UPPSALA
UNIVERSITET

UPTEC F12 007
Examensarbete 30 hp
Januari 2012

Social Network Analysis Utilizing Big Data Technology

Jonathan Magnusson

UPPSALA UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Social Network Analysis Utilizing Big Data Technology

Jonathan Magnusson

As of late there has been an immense increase of data within modern society. This is evident within the field of telecommunications. The amount of mobile data is growing fast. For a telecommunication operator, this provides means of getting more information of specific subscribers. The applications of this are many, such as segmentation for marketing purposes or detection of churners, people about to switching operator. Thus the analysis and information extraction is of great value. An approach of this analysis is that of social network analysis. Utilizing such methods yields ways of finding the importance of each individual subscriber in the network. This thesis aims at investigating the usefulness of social network analysis in telecommunication networks. As these networks can be very large the methods used to study them must scale linearly when the network size increases. Thus, an integral part of the study is to determine which social network analysis algorithms that have this scalability. Moreover, comparisons of software solutions are performed to find product suitable for these specific tasks.
Another important part of using social network analysis is to be able to interpret the results. This can be cumbersome without expert knowledge. For that reason, a complete process flow for finding influential subscribers in a telecommunication network has been developed. The flow uses input easily available to the telecommunication operator. In addition to using social network analysis, machine learning is employed to uncover what behavior is associated with influence and pinpointing subscribers behaving accordingly.

Handledare: Tor Kvernvik
Ämnesgranskare: Justin Pearson
Examinator: Tomas Nyberg
ISSN: 1401-5757, UPTEC F12 007

Sammanfattning

Den tekniska utvecklingen medför att en alltid tilltagande mängd av data genereras. Nya innovationer tas fram hela tiden och den redan befintliga tekniken förfinas allt mer. Tekniska processer får allt fler delsteg och komplexiteten av varje enskilt delsteg ökar. En logisk följd är att också skapandet av data ökar. Kommunikation mellan de olika delstegen innebär dataöverföringar och fler samt mer noggranna loggar av varje delsystem måste föras, då antalet möjliga felkällor växer.

Ett område där detta i allra högsta grad är aktuellt är telekommunikation. En stor del av jordens befolkning är idag uppkopplat till ett telekommunikationsnätverk. Varje interaktion i nätverket, abonnenter emellan, resulterar i en automatisk notering, utförd av nätverksoperatören. Denna lista av noteringar kan sedan användas i syfte att debitera rätt avgift till varje abonnent. Givetvis ökar antalet noteringar i denna lista då antalet personer i nätverket tilltar, men också i samband med att ny teknik blir tillgänglig och fler möjliga sätt att kommunicera utvecklas.

För en nätverksoperatör implicerar detta att kapacitet att hantera stora datamängder måste upprätthållas. Men det innebär också att möjlighet att studera abonnenternas beteende erhålls. Ur varje notering går att utläsa en mängd information om interaktionens natur. Där framgår vilken typ av transaktion som utfördes, hur länge den pågick och vart avsändaren befann sig, för att nämna några saker. Att granska noteringarna över tid ger en inblick i hur abonnenter använder operatörens tjänster och vilka kontakter, dvs andra abonnenter, som de interagerar med. Att äga kunskap om användarens beteende kan av en nätverksoperatör omsättas i fördelar gentemot konkurrenter och i förlängningen en starkare position på marknaden. T.ex. blir det möjligt för operatören att skräddarsy sin tjänst för enskilda abonnenter, utföra effektivare marknadsföring och se till att minimera antalet abonneter som konverterar till en annan operatör.

I detta skede reses en viktig fråga; hur ska den ostrukturerade datan omvandlas till överskådlig information som kan ge relevanta upplysningar om abonnenter? Denna fråga kan och har attackerats på ett antal olika sätt. Vidden av projektet beskrivet i denna rapport avgränsas dock till studier av telekommunikationsnätverk i termer av social nätverksanalys. Inom social nätverksanalys studeras både den generella strukturen av ett nätverk och, på den mindre skalan, rollen av enskilda objekt i nätverket. Fokus här har varit på att avgöra hur enskilda abonnenter uppför sig och deras möjlighet att påverka andra abonnenter i nätverket.

Att storleken på nätverken ständigt ökar och så också den datamängd som genereras av dem har givit projektet huvudinriktingen att finna en skalbar lösning för att med social nätverksanalys finna viktiga abonnenter. Målet är att lösningen ska kunna hantera också framtidens nätverk. Relaterat till detta är att välja den mjukvaruprodukt som är mest lämpad för storskalig social nätverksanalys. Viktigt är här att hårdvaran i så liten mån som möjligt ska begränsa analysen av nätverken då storleken ökar. En tredje del av skalbarheten är att välja ut de sociala nätverksalgoritmer vars exekveringstid växer linjärt med storleken på nätverket. Detta handlar helt enkelt om en optimering av utvunnen information och tidsåtgång.

Till detta har projektet innefattat att tolka den information som den sociala nätverksanalysen ger. Varje algoritm som tillämpats ger en bild av en viss aspekt av influens för den enskilde abonnenten i nätverket. Att väga samman dessa olika aspekter är ingen trivial uppgift. Därför har studien också riktats mot tekniker för att göra denna sammanslagning automatiskt.

Lösningen för detektion av intressanta abonnenter som har tagits fram är ett processflöde som grovt kan delas upp i tre steg. Här ingår dels att processera den rådata som skapas i nätverket av operatören. Det innebär att från en lista över samtal och meddelanden skapa en representation av ett socialt nätverk. Det andra steget handlar om att på detta sociala nätverk applicera de algoritmer som används inom social nätverksanalys. Varje algoritm räknar fram ett mått på viktigheten av varje abonnent i nätverket. Av detta steg erhålls en lista på alla abonnenter i nätverket och, för varje abonnent, en mängd värden som indikerar influens i någon mening. Som avslutning inkoopereras också det andra huvudmålet i lösningen; att automatiskt väga samman de olika måtten till ett aggregerat värde. Det visar sig att, med rätt valda algoritmer, kan denna process genomföras på under två timmar för riktig nätverksdata, motsvarande ett nätverk på 2,4 miljoner abonnenter, där interaktion studerats under en månads tid.

När det kommer till rätt typ av mjukvaruimplementation har två huvudspår undersökts. Dessa är parallelisering via ett distribuerat system och grafdatabaser. Tester antyder att grafdatabasspåret begränsas mer av hårdvaruresurser och att den distribuerade lösningen klarar skalbarhetskravet bättre.

Resultaten av studien visar slutligen att ett lovande sätt att angripa problemet med att tolka resultaten av nätverksanalysen på ett automatiskt sätt är att nyttja maskininlärning. De tester som genomförts har rört övervakad maskininlärning som kräver väldefinierade exempel på de mönster som ska detekteras för att träna algoritmen. Resultaten indikerar att om man tar hänsyn till tillräkligt många aspekter av det som ska predikteras, t.ex. influnens, har maskininlärning potential att ge akurata estimeringar.

Key Words

Social Network Analysis, Telecommunication Networks, Hadoop, Machine Learning

Contents

1 Introduction

1.1 Background

As of late there has been an immense increase of data within modern society. This is evident within the field of telecommunications. The amount of mobile data is growing fast. This is partly due to a spread in mobile technology, not least to the third world, but also due to more complex technology and increased capacity when it comes to transmitting information [1]. Many devises developed today have wireless connections of some sort. Predictions have it that within a foreseeable future 50 billion entities will be connected [2]. This implies not only increasing demands on telecommunication equipment, but also a possibility of analyzing and deducing more information from network traffic.

For a telecommunication operator, this provides means of getting more information of specific subscribers. The applications of this are many, such as segmentation for marketing purposes or detection of churners, people about to switching operator [3] [4]. Thus the analysis and information extraction is of great value.

An approach of this analysis is that of social network analysis [5]. In accordance with this field of research the telecom network can be represented as a graph. This contains vertices, the subscribers, and edges, relations between subscribers. This graph can then be analyzed to find the importance of each individual subscriber in the network.

1.2 Problem Statement

The aim of this thesis is to resolve the following questions:

1. How can extremely large telecommunication networks be analyzed in terms of social network analysis?
    (a) Can a linearly scalable solution be found?
    (b) Which technology is suitable for this task?
    (c) Which social network analysis algorithms can be implemented in a scalable way?
2. How can the information found in the social network analysis of the telecommunication network be used to identify important subscribers?

1.3 Method

The main obstacle regarding social network analysis of telecommunication networks is the vastness of the dataset. Analysis of a network consisting of millions of connected objects is usually very computationally costly and may take quite some time to perform [6]. As an answer to the problems of very large data sets, as well as related problems, the development is directed towards parallel processes [7]. Examples of this can be found in commodity computing and cloud-services. The idea is to divide a problem into parts and let several machines processes it at the same time, separately. This solution is generally preferred before one single strong computer, perhaps with multiple processors, as it is economically cheaper. Google is, as of now, the pioneer of this approach with their software solutions MapReduce [8] and Pregel [9]. However, open-source alternatives, most prominently Hadoop [10], are making its way into the industry.

A somewhat different approach is that of using graph databases. Instead of the strict order found in relational databases, such as SQL, a graph database does not require any given pattern in which entries are defined. An example is Neo4j [11]. A Neo4j database consists of a set of nodes and relations, which both has properties related to them. Nodes and relations are regarded as equally important, and this makes traversals of the graph faster than for RDBMSs.

The task at hand regards analysis of large social networks that is a representation of a telecommunication network. The approaches mentioned above are tested in order to determine their ability to solve this problem.

The analysis involves measuring the importance of each subscriber within the network. For this, metrics thought to be related to the ability of a subscriber to spread information within a network are calculated. To do this satisfactory the whole graph must be available. This can cause problems for large graph if the storage size is greater than the size of the hard drive. Additionally, solving the problem in parallel using MapReduce or Hadoop implies storing different parts of the data on separate hard drives. Each process then accesses only the locally stored part of this data, thus only a confined portion of the graph representing the network. Clever algorithms, taking into account the issues of parallelization, are a solution to this. In graph databases, such as Neo4j, without parallelization the whole graph will be available at all times, but the issue of graphs larger than disk space still exists.

Using the technology of the Hadoop framework and making use of knowledge form the field of machine learning, a prototype for detecting subscribers of particular interest is developed. This uses Call Detail Records (CDR) generated by the operator, for charging purposes, as input. From them a graph in created and social network analysis metrics are determined for each subscriber. A training set is provided to create a decision model for classifying the subscribers.

1.4 Outline of the Thesis

The actual thesis starts with section 2, which is a theoretical review. First, the general field of social network analysis is visited and the relations to telecommunication networks are discussed. Following that, two possible techniques for performing social network analysis effectively, parallelization and graph databases, are studied. Furthermore, some algorithms for machine learning is described and the chapter is ended with a summary.

The next part of the thesis, section 3, covers how the conclusions made from the theoretical part has been implemented to solve the stated problems of the thesis. It gives an account of the systems used to perform the test, as well as the input of the algorithms. Moreover, it introduces a prototypical solution of the whole process of identifying subscribers of particular interest, from the input of CDRs to a list of significant subscribers as output.

Section 4 is a summary of the results obtained when testing the algorithms related to this study. This is divided into parts comprising the pre-study results, the results of tests of Hadoop in clustered mode and the experiments related to the prototype. Additionally, some results of the distribution of social network analysis metrics in a real telecommunication network is accounted for. The results are then discussed in section 5

A summary of the most important results and their implications is found in section 6. Furthermore, some suggestion for further work is listed.

Subsequent to the references, a couple of appendices is present. They involve ways of parallelizing social network analysis algorithms and specifications for the computer cluster used during the study.

2 Theory

This chapter of the thesis involves a theoretical approach of analyzing telecommunication networks in terms of social network analysis. It begins with a discussion of social network analysis in general and its implementations in terms of telecommunication networks in particular. The focus is then directed toward tools for performing a large scale social network analysis. A more detailed description of the Hadoop and Neo4j systems is found in these parts. Finally, an investigation of machine learning algorithms is done, as this can be used to interpret the results of the social network analysis.

2.1 Social Network Analysis

2.1.1 Graphs

In general, social networks can be viewed as a set of connected entities. This is commonly represented by a collection of vertices and edges. A vertex is an abstraction for an actor in the network whereas edges are relations between these actors. Within mathematics, this is known as a graph [12]. Formally, a graph can be written as G(V,E), where V is a set of vertices and E is a set of edges connecting vertices in V. The number of vertices is denoted n and the number of edges m, with common notation. A second graph S(V',E') is a subgraph of G(V,E) if V' $\subseteq$ V and all edges in E' exists in E.

The simplest type of graphs is undirected and unweighted graphs. An undirected graph is defined by the edges always being two-way relations and in an unweighted graph all edges are equally strong. A more general type of graph is the directed and weighted graph. Directed implies that the relations can be existing as a single-way or two-way relation. Weighted refers to the fact that relations may be of varying importance. This is represented by a number called the weight w. For an unweighted graph, all weights are 1 [12].

To represent a graph, for instance in a computer, adjacency matrices and adjacency lists are frequently used. An adjacency matrix is an n-by-n matrix that has A(i,j) = w if there is a relation between i and j, with weight w, and A(i,j) = 0 otherwise. The adjacency list provides an array to each vertex with numbers corresponding to its neighboring vertices and weights of the relations.

2.1.2 Telecommunication Networks

As of late, graphs have had a lot of scientific attention [13]. With the growth of the Internet and the connectivity of people around the world, areas of use for graphs are becoming increasingly common. Applications range from biology to economics [14]. Graphs are also commonly used to represent groups of people, one example being subscribers of a telecommunication network.

A characteristic of telecommunication networks is their large-scale. It is not unusual to find networks with over a million vertices and even more edges. Another typical characteristic is the sparseness of the network, that is, the number of edges is much smaller than the maximal number of edges. Furthermore, a common feature of telecommunication graphs is that they are scale-free, implying that the distribution of number of relations of every subscriber follows a power law behavior. Additionally, the small-world phenomena, low degree of separation in average, is usually prevalent in telecom graph.

The general structure of a telecom graph can be described by the Treasure-Hunt model [15]. This is similar to the Bow-Tie model and describes the graph as being composed of one very large connected component. A great part of this component is quite strongly connected, called the Maze, but there are additionally smaller parts reaching out from this Maze. This is shown in figure 1.

