wiki/Cloud\\_computing](https://en.wikipedia.org/wiki/Cloud_computing)
- [5] <https://snap.stanford.edu/data/>
- [6] M. Newman. The Structure and Function of Complex Networks. SIAM Review 45: 167-256, 2003
- [7] Peter Mell Timothy Grance, The Nist Definition of Cloud Computing, 2011
- [8] Basaras, Detecting Influential Spreaders in Complex, Dynamic Networks, 2013
- [9] Vishnu Sankar, Scalable Community Detection and Centrality Algorithms for Network Analysis, 2014
- [10] Sushant S.Khopkar, Rakesh Nagi and Alexander G.Nikolaev, An Efficient Map-Reduce Algorithm for the Incremental Computation of All-Pairs Shortest Paths in Social Networks, 2012
- [11] Nikhitha Cyril, Arun Soman, Map-based Multi-Source Message Passing Model to find Betweenness Centrality using MapReduce, 2015
- [12] Jonathan Magnusson, Social Network Analysis Utilizing Big Data Technology, 2012
- [13] Pei-Ling Chen, Chung-Kuang Chou and Ming-Syan Chen, Distributed Algorithms for k-truss Decomposition, 2014
- [14] Fragkiskos D.Malliaros, Apostolos N.Papadopoulos, Michalis Vazirgiannis, Core Decomposition in Graphs: Concepts, Algorithms and Applications, 2016
- [15] C. E. Tsourakakis, U. Kang, G. L. Miller, C. Faloutsos. Doulion: Counting triangles in massive graphs with a coin. KDD, 2009
- [16] U. Kang, Charalampos E. Tsourakakis, Christos Faloutsos, "PEGASUS": A Peta-Scale Graph Mining System," icdm, pp.229-238, 2009 Ninth IEEE International Conference on Data Mining, 2009

- [17] J. Bank and B. Cole, Calculating the Jaccard Similarity Coefficient with MapReduce for Entity Pairs in Wikipedia. December 2008.
- [18] Sabeur Aridhi, Vincent Benjamin, Philippe Lacomme, Libo Ren, Shortest Path Resolution using Hadoop, 2014
- [19] Alexis Papadimitriou<sup>1</sup>, Dimitrios Katsaros<sup>2</sup>, Yannis Manolopoulos, Social Network Analysis and its Applications in Wireless Sensor and Vehicular Networks,
- [20] Ulrik Brandes, A Faster Algorithm for Betweenness Centrality