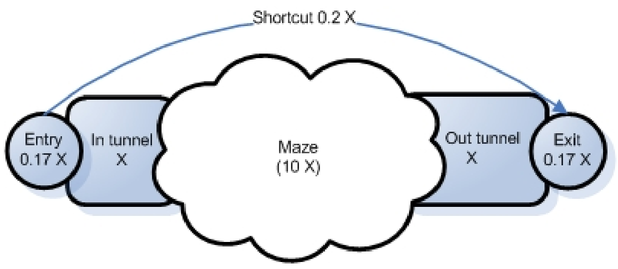


Figure 1: The Treasure-Hunt model. Courtesy of Fredrik Hildorsson.

What is often found in graphs representing social interactions of biological entities is that there is a certain assortative mixing [16]. This refer to the tendency that well connected vertices in the network is more prone to be connected to other well connected vertices, compared to predictions in line with random models.

2.1.3 Synthetic Graphs

There are a number of commonly used methods for generating synthetic graphs. An investigation of some of these in the context of telecommunication networks are given in [17]. There, 4 methods are mentioned, namely Barabasi-Albert [18], Kleinberg (directional) [19], Watts Beta Small World [20], Eppstein Small World [21]. It is concluded that they all manages to capture certain characteristics of real telecom networks. Especially, they tend to be subjected to a power-law distribution, the small world phenomenon, high degree of clustering and assortative mixing of degree of adjacent vertices. This is decsribed in further detail in section 2.1.2. None of the graph generating algorithms mentioned above is able to incorporate all of these features. Moreover, none of the algorithms are well suited for implementation in a parallel environment of the MapReduce logic. More appropriate methods, due to the ease with which they are parallelized, are a Random generator and the Erdõs-Rényi algorithm, which will be described below.

The Random graph generator determines the degree of each vertex in the graph according to a uniform distribution over a customizable interval. Now, for each vertex a number of other vertices, equal to the predefined degree for the vertex, is chosen as neighbors. Additionally, if the graph to be created is weighted, a random weight between 0 and 1 is picked. This may result in some overlap as identical edges could be generated twice. Should this situation occur, the edge will be created only once for the unweighted case and the weights added if the graph generated is weighted.

The Erdõs-Rényi graph generating algorithm is somewhat more elaborate. A detailed definition of it is available in [22]. For each pair of vertices in the graph, an edge will connect them with a constant probability. In practice, a vertex is created and the connection test is performed for all previously created vertices. That way every pair are tested only once and double edge creation is avoided.

Both these algorithm is readily implemented in a parallel environment. This as neither of them requires any information regarding the graph except the local environment within the graph.This is important if very large network is to be generated.

2.1.4 Some Common Metrics

Analyzing graphs often involves calculation of a number of metrics. One group of metrics is the measures of centrality. These are quantities that illustrate the importance of each vertex within the network. The importance can be with respect to being related to other actors within the network or being minimally separated from all other actors. The most common are degree centrality, closeness centrality, eigenvector centrality and betweenness centrality. Figure 2 shows an example network. Notice that the centrality metrics generally do not agree on which vertex has the highest score, thus being deemed most important. The metrics will each be discussed in more detail in the following sections.

Degree Centrality

The simplest centrality metrics is the degree centrality. It is defined as the number of vertices that a given vertex has a relation to. Classically, this number is only relevant for unweighted graphs, but has been extended for weighted graphs to be the sum of all weights of a vertex relations [23]. This metric is readily calculated with access to an adjacency matrix or an adjacency matrix. With an adjacency matrix the degree centrality is the sum of the entries on the row or column corresponding to the vertex. in an adjacency list the length of the array listing the neighbors of a vertex is the degree of the vertex. This can be calcualted in $\mathcal{O}(\mathrm{n})$ time. For directed graphs this metric can be expanded into to separate metrics; in-degree and out-degree. This is the number of relations in to and out from the vertex, respectively.

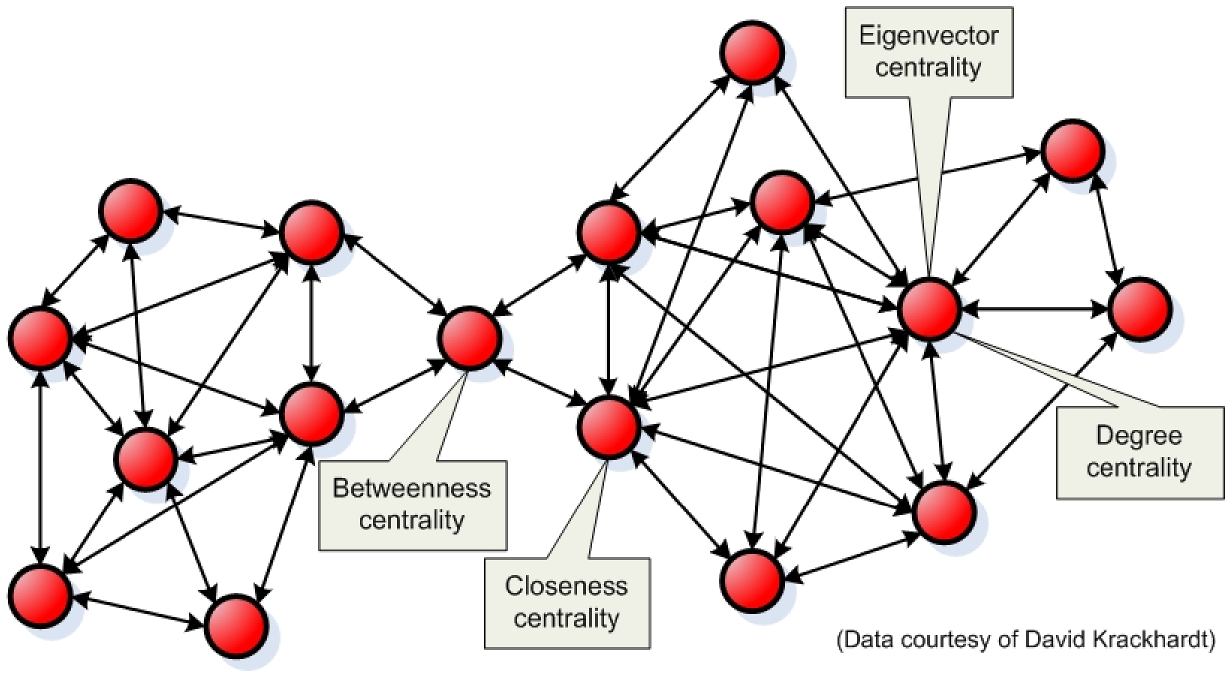


Figure 2: A small social network, where the most important vertices according to different centrality measures has been marked.

Closeness Centrality

Closeness centrality can be viewed as an extension of degree centrality. Instead of only considering the neighbors of a vertex, all other vertices are taken into account. The metric measures the average distance from the source vertex to any other vertex within the graph. Formally, this can be written as:

$$C_C(v) = \frac{\sum_{t \in V \setminus v} D_G(v,t)}{n-1} \tag{1}$$

where $\mathrm{D}_G(v,t)$ is the shortest path, called the geodesic, between vertex v and vertex t. This would reflect the time information needs to spread from a source vertex. Sometimes the closeness centrality is defined as the reciprocal of the value given in (1), which corresponds to the speed of information flow from the source vertex.

A number of definitions of closeness centrality have also been made for weighted graphs. The definition to be used in this paper was proclaimed by Fowler in [24]. He defines the distance between two related vertices as the reciprocal of the weight of their relationship and uses that distance to find geodesics.

Eigenvector Centrality

The third metric mentioned above, eigenvector centrality, is the values of the eigenvector corresponding to the largest eigenvalue of the adjacency matrix. Thus, this can be written as:

$$C_E(v) = \frac{1}{\lambda} \sum_j R_{ij} (C_E(v))_j \tag{2}$$

where R$_{ij}$ is the i:th eigenvector of the adjacency matrix of the network regarded.

Eigenvector centrality can be applied to both weighted and unweighted graphs, as well as directed and undirected. This can be thought of as an iterative process, where in each step a vertex sends information to its neighbors. The amount of information sent depends on the importance of the vertex itself and the strength of the relation to the receiving vertex. As the adjacency matrix is of size n×n, with n usually being very large, it might be computationally costly to calculate eigenvectors of the matrix. A solution to this is an iterative method called the power iteration. The method consists of repeatedly multiplying the adjacency matrix of the graph with a vector b, which is updated as the result of the multiplication at each step. Additionally, the vector b is normalized at each step. The kth iteration can be represented as follows:

$$b_{k+1} = \frac{Ab_k}{\|Ab_k\|} \tag{3}$$

The initial vector, b$_0$ can be chosen randomly, but a common choice is to give each vertex equal starting value, e.g. 1. This algorithm has a time complexity of $\mathcal{O}$(n+m).

A famous algorithm for graph analysis is PageRank$^{\text{TM}}$, developed by Google, which is based on eigenvector centrality [25]. It is a tool for analyzing directed graphs, Google uses it to evaluate the importance of linked web pages. In addition to the original metric of eigenvalue centrality, PageRank$^{\text{TM}}$ incorporates a damping factor, which represents web surfers choosing a random web page instead of clicking on a link at the page visited at the moment.

Betweenness Centrality

Betweenness centrality is a concept commonly used in analysis of graphs. It measures to what degree actors have to go through a specific actor in order to reach other actors. Or, in other words, on how many geodesic paths, out of all geodesics, between a pair of actors that the specified actor lies. This is summed over all pair of vertices. Formally, this can be expressed as:

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{4}$$

where $\sigma_{st}(v)$ is the number of shortest paths between s and t that runs through v an σ_{st} is the total number of shortest paths from s to t.

This is a metric of computational heaviness, as it involves calculating the shortest paths between all vertices in the graph. For large graphs this can be extremely costly in terms of time. An algorithm calculating betweenness centrality for sparse networks was proposed by Brandes in [26]. Brandes' algorithm incorporates breadth-first search (BFS), an algorithm for finding all shortest paths from a single source. The result of the BFS algorithm can be used to determine the closeness centrality of the source vertex. The computational cost for this algorithm is of the order $\mathcal{O}$(nm). A variation on this theme is the flow betweenness centrality. It is calculated in much the same manner as the regular betweenness centrality but takes into consideration all possible paths between any pair of vertices, not only the geodesics.

An approximation of this measure that gives significantly shorter computation times for large networks is the ego betweenness centrality. In order to calculate this measure an ego network for each vertex must be created. The ego network of a vertex is the network containing the vertex and its closest neighbors, as well as all relations between these vertices. The ego betweenness centrality of a vertex is essentially the betweenness centrality of the vertex within its ego-network. A simple way, given the adjacency matrix of the ego network, of calculating this is described in [27]. The time complexity of this algorithm is $\mathcal{O}$(n+m).

2.1.5 Communities

Within the field of graph theory, there are various ways by which graphs can be dismantled into subgraps. One of them is detection of communities. A wide range of definitions of communities within social networks have been proposed. Common for these definitions are that they describe communities as sub-graphs of a network that is strongly connected internally. One definition involves the quality function of modularity. It states that communities of a network are those sub-graphs that, when splitting the network, yields the optimal value of modularity [28]. Modularity can be defined through the following mathematical expression:

$$Q = \frac{1}{2m} \sum_{i,j} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \tag{5}$$

where A$_{ij}$ is the weight of the edge connecting vertex i and j, k$_i$ is the number of edges connected to vertex i, c$_i$ is the community of vertex i, m is the total sum of weights in the whole network and δ(i,j) is the dirac-delta, defined as

$$\delta(i,j) = \begin{cases} 0 & \text{if } i \neq j \\ 1 & \text{if } i = j \end{cases} \tag{6}$$

This modularity measure is valued from -1 to 1 and is a comparison of the number of links within communities and those connecting different communities.

A number of algorithms for optimizing this global graph measure have been proposed. To be able to handle very large networks, these algorithms should preferably be optimized for speed. An algorithm that has proven useful and efficient in terms of computational time was proposed by Blondel et al. [29]. It is an iterative algorithm that for each step merges neighboring communities if that merging will result in an increase in modularity. In the initial step, every community consists of a single node. As the iteration proceeds, these single vertex communities will be joined together into larger communities. This procedure stops when no further modularity increasing merging can be performed. At that point the next phase of the algorithm commences. It involves transforming each community created in the first phase into a new vertex. This vertex has self-looping edges corresponding to the vertices in the community and edges connected to other vertices with a weight equal to the total weight of edges between vertices in the respective community. This completes the iteration and the next iteration starts using the new network as input.

It is important to note that this algorithm implies a lot of calculation of change in modularity. This can be done in a rather quick way using the formula, as stated in [29]:

$$\Delta Q = \left[\frac{\Sigma_{in} - k_{i,in}}{2m} - \left(\frac{\Sigma_{tot} + k_i}{2m}\right)^2\right] - \left[\frac{\Sigma_{in}}{2m} - \left(\frac{\Sigma_{tot}}{2m}\right)^2 - \left(\frac{k_i}{2m}\right)^2\right] \tag{7}$$

where Σ_{in} is sum of the weights in community that the vertex is moved to, called C, Σ_{tot} is the aggregated weights inbound for C, k_i is the total incomming weights for vertex i, $k_{i,in}$ is the same quantity as k_i but only counting edges in C. Additionally, the change in modularity for removing a vertex from a community can be calculated in an analogues fashion. The difference between these modularity changes will reveal the total modularity change.

This approach is not guaranteed to find the graph decomposition that yields the optimal modularity. However, test results indicate that the algorithm is able to achieve high values of modularity for several types of networks [29].

2.2 Parallelization

Large sets of data are becoming more common within a number of fields of research, not only social network analysis. As a response to this, new methods of processing this data have to be developed. An idea gaining foothold among computer scientists is parallelization. This is basically doing parts of a process separately and in parallel. MapReduce is a framework of parallelizing a computation process on a cluster of computers. This system was developed by Google and is protected by patent laws. It is described in [8]. A similar open-source project is Hadoop [10]. These systems are frameworks which deal with the parallelization, fault tolerance and scheduling of machine communication for large tasks. In essence they are using the same idea, which will be described in the following section.

2.2.1 Tools of Parallelization

Any computer process performed in parallel requires a computer cluster. A computer cluster consists of a number of computers, usually referred to as nodes. When the process is initiated, separate programs start on each computer in the cluster. In a Hadoop cluster, one computer has a special role, the Master. It contains meta-data that controls and organizes the process, with scheduling and load-balancing as well as fault handling. Every Hadoop job is divided into smaller processes. The compulsory kind of process is map tasks, but many implementations makes use of other types of processes, such as combine tasks and reduce tasks. Figure 3 gives a detailed overview of this. At first the procedure is divided into M map tasks and R reduce tasks by the Master, where M is usually chosen to make the size of each part between 32 and 64 MB by default. The Master then distributes the work processes among the other machines; the Workers or Slaves.

Each input data is assigned a key and a value, as indicated in figure 3. This is processed in the map task and the output is a new key with a new value. This is called the intermediate data, which will be the input of the reduce task. The Master keeps track of the intermediate data and based on the key of this data, it is spread into different reduce tasks. More specifically, all data with the same key is gathered in one reduce task. The data is further processed in the reduce task, which do not start until all map tasks are finished and all intermediate data is available. During the reduce process, the records are inspected and handled one by one using an iterator which is finally giving an output key and value.

The map and reduce processes can be adapted to a large degree to conform with the preferences of the user and thus they have potential of performing a wide range of computational jobs. A typical example of a MapReduce job is the WordCount problem [30]. This consists of counting the number of times a word occurs in an input text. To do this the input file is split into a number of blocks. Each map task will have such a block as input. For each word in the block an output will be created by the mapper, having the word as key and the number 1 as value. The sorting function of the Hadoop framework makes sure that each output of a certain key, in this case word, end up in a specific reduce task. In the reduce phase the values are simply added and the total number of times a word occurs is obtained. The complete process of the WordCount implementation can be viewed in figure 4.

In general, for a task to be suitable to be processed in a MapReduce-like fashion it must be dividable in independent parts. At a first glance it might seem as graph problems is not dividable into independent parts, but the fact that every vertex can be processes by its own makes the division possible. An example of this can be found in [31].

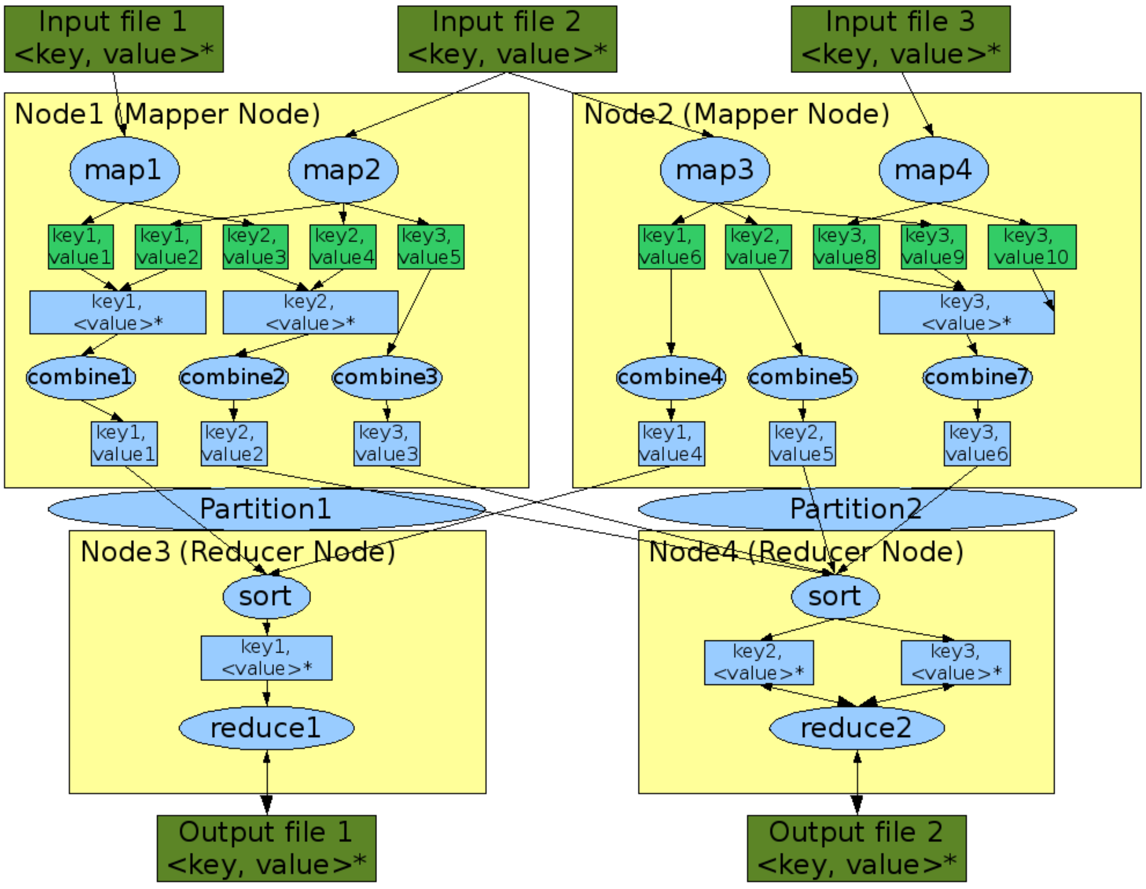


Figure 3: An overview of the typical composition of a MapReduce job. Note that a mapper and reducer node could be situated at the same machines.

An example relevant to the studies of this thesis is shown in figure 5. This particular example involves ego betweenness centrality. The parallel implementation of the algorithm focuses on separate vertices in each task. In the map task, each vertex sends one message to each of its nieghbors in the graph. Th message proclaims the ID of the sending vertex and a list of the sending vertex' neighbors. In the reduce task, each vertex recieve one message per nieghbor. The information in these message is used to create an adjacency matrix for the ego network of the recieving vertex. Lastly, the adjacency matrix is manipulated as to yield the value of ego betweenness centrality for the recieving vertex and this is sent as output. A more detailed description of other social network analysis algorithms implemented during this study can be found in Appendix A.

Described above is the processing part of Hadoop. Additionally, there is a storage part of the framework. Each computer in the cluster has a hard drive for storing information. Part of each existing hard drive in the cluster is allocated to be a part of the distributed file system used by Hadoop, called HDFS. The system has been design to handle large data set,

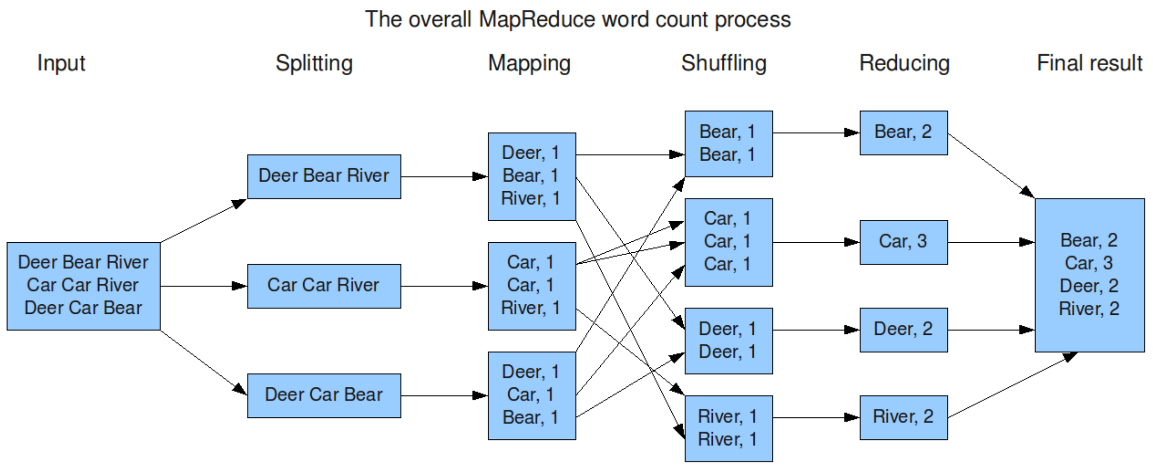


Figure 4: An example of the WordCount process. Note that a mapper and reducer node could be situated at the same machines.

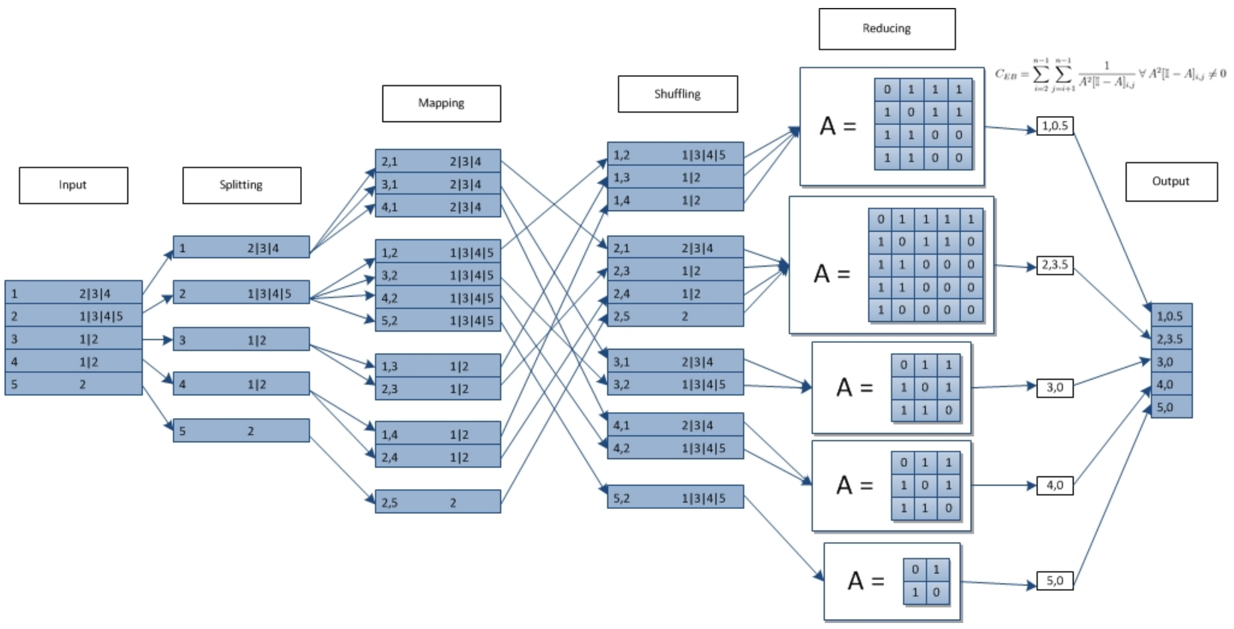

Input
Splitting
Mapping
Shuffling
Reducing
Output
$C_{EB} = \sum_{i=2}^{n-1} \sum_{j=i+1}^{n-1} \frac{1}{A^2[\mathbb{I} - A]_{i,j}} \forall A^2[\mathbb{I} - A]_{i,j} \neq 0$
A =
1,0.5
2,3.5
3,0
4,0
5,0

Figure 5: An implementation of the algorithm for calcualting ego betweenness centrality as a MapReduce job.

frequent machine failure and high throughput access. Like the processing parts of Hadoop, HDFS implements a hierarchy where one node has a Master-like status, the Namenode. The Namenode is commonly situated on the Master machine. The slave machines' contribution to the HDFS is called Datanodes. The structure of the HDFS can be seen in figure 6

A property of the Hadoop framework is that all input is read from the HDFS and all output is written to it. This might be time-consuming for large files. Many parallelized algorithms for graph analysis must be done iteratively, consisting of a large number of map and reduce jobs. This might increase the total time of the complet procedure since Hadoop writes the reducer output to a file in the HDFS, and has to do this repeatedly. A number of projects are trying to work around this problem. Notable examples are Pregel [9], Spark [32] and Hama [33].

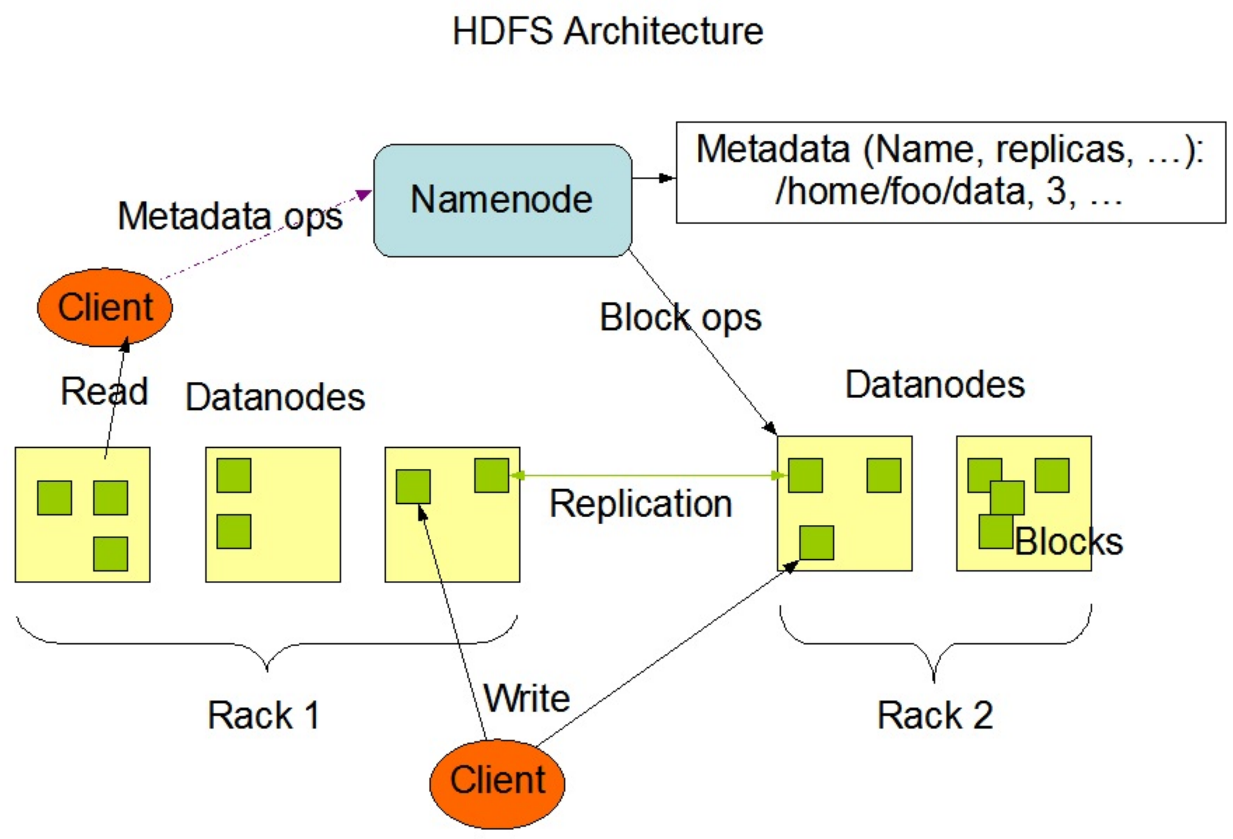


Figure 6: An overview of the typical composition of a MapReduce job. Note that a mapper and reducer node could be situated at the same machines.

A very important part of the MapReduce framework is the blocking mechanism. This is the way MapReduce deals with machine failures. The Master sends signals to each Worker periodically and if it does not receive an answer within reasonable time the machine is deemed broken and the process it was performing is assumed failed. If the task is not completed it will be re-scheduled for a different machine. If the task is completed however, it will be treated differently depending on if it is a map task or a reduce task. For a map task it will be reprocessed as the output data is stored locally on the failed machine, thus unreachable. A reduce task, on the other hand, stores its output in the HDFS. Hence, it could be used and it is not necessary to redo the task.

For the user the job is to define a map and a reduce, and in addition to this a driver program must be made that defines the properties of each job and decides the steps between and around the map and reduce tasks. Moreover, a commonly used type of job is a combine task. A combiner is in many ways very similar to a reduce task. However, it operates locally on output given from a single map task. The aim is to reduce the output from one mapper into a single output message for each reduce task addressed. In some applications the reduce task can be used as a combine task. An example is the iterative job of finding eigenvector centrality in a graph. The same is true for PageRank, which is a modified version of eigenvector centrality. This, along with a few other optimizations, was shown to have reducing effect on computational time of PageRank in [34].

2.3 Graph Databases

In general, a graph database is a database consisting of the basic graph structures defined in section 2.1.1, that is, vertices and edges, as well as properties. As for graph theory in general the vertices represents entities that are related to each others by edges. Vertices is analogues to objects in traditional object-oriented databases. Properties are information associated to either a vertex or an edge.

Graph databases is often performing computations faster on related data, compared to relational databases. Additionally, databases based on graphs tend to scale better to very large data sets as they are not dependent on join operations [35]. Furthermore, graph databases are more adaptable to unsymmetrical data and dynamic data that changes over time.

A number of systems for storing information as graphs are being developed. A system that has gained some publicity as of late is the graph database Neo4j, which will be described below.

2.3.1 Neo4j

Neo4j [11] is a graph database and follows the general definition as stated in the section above. That is, its data structure consists of vertices and edges, both which can be described by properties. A typical example of such a data structure is shown in figure 7.

Queries to the database take the form of traversals. A traversal basically involves moving in the graph from node to node using edges. The traversal ultimately returns some output related to the nodes visited.

All modifications of a database take place within a transaction. Neo4j aims at obtaining ACID properties. This comprises transactions that have atomicity, consistency, isolation

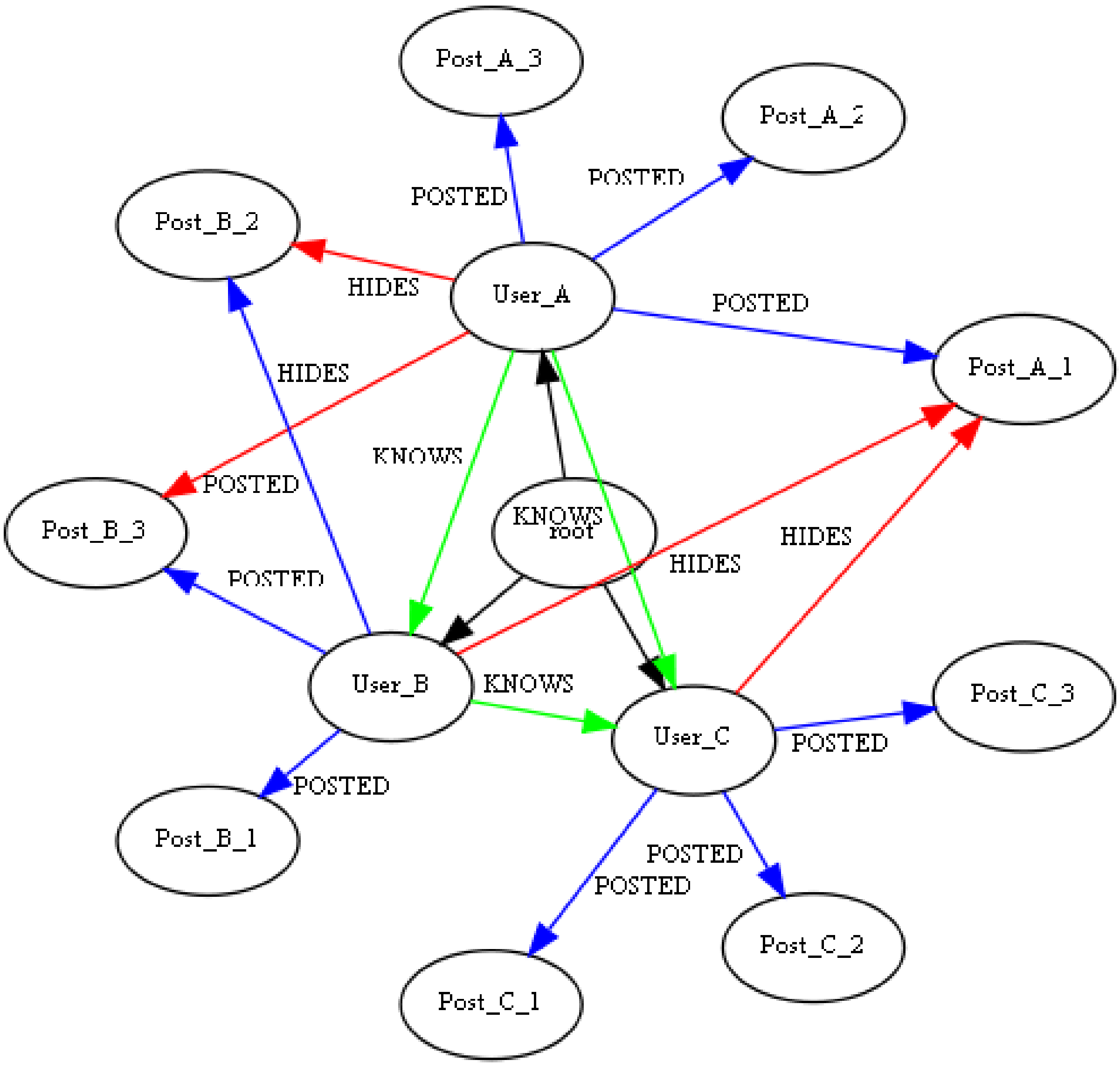


Figure 7: An example graph database structure that illustrates a network of three users and posts posted by them. Courtesy of Rob Olmos.

and durability. Atomicity refers to the need of a transaction to finish all parts of it correctly. If just a single part fails, the complete transaction will be invalid. Consistency forces every modification on a database to be consistent with rules defined for the data. For example, copies of the same source data, stored at separate places, should be identical if consistency is maintained. The third property, isolation, implies that transactions do not influence each other. This ensures that two transactions cannot work on the same piece of data simultaneously. Lastly, durability requires that a completed transaction will remain completed indefinitely.

It is possible to run the system on a single machine, embedded mode, or implement it in a cluster of several computers. However, it should be noted that this parallelization does not imply horizontal scalability. Horizontal scalability, or sharding as it is called within the field of database science, refer to storing different parts of the database on separate machines and transactions relevant to that data is performed on the machine housing the data. Horizontal scalability is a planned future feature. Currently, the parallelization implemented in Neo4j involves identical copies of the database stored on every machine in the cluster. To make the transition between embedded and clustered modes as simple as possible, Neo4j implements a module called High Availability. An aim of the database system is to ensure that the consistency properties are maintained. However, in the distributed High Availability mode, this will not be absolutely possible in a ACID sense, but rather achieving an eventual consistency, known as BASE properties. This is a common concept within parallel computing and essentially implies that given a long enough time period, the different copies of the database will be consistent with each other.

2.4 Machine Learning

Machine learning is a part of the field of artificial intelligence. It mainly focuses on the development of certain reactions in specific situations. These reactions are based on information gathered from empirical data. This is closely related to data mining, where patterns are detected in large data sets [36].

There is a large number of techniques which can be referred to as machine learning algorithms. Two main groups can be defined; white box and black box methods. White box methods are translucent in that they reveal the process by which the output is created from the input. Black box methods, however, do not inform the user of how an output is produced. When knowledge regarding the properties of the input data is sought, white box methods are preferable.

In the sections below the white box methods of decision trees and logistic regression, as well as the black box method of neural networks, will be discussed. All of these methods are so called supervised learning method. For these types of methods input examples with known outputs are needed to train the machine.

2.4.1 Decision Trees

The decision tree algorithm constructs a set of rules that forms a tree like structure. This rule set is used to classify examples, or instances, into one of a number of predefined outcomes. Each instance is composed of a number of attributes which characterize the instance.

The tree is a set of nodes, each representing a rule, used to classify the input instances of the problem. Branches will connect the nodes and lead the classifying algorithm on a specific path depending on the value of each attribute of an instance. Ideally, each instance in the training set can be classified correctly using the decision tree. However, this is not always possible for an arbitrary set of instances. Figure 8 illustrates this with an example, where subscribers are classified as important or unimportant depending on social network analysis metrics calculated for them.

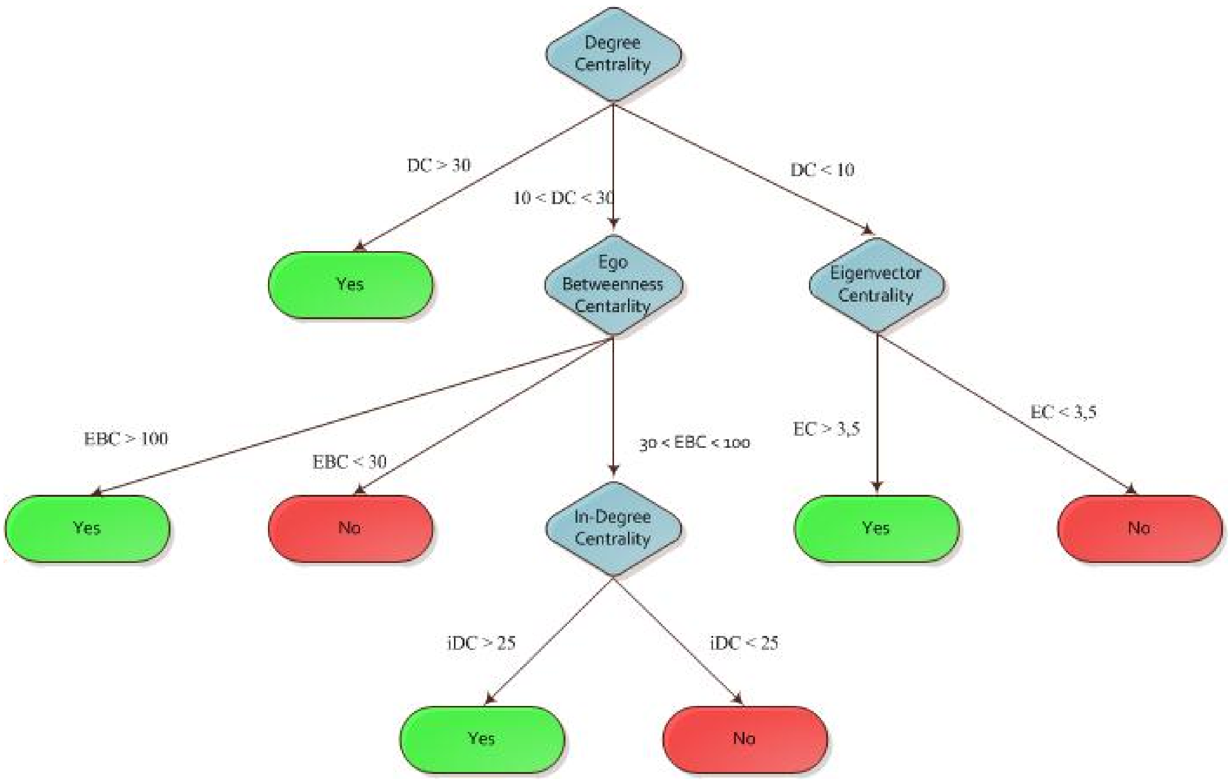


Figure 8: A simple example of a decision tree, based on the social network analysis metrics introduced in earlier sections. Yes indicates importance.

A commonly used basic algorithm for constructing a decision tree is called ID3 [37]. It is a recursive algorithm that for each step considers each of the attributes separately. For each attribute an information gain is calculated. Information is enumerated in the same way as entropy is, thus according to the formula

$$\begin{aligned} \text{info}(p_1, p_2, \ldots, p_n) &= \text{entropy}(\frac{p_1}{P}, \frac{p_2}{P}, \ldots, \frac{p_n}{P}) = \\ &= -\frac{p_1}{P}\log\frac{p_1}{P} - \frac{p_2}{P}\log\frac{p_2}{P} - \ldots - \frac{p_n}{P}\log\frac{p_n}{P} \end{aligned} \quad (8)$$

where

$$P = \sum_i p_i \tag{9}$$

The information value is calculated for each branch yielded by the attribute, and the average is taken. This value is subtracted from the current information value of the node, which gives the information gain. The attribute earning the highest information gain is used to construct a rule. A number of branches will specify possible paths from the node. For each of these branches the algorithm will repeat the information gain optimization for the remaining attributes. This will continue until no improvement can be made to the classifying abilities of the decision tree.

In a real industrial environment a few difficulties usually occurs that cannot be handled by the ID3 algorithm alone. As a response to this some further development have been made to the algorithm, resulting in a system for decision tree induction named C4.5 [38]. It increases the robustness of the data handling and enable support for numeric attributes, missing values, noisy data and generating rules from trees.

In the case of noisy data, or when the training set is small, artificial patterns might be detected. For noisy data erroneous instances may cause this, while in the case of small training set chance can result in fictitious regularities appearing. This is known as over-fitting. Formally, it implies that the classification rules perform as well as another collection of rules on the training set but worse on the actual data [39]. To avoid this potential problem reduced-error pruning with a validation set is commonly used. This involves removing nodes in the completed decision tree. Additionally, the branch with the removed node as root is pruned and replaced with a classification according to the most common class in the branch. In practice, this is done by considering each node separately for removal. If the performance with respect to the validation set is not deteriorated, the node is removed.

This will result in a less complex decision tree that performs at least as well as the original tree, in many cases better. However, when training data is scarce, sparing instances to the validation data could decrease the classifying qualities of the decision tree even further [39].

2.4.2 Logistic Regression

Linear regression is a method for fitting a set of data to a linear function of some variables. This is a simple and sometimes suitable way to predict values that varies continuously. The hypothesis representation of linear regression is as follows

$$h_\theta^{Lin} = \theta^T x \tag{10}$$

where θ is the linear parameters of the model and x is a vector containing the input attributes.

Logistic regression is not actually a regression method, but rather a classification algorithm. The description of logistic regression given in this study will be limited to consider problems with only two classes, which could be called 0 or 1. It is in a way an extension of linear regression as it involves the same hypothesis model. However, the hypothesis of logistic regression is the hypothesis of linear regression wrapped between the boundaries 0 and 1, usually with the Sigmoid function [40]. Thus, for logistic regression the hypothesis can be written as

$$h_\theta^{Log} = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}} \tag{11}$$

This can be thought of as the probability that the class of the values for x, parametrized by θ, is positive or 1. If this probability is above 0.5, it is more likely that the example is of positive class than of negative, and the prediction will be positive class. For values of the hypothesis lower than 0.5, negative class or 0 will be predicted.

In order to fit complexed shaped functions into the hypothesis model, input could be mapped into complex functions of the original input attributes. For instance, higher order polynomial combinations could be created. However, this generally implies that a lot more attributes will have to be handled, especially if the original attributes were many. This in turn often causes the decision model building algorithm to require a lot more time.

An efficient way of optimizing the parameters of θ is gradient decent. Basically, the algorithm involves searching for a minimum point of some cost function of some input parameters by looking into the direction yielding the fastest decrease of the function value. This method can be used as an optimization tool in many different machine learning application, only the cost function has to be adapted accordingly. For logistic regression in the simplest case, the cost function can be written as

$$J(\theta) = -\frac{1}{m} \sum_{i=1}^{m} (y^{(i)} \log h_\theta(x^{(i)}) + (1 - y^{(i)}) \log (1 - h_\theta(x^{(i)}))) \tag{12}$$

A thorough analysis and description of different aspects of logistic regression can be found in [41].

2.4.3 Neural Networks

As a response to the problems regarding very large sets of attributes to fit complex functions in implementations of logistic regression, other machine learning techniques can be utilized. Neural networks is a commonly used method that prevents the use of a vast amount of attributes to handle non-linear problems.

Inspired of the way neurons in a biological brain communicates with each other through impulse signals, the neural network model is constituted of connected nodes. These nodes are usually arranged into layers. Common for all layer implementations of neural networks is an input layer and an output layer. The input layer has one node for each attribute that is used as input to the algorithms. The output layer could be constructed with a bit more freedom, although a common conventions for classification problems is to use one node per possible output class. In order to be able to represent more complex patterns, hidden layers is sometimes placed between the input and output layer. The number of output layer and nodes in each layer may vary and the optimal set-up is usually dependent on the problem at hand. Figure 9 illustrates a neural network with two hidden layers.

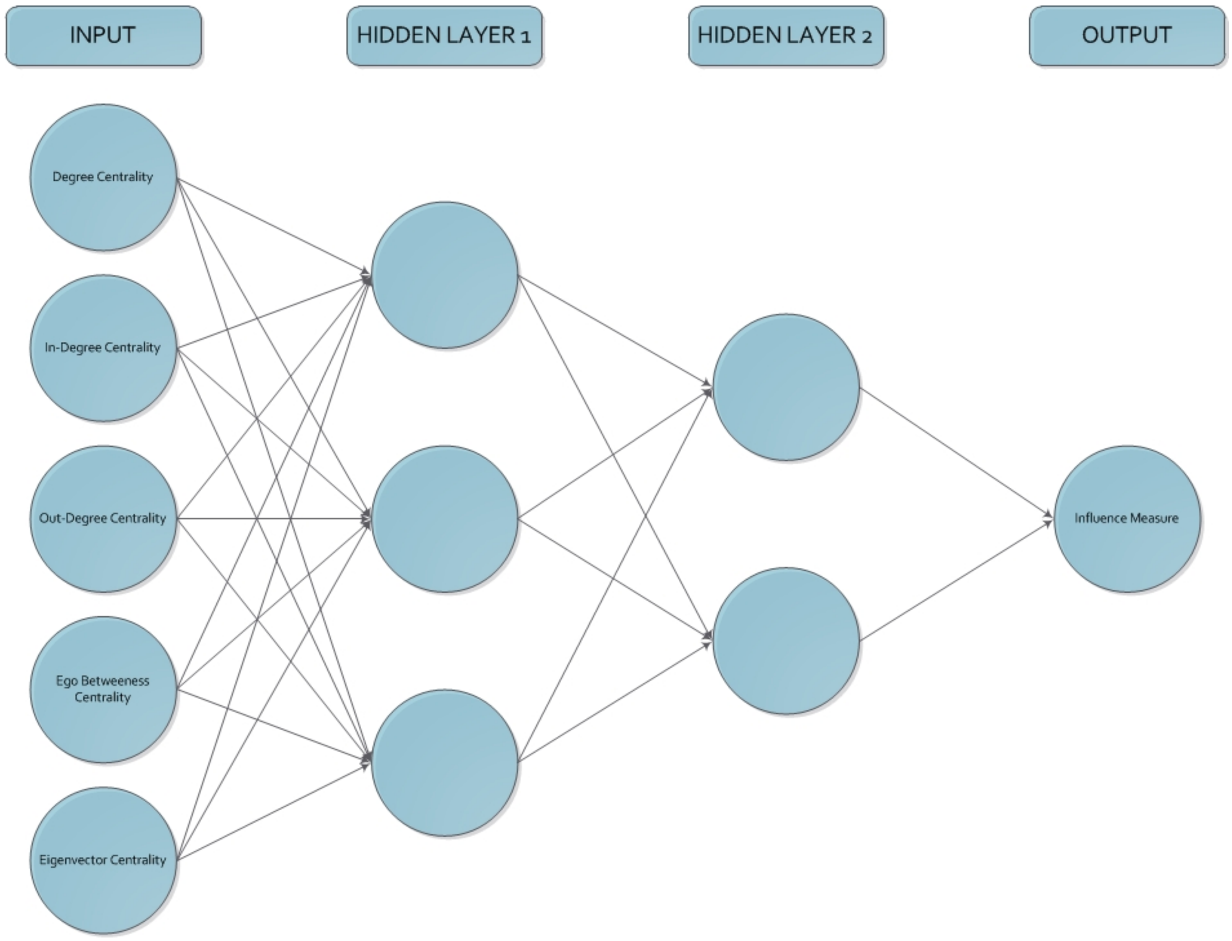


Figure 9: An example neural network having two hidden layers. The input attributes are five different social network analysis metrics and the single output is a measure of influence in the network.

Each node, except for nodes in the input layer, is by its own right a logistic regression model. The arrows in figure 9 indicate that the value obtained at a node in a precedent layer is used as input to the logistic regression model of every node in the subsequent layer. Thus, the value at a node is a weighted sum of the values at the nodes in the precedent layer squeezed into the interval 0 to 1 using the Sigmoid function. All in all, this implies that the neural network algorithm in a way is a combinations of logistic regressions performed in series and in parallel.

As in the case of logistic regression, gradient decent can be used to optimize the weights between the layers in the model. The cost function could in this case be represented as

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}(y_k^{(i)} \log{(h_\theta(x^{(i)}))_k} + (1-y_k^{(i)})\log{(1-(h_\theta(x^{(i)})))_k}) \tag{13}$$

More information regarding neural networks within the field of machine learning can be found in [42]

2.5 Theoretical Summary

The importance of making use of the data available to companies cannot be emphasized enough. In the case of telecommunication operators, this data takes the form of information regarding transactions to, from and within the network.

A telecommunication network can be view as a social network. It has subscriber related to each other through different kinds of friendships. Thus, a telecom network could very well be represented as a social network graph. And in accordance with social network analysis, the telecom network can be analyzed in order to find the influence of each individual subscriber in the network. Especially, a group of such measures that is commonly used for influence estimation is centralities.

A problem with the social networks created from telecommunication networks is the size of them. Networks with more than 10 million subscribers is common in some parts of the world. To be able to analyze such networks within reasonable time, state-of-the-art technology have to be utilized. Two promising ways of handling social network analysis in very large networks is parallelization and graph databases.

When a number of influence measures are obtained, it is not a trivial task to interpret the result. The measures all take into account different aspects of influence and to aggregate the values into a single measure of influence is daunting. A way of dealing with this problem is to implement machine learning. By making use of supervised learning, a decision model can be trained to fit a preferred result. Three methods that has been studied is decision trees, logistic regression and neural networks.

3 Implementation

In this section the systems used to implement the algorithms of social network analysis is covered. This includes a presentation of what input data has been used and how CDRs can be processed into a social network graph. Additionally, a parallel approach for enumerating different aspects of social network analysis is described. Various practical properties of the Hadoop framework is discussed. Finally, a prototype for processing raw CDRs to yield a classification of the subscribers is introduced.

3.1 General

3.1.1 System Setup

Early tests and benchmarking of Neo4j versus Hadoop were performed on a single machine. The operating system of the machine was Windows 7 32-bit. The CPU had a clock rate of 2.56 GHz and the RAM summed up to 4 GB, of which 2.98 GB was available for use. For the test on a single machine, the Java 6 SE environment and MATLAB R2010a was used.

For algorithms implemented in parallel the Hadoop 0.20.2 framework was exploited and a computer cluster was utilized. This consisted of a total of 11 computers. The Master had 16 processors with 4 cores each, running at a clock rate of 2.67 GHz. Furthermore, the Master had a total of 48 GB of RAM. The Slaves had 2 processors, each having 2 cores and running at 3 GHz. The RAM was 4 GB each. As on the single machine the Java 6 SE environment was used. Common for all nodes in the cluster was that they used Ubuntu 10.04.2 LTS. A detailed description of the cluster's technical specifications can be found in Appendix B and an illustration of the cluster can be seen in figure 10.

3.1.2 Input Data

In order to test the algorithms, and examine their scalability properties, synthesized data have been generated. The algorithms for creating these random graphs are the Random algorithm and Erdõs-Rényi algorithm described in section 2.1.3. For the Random algorithm, the degree values have been selected uniformly within the interval 1 to 21. This corresponds to an average degree of about 12. In the Erdõs-Rényi case, the probability of the existence of a relation has been chosen to yield an average degree of 20.

Although synthetized data can be excellent for providing opportunities of testing, the properties of the generated graphs are not necessarily the same as for a real telecommunication network. Thus, the implementations have been tested on CDRs generated from an actual telecommunication network. All in all, data for 2.4 million subscribers and about a month of time were analyzed. The CDRs have been processed in accordance with the procedures described in section 3.1.3.

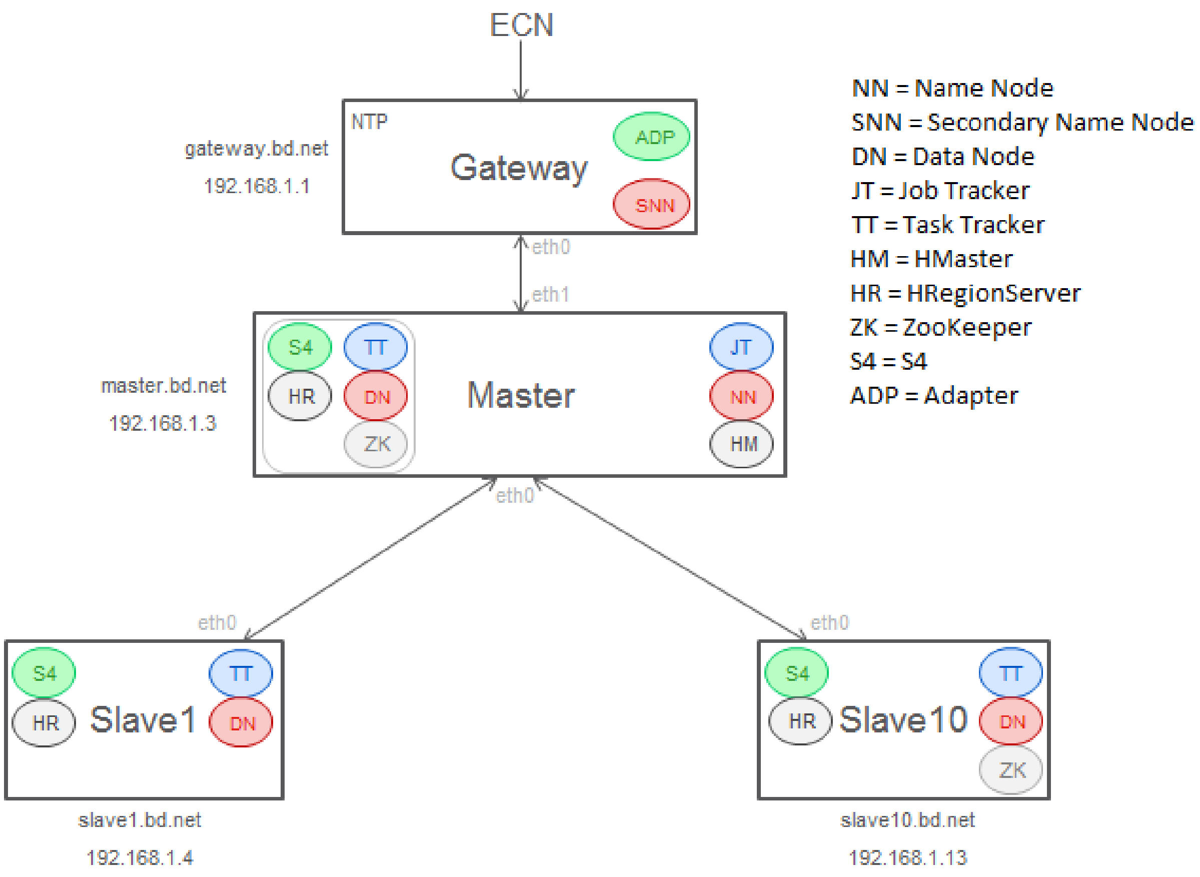


Figure 10: A schematic view of the cluster used to run Hadoop jobs.

3.1.3 Processing CDRs

Within a telecommunication network, information transfers between subscribers are made constantly. These include calls, SMS, MMS, etc. For charging purposes, each such transfer generates a certain amount of information. This information is called a Call Detail Record (CDR) and is stored by the telecommunication operator. By analyzing CDRs, behaviours of subscribers can be deduced.

In terms of graph theory, relations between subscribers can be revealed through studying the CDRs. A transfer between two subscribers seem to indicate that the subscribers are connected, perhaps even know each other. Thus, a graph could be created from the information contained within the CDRs, where vertices would represent subscribers and edges would represent a knows-relation. When implementing this graph generation, some special considerations must be taken into account. These considerations vary depending on the type of graph generated. For this reasons, the following sections are separated to treat unweighted, undirected graphs and weighted, directed graph, respectively.

Unweighted, Undirected Graphs

As the name would suggest, unweighted graphs does not separate relations by weight. Thus, the relations are binary; either they exist or they do not. This simplification of the network involves a loss of information. However, several SNA metrics are specifically constructed to be applied on unweighted graphs and constructing weighted graphs implies a need for more computational power as well as storage space. Hence, unweighted graphs are preferably used as input for some metrics.

When constructing the relations, it is important to use a proper criterion for determining relation existence. A simple choice is to create a relationship whenever a transfer of information has occurred between two subscribers. This will result in a relatively large graph where a lot of information within the CDRs are taken care of. On the other hand, this might yield irrelevant or even inaccurate network structures. For instance, telephone operators, who tend to make a large number of calls but receive none, will obtain an important role in the network. This is in many cases undesirable.

A stricter condition for creating a relationship is to demand calls to be made in both ways. This will effectively remove many of the relations attached to a telephone operator. In addition to this, a threshold could be set with respect to the number of calls in each direction. While this will eliminate some false network structure, it might additionally delete subtleties in the network that actually is an accurate representation of the real underlying network.

Moreover, some previous studies have chosen to omit all calls of less than ten seconds of duration [15]. This as such a short conversation is not likely to indicate a knows-relation, but rather a possible wrong connection. This can equally well be implemented when generating the weighted graph.

Weighted, Directed Graphs

For a weighted graph conditions of relation existence is of less concern then for unweighted graphs. A transactions will simply yield an edge in the given direction between the subscribers. Rather, the difficult issue for the weighted case is how to distribute the weights in accordance with the real underlying network. Three parameters will be taken into account here: recency, type of transaction and, in the case of calls, call duration. These factors will be described below.

The first parameter, recency, describes the amount of time that has passed since the transaction was occurring. A logical conclusion would be that transactions further back in time is less representative of the current state of a subscribers relations than a more recent one is. However, the importance of old calls should not approach zero too fast either, as even transactions far back in time might indicate a certain knows-relation. An exponential function is used to describe the decreasing importance of a transaction as time passes. This can formally be expressed as follows:

$$f_r(t_{trans}) = \left(\frac{1}{2}\right)^{\frac{t_{curr}-t_{trans}}{\lambda}} \tag{14}$$

where t$_{curr}$ is the current time, t$_{trans}$ is the time when the transaction occurred and λ is the half-life of the recency factor.

In the case of type of transaction, a considerably simpler function is used. As text messages generally indicates a more intimate relation, it will be given double importance. Thus, the type of transaction factor can be written as

$$f_t(type) = 2^{\delta(2,type)} \tag{15}$$

where type = 1 represents a call and type = 2 a text message. δ(i,j) is the dirac-delta, as defined in equation 6. Lastly, if the transaction is a call, a factor for call duration is taken into account. In this case, a longer call will usually mean a closer relationship between the subscribers involved. The factor should thus increase with time, but not towards infinity. It should approach a constant value, one for instance, as call duration increases. Furthermore, a very short call should also have some impact on the weight. Thus the function representing the call duration factor needs to have an initial value, e.g. a half. A function fulfilling these properties is the Sigmoid function, mentioned in section 2.4.2 and described in [40], which can be represented as

$$f_d(t_{dur}) = \frac{1}{1+e^{-kt_{dur}}} \tag{16}$$

where t$_{dur}$ is the call duration and k is an adjustable constant.

The total weight of a relation between two subscriber in a specific direction is the sum of the resulting product of the three factors described above. Hence

$$w = \sum_i dw_i \tag{17}$$

where i runs over all transactions and

$$\begin{aligned} dw((t_{trans}, type, t_{dur}) = f_r(t_{trans})f_t(type)f_d(t_{dur}) = \\ = \left(\frac{1}{2}\right)^{\frac{t_{curr}-t_{trans}}{\lambda}} 2^{\delta(2,type)} \left(\frac{1}{1+e^{-kt_{dur}}}\right)^{\delta(1,type)} \end{aligned} \tag{18}$$

3.2 Hadoop

Hadoop, as well as MapReduce-like frameworks in general, can be used to deal with a great variety of problems. The only constraint on the problem formulation is that the input and output can be represented as a key-value pair. One of the main strengths of Hadoop is that it enables scalability of computational time as data sets increase in size. Performance with respect to time can easily be improved by adding more nodes to the computer cluster. However, with the concept of parallelization, certain issues arises. How to handle these issues in a social network analysis context is described in detail in Appendix A.

3.3 Prototype

Recently, telecommunication operators have seen a significant increase in data related to subscriber activity generated in their networks. This data represents an opportunity to gain competitive advantage, if utilized properly. The utilization involves processing vast amounts of data into information and analyzing this information. Specifically, classification or ranking of subscribers with respect to some predefined attribute, such as influence within the network, is important applications. A system for performing this complete process is proposed here and will henceforth be referred to as the prototype. The system will be described in detail in this section.

3.3.1 Overview

There are a large number of social network analysis algorithms and methods that can be used to find valuable information about the social network representing the telecommunication users, derived from mobile network data. One is centrality methods, which can be applied to find the most central nodes in the network. A common approach of analyzing social networks is by considering these centrality measures separately. By doing this only one aspect of influence is taken into account and some information existing within the network is ignored. A more accurate analysis could be performed by including as much information as possible e.g. hidden patterns of the relations between the different flavors of importance.

However, it is a challenge for the network operator to know which algorithms to use and how to evaluate the resulting measures from all different algorithms in order to pinpoint the most influential subscribers or other interesting segments of subscribers. In order to deal with this difficulties machine learning algorithm is implemented. The general idea is to generate a social network graph from the information in a CDR-file. Using this graph, a number of SNA metrics is calculated. At this point the machine learning algorithm is implemented to created a model for handling each subscriber. For instance, this can be a classification model where each subscriber is classified with respect to an attribute. To be able to create the model, a training set must be provided. The training set is constituted of a sub-set of the network in which, for each subscriber, the predefined attribute is known exactly. This model can then be used to gain information regarding the subscribers of the complete network. Figure 11 gives a synoptic view of the prototype.

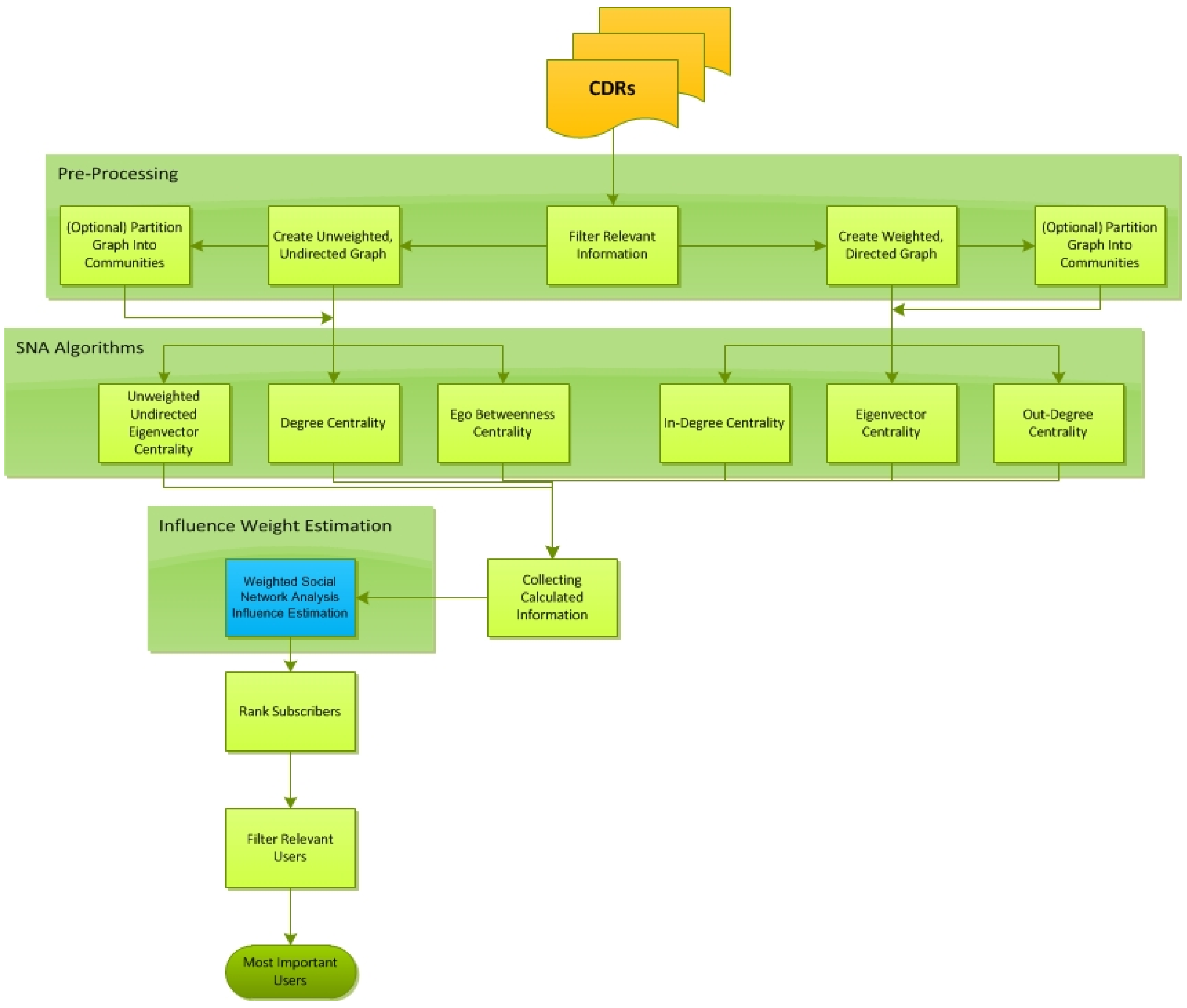

CDRs
Pre-Processing
(Optional) Partition Graph Into Communities
Create Unweighted, Undirected Graph
Filter Relevant Information
Create Weighted, Directed Graph
(Optional) Partition Graph Into Communities
SNA Algorithms
Unweighted Undirected Eigenvector Centrality
Degree Centrality
Ego Betweenness Centrality
In-Degree Centrality
Eigenvector Centrality
Out-Degree Centrality
Influence Weight Estimation
Weighted Social Network Analysis Influence Estimation
Collecting Calculated Information
Rank Subscribers
Filter Relevant Users
Most Important Users

Figure 11: The proposed prototype. Some of the processes could be replaced with other, similar, processes.

As this process generally is done on very large networks, a parallel implementation is beneficially used to ensure scalability. Thus, the prototype has been implemented in the Hadoop framework, described in section 2.2.1. To be precise, almost every box in figure 11 can in principle be perform by a MapReduce job, or in some cases an iteration of several jobs. For testing the complete system, the real network data described in 3.1.2 is used as input.

The prototype can be divided into three main parts, according to figure 11. These are Pre-Processing, SNA Algorithms and Influence Weight Estimation. Each of the parts of this system will be described in further detail in the following sections.

3.3.2 Pre-Processing: Filtering CDRs and creating network graphs

When access to the CDRs is established, the first part of the pre-processing involves filtering each record and only keeping the relevant information. This typically involves removing records that are missing or having irregularly formated entries. The vertices to be included in the graphs will be those found in the CDRs after filtering has taken place. These filtered records are the input of the procedure of creating network graphs, both a weighted and directed and an unweighted and undirected, as described in 3.1.3.

An optional extension to the model could be to add a partitioning step. The graph would then be split into communities and the analysis, which will be described in the next section, will be done on each community separately. An algorithm for such partitioning is described in 2.1.5 and [29].

To test the prototype, all records in the particular CDR containing 89 or 108 entries where kept. The ID-numbers of the caller and receiver were constrained to have 13 or 14 characters. This was implemented as the ID:s of longer and shorter number combinations showed odd behaviors and is viewed as irrelevant. When creating the unweighted and undirected graph no threshold were demanded, a single call in any direction is sufficient for the existence of a relation. Furthermore, the rule of filtering out calls of less then 10 seconds, as mentioned in 3.1.3, has been omitted.

3.3.3 Calculation of SNA metrics

This part involves determination of several of the SNA metrics described in section 2.1.4. The prototype tested has 9 algorithms implemented. These are total degree centrality, in-degree centrality, out-degree centrality and eigenvector centrality. These metrics are calculated both in the weighted and the unweighted case. Additionally, ego betweenness centrality is calculated for the unweighted case. As this algorithm do not extend to the weighted case in a natural way, such an implementation was not tried. The output from this part of the prototype is a list of all subscribers with the 9 corresponding SNA metrics. This list constitutes the input for the next sector, which is described next.

3.3.4 Importance Determination Using Supervised Learning

This step of the process aims to perform as much of the data interpretation as possible, to relieve this responsibility from the user of the system. This interpretation is performed by the use of pattern detection in line with common data mining techniques, sometimes referred to as machine learning. The input to a machine learning algorithm of the category supervised learning is a list of examples where a number of attributes is given. One of these attributes is the actual value of unit to predict, called the concept. In the case of the tests made for this report, influence within the network is to be predicted. The most natural and perhaps the best way of obtaining an accurate training set is to let the user provide it. For instance, it is likely that the network operator have performed marketing campaigns in the past. Analysis of the result of such a campaign could very well yield information regarding individual subscribers ability to spread information through the network, by for instance studying the amount of buys of the marketed service that has occurred among friends of the subscribers observed.

If this kind of training set is absent however, there are ways to generate a training set. This approach has been used during the tests of the system. The measure of influence chosen has been that of exact betweenness centrality. This algorithm is very difficult to perform in reasonable time for large networks. Thus, calculating betweenness centrality for the whole network is an inappropriate way of facing the problem. A more suitable approach is to do the determination of betweenness centrality for only a subset of the complete network. Be sure to note that it is not at all guaranteed that betweenness centrality is an accurate measure of influence in the network. Rather, it is an appropriate measure to use for generating the training set as it is not represented among the attributes for each subscriber. Thus, the machine learning algorithms will probably not be able to find any trivial decision models for classifying subscribers.

A fast way of dividing graphs into sub-graphs is fast-unfolding community detection, described in section 2.1.5. This will ensure that few relations to subscribers outside the sub-graph are removed and the value of betweenness centrality calculated within the community will be approximately the same, for the subscribers in the community, as if the calculation were made on the whole network. A community of 20737 subscribers were found to be appropriately large and was used as training set.

The values obtained by these calculations will hence act as the concept to be predicted by a combination of the other attributes, namely the 9 social network analysis metrics mentioned above. During the experiment for testing the prototype the centrality calculated for the community were used to rank the subscribers. The top 30 % of the community subscribers, with respect to betweenness centrality, is regarded as influential. The remaining subscribers are considered non-influential. The objective function was then to predict the class of a specific subscriber.

There are a number of ways to fit the example attributes to the given concept. In common for these methods is that they can be seen as data mining or machine learning techniques. Examples are different kinds of regression, neural networks as well as rule based decision models. When implemented to deal with the classified data decision tree generating algorithms was used to create the decision model, in line with the general ideas formalized in section 2.4.1. Information about the algorithms related to logistic regression and neural network can be found in section 2.4.2 and 2.4.3, respectively. All machine learning algorithms were performed using the Weka 3.6.5 toolbox.

To estimate the performance of the different machine learning methods a few standard performance measures have been calculated. Specifically, these are precision, recall and

F-value. Simply put, precision is the number of correctly classified instances of the positive class, in this case the influential subscribers, divided by the total number of positively classified instances. Recall is the number of correctly classified instances of the positive class over the total number of instances of positive class in the validation set. Lastly, F-value is a combination of precision and recall which can be expresses as

$$F = 2\frac{PR}{P + R} \tag{19}$$

where P is the precision and R is the recall.

4 Results

In this section results regarding the performance of the implementations are presented. The section 4.1 comprises the first tests made to compare the Hadoop and Neo4j systems. Hadoop turned out to be a more promising solution and was tested further. Tests were performed both on synthesized and real data. Additionally, the prototype described in section 3.3 was tested. Finally, the results obtained while analyzing a real telecommunication network is compared statistically.

4.1 Early Tests

4.1.1 Neo4j

Tests have been made in order to determine the performance of the graph database Neo4j. All test were performed on a single machine as described in section 3.1.1. Figure 12 shows the computational time of calculation of ego betweenness centrality for different network sizes using the Neo4j database system. The algorithm used synthesized data as input.

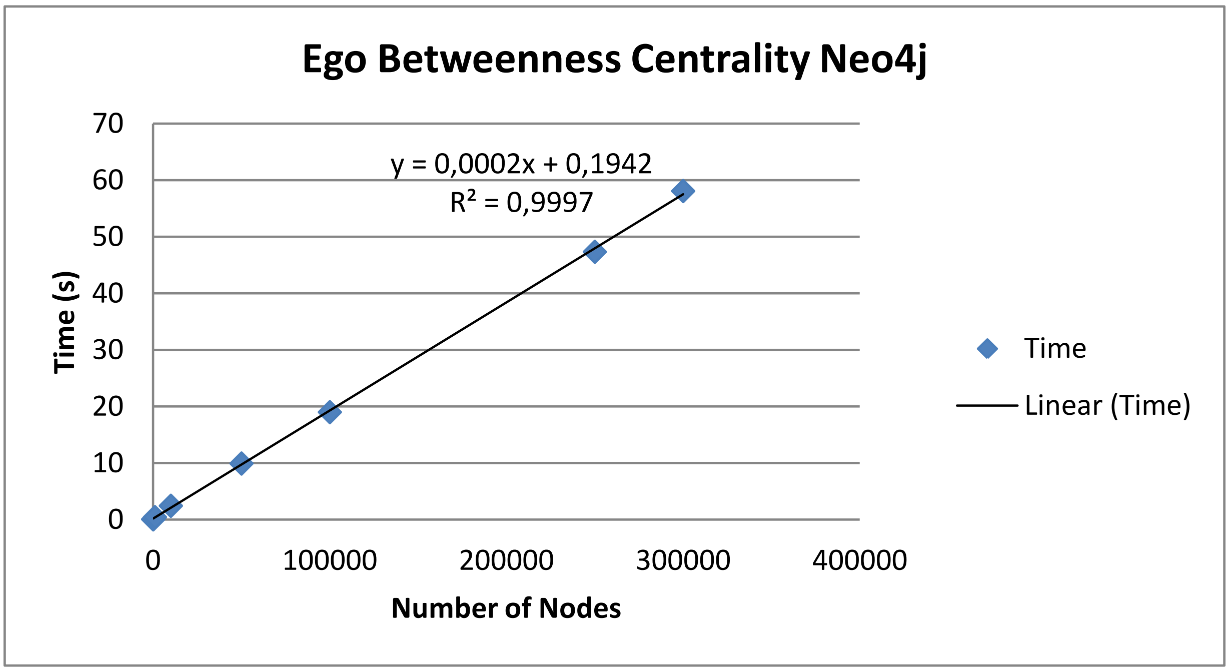


Figure 12: Computational time of the algorithm for finding ego betweenness centrality, implemented in Neo4j.

Notice especially that the computational time scales linearly with respect to number of vertices in the network. However, this behavior is only observed until a certain limit, at about 300000 subscribers.

4.1.2 Hadoop on a Single Machine

As Neo4j has only been tested in a single machine setup, some tests of the Hadoop framework has also been performed on a single machine. In particular, test of ego betweenness centrality has been implemented and the results are illustrated in figure 13.

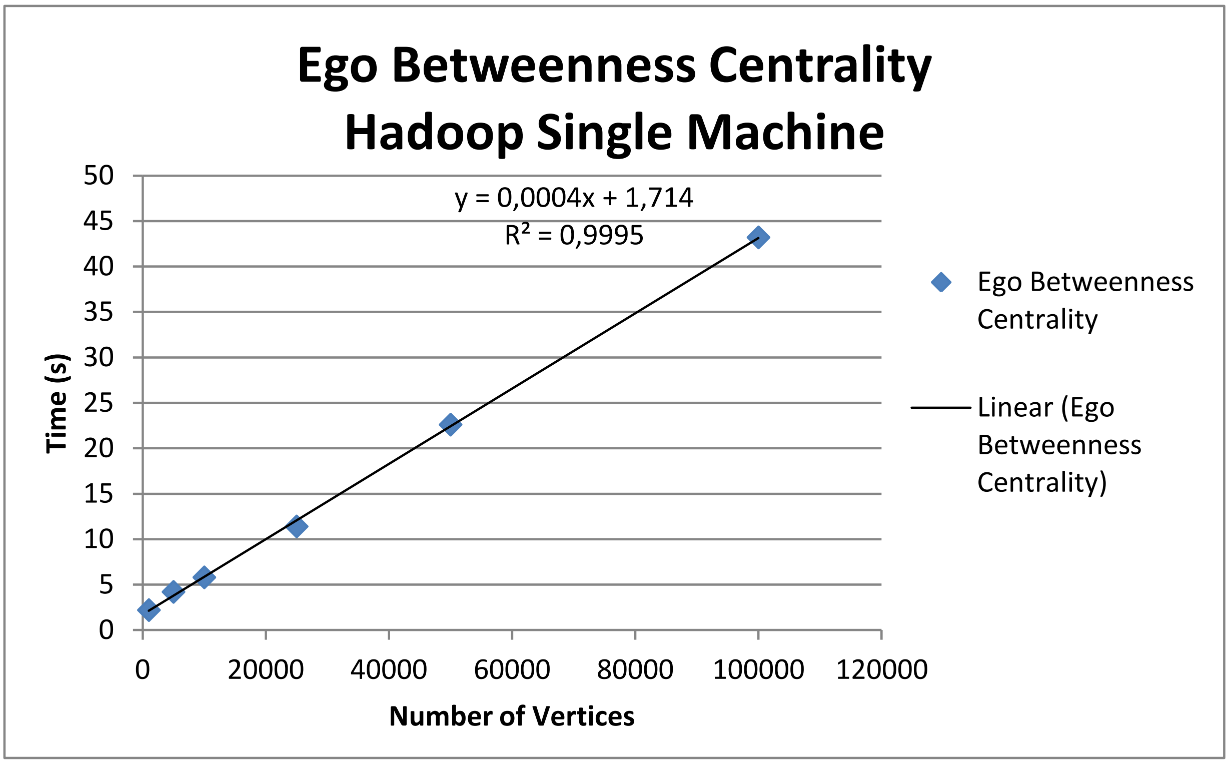


Figure 13: Computational time of the algorithm for finding ego betweenness centrality, implemented in Hadoop and ran on a single machine.

As in the Neo4j case a linear scalability is obvious in figure 13. A graph showing both curves in comparison is provided in figure 14.

4.2 Hadoop in Cluster Mode

4.2.1 Scalability with Respect to Number of Machines

In order to determine the reliability of the assumption that adding machines to a computer cluster will reduce computational time, tests with varying number of machines have been performed. The computational time as a function of number of processors in the computer cluster is displayed in figure 15 for degree centrality and in 16 for ego betweenness centrality. A 10 million network, generated using the Erdõs-Rényi algorithm, has been used as input in both cases.

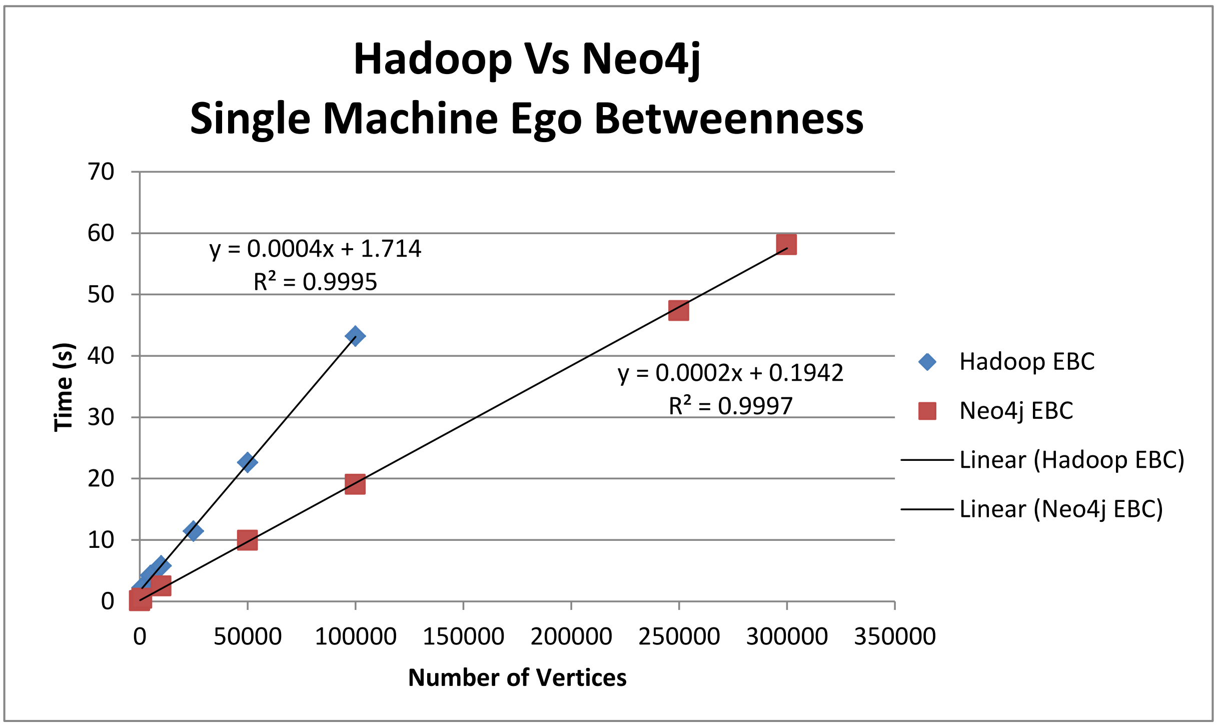


Figure 14: Computational time of the algorithm for finding ego betweenness centrality, implemented in Hadoop and ran on a single machine.

For the ego betweenness centrality, an even decrease is obvious. A similar trend can be made out for the test regarding degree centrality.

The relative speed-up with respect to number of cores in the computer cluster for both these algorithms is shown in figure 17. The red squares connected by a black line represents linear speed-up.

The relative speed-up of ego betweenness centrality is actually found to be faster than linear, whereas that for degree centrality is somewhat slower. An exception to this trend is the result found for calculating degree centrality on a 30 processors system. The measured value for that set-up yields a relative speed-up larger than the linear comparison.

4.2.2 Full Cluster Tests

To test the scalability of the separate algorithms, various experiments have been run on a computer cluster, as in section 3.1.1. All networks used as input in these tests were generated synthetically using a uniform degree distribution or the Erdõs-Rényi algorithm, as described in 2.1.3. The algorithms tested are degree centrality, ego betweenness central-

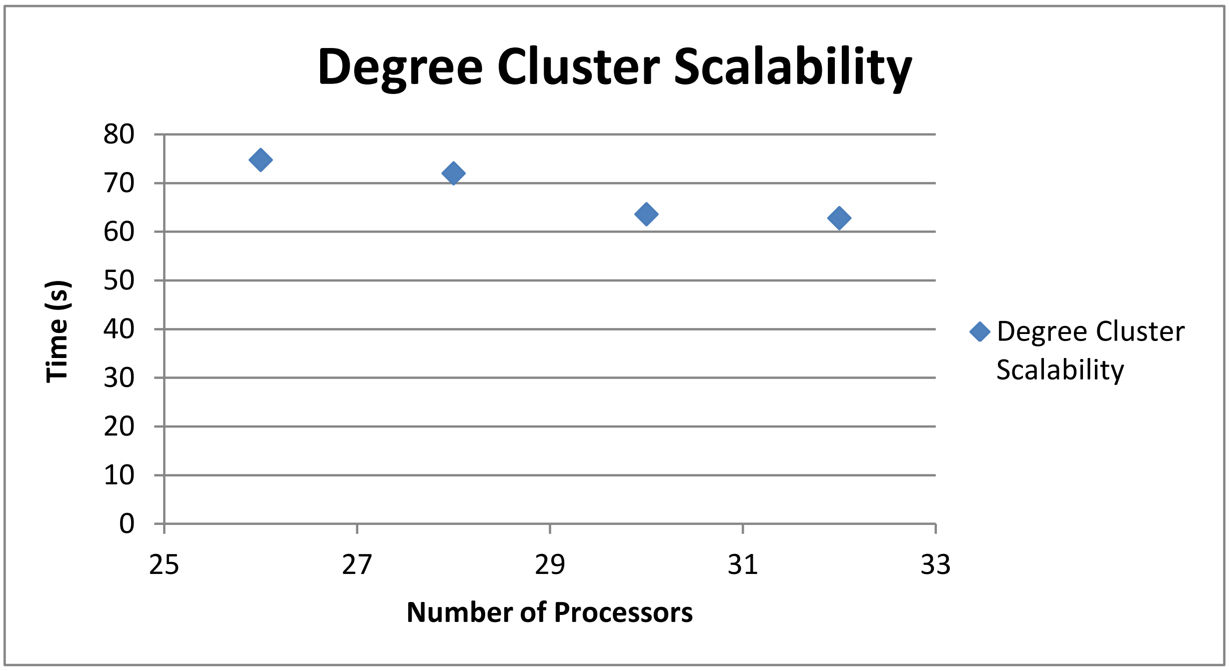


Figure 15: Computational time of degree centrality of 10 million vertices network, generated with the Erdõs-Rényi algorithm, implemented in Hadoop as a function of the number of processors in the network.

ity, eigenvector centrality and betweenness centrality. The gathered result is depicted in figure 18

The results of these test implies that degree centrality is the fastest algorithm to execute. To calculate this measure for a network 100 million vertices less than 4 minutes are needed. Figure 18b indicates that the algorithm for calculating ego betweenness centrality is also able to handle network of a 100 million vertex size. However, the time required for the computations are two orders of magnitude higher than for the degree centrality case, at about 4.5 hours for 100 million vertices. Eigenvector centrality implies even more time complexity and tests have not been performed on networks larger than 2500000 vertices. For a network of that size the computational time amounts to a bit more than an hour, as shown in figure 18c. For all these three metric algorithms a linear scalability with respect to number of vertices in the network is apparent. This linearity is not shared by the betweenness centrality algorithm. As suggested in figure 18d the increase of computational time as a function of the number of vertices resembles a second order polynomial trend. Additionally, this algorithm is unable to deal with large networks as a network as small as 25000 vertices demands more than 16 hours of computations.

Similar tests have been performed for graphs randomized in accordance with the Erdõs-Rényi algorithm, covered in section 2.1.3. As with the data sets with uniform degree distribution, the algorithms degree centrality, ego betweenness centrality, eigenvector centrality

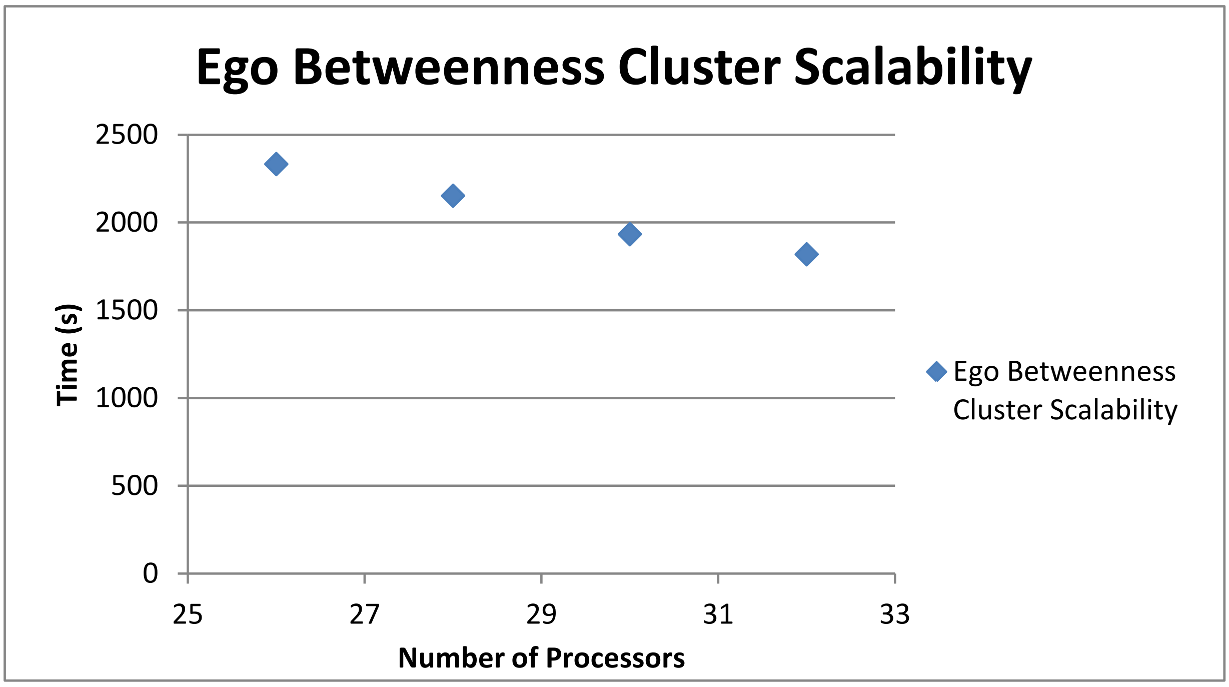


Figure 16: Computational time of ego betweenness centrality of 10 million vertices network, generated with the Erdõs-Rényi algorithm, implemented in Hadoop and ran on a single machines as a function of the number of processors in the network.

and betweenness centrality have been tested. The result can be studied in figure 19.

Overall, the same trends are discovered here as in figure 18. Degree centrality is the fastest algorithm investigated and a graph of 10 million vertices is analyzed in less than a minute. All algorithms except betweenness centrality are found to be linearly scalable and this trend is most obvious in the result obtained for the ego betweenness centrality algorithm, shown in figure 19b. Additionally, as in previous experiments, the betweenness centrality algorithm are obeying tendencies similar to that of a second order polynomial.

Note that the results yielded from the different data sets are not immediately comparable. This as the properties of the synthesized graph are different depending on method used to generate them. Additionally, for the tests concerning eigenvector centrality, different convergence thresholds have been used. The threshold was the difference in eigenvector centrality for each single vertex of 0.0001 for the Random graphs and 0.01 for the Erdõs-Rényi graphs. In the latter case, this has been done to speed up the testing of the algorithm.

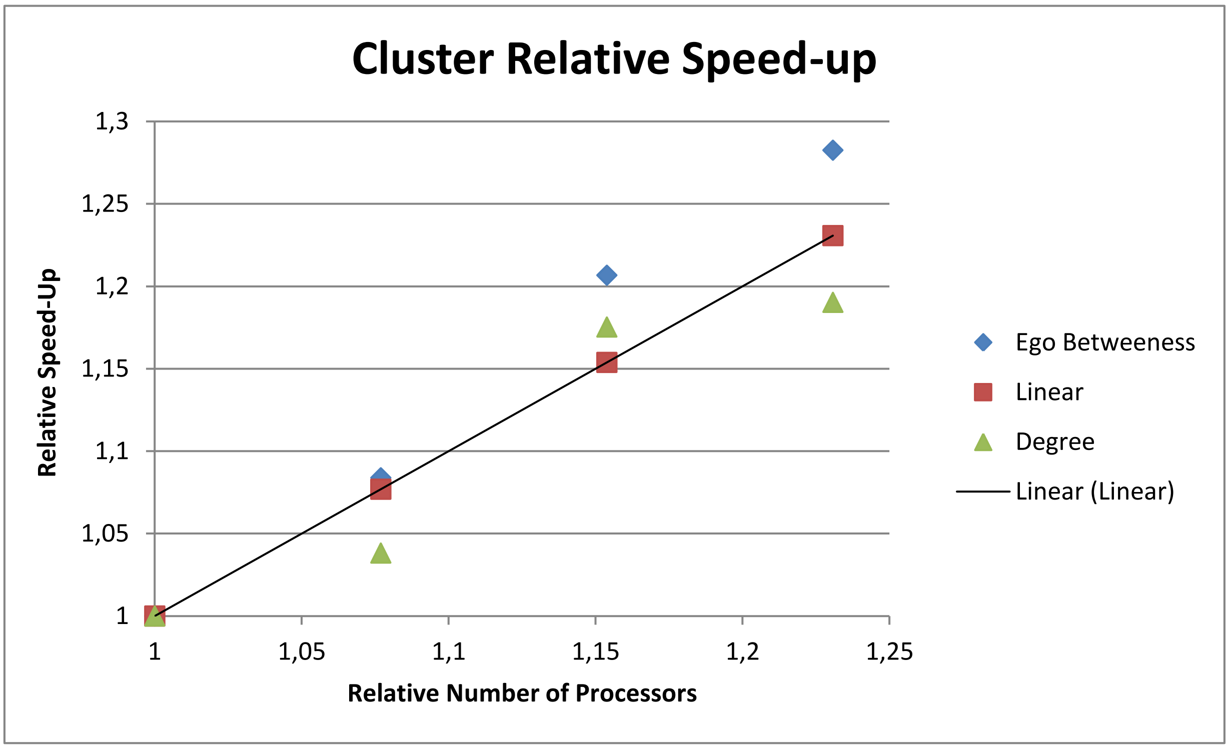


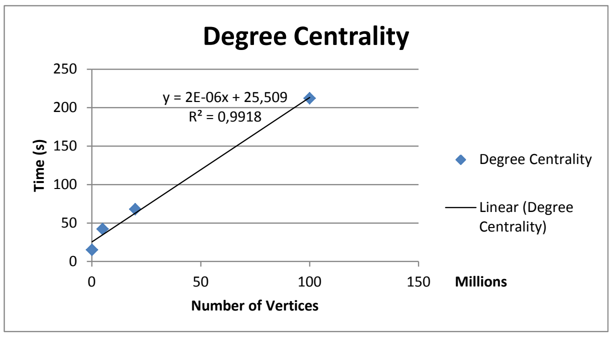
Figure 17: Computational time of the algorithm for finding ego betweenness centrality, implemented in Hadoop and ran on a single machine.

4.3 Prototype

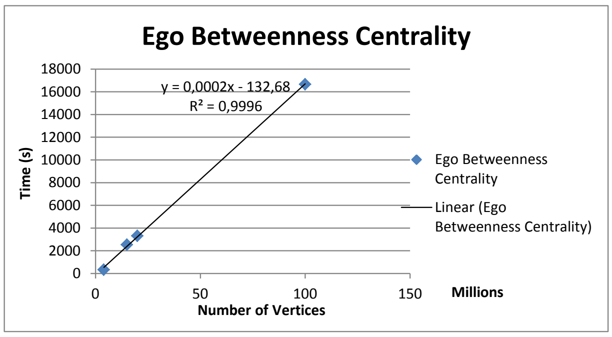
4.3.1 Computational Time

This section presents results obtained when testing the prototype on a real CDR data set. The data consists of 2.4 million subscribers and a more complete description of the data set can be found i section 3.1.2. All processes described in section 3.3 have been performed, although in some cases several of the described processes is rendered in a single MapReduce job. Figure 20 presents most of the jobs performed and the computational time required for each of them. The results for the machine learning module, labeled "Influence Weight Estimation" in figure 11, is presented in the next section. Note that the dismantlement of the the graph into communities were performed on a single computer with the set up described in section 3.1.1.

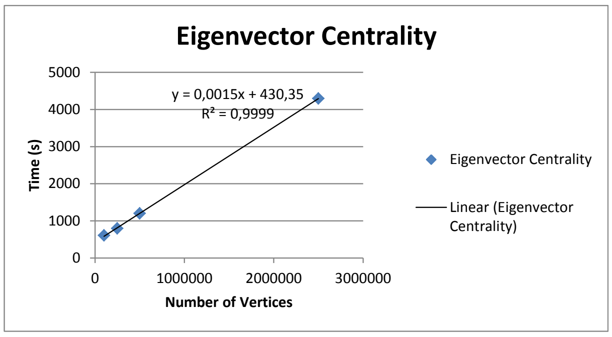
Of interest is to note that most of these processes can be carried out in a time of the order of an hour or less. The two exceptions are the iterative jobs, calculation of unweighted and weighted eigenvector centrality, which both needed more than 24 hours for completion.



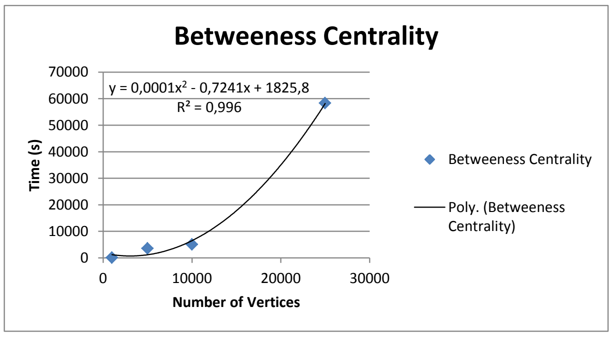
(a) Computational time degree centrality.



(b) Computational time ego betweenness centrality.



(c) Computational time eigenvector centrality.



(d) Computational time betweenness centrality.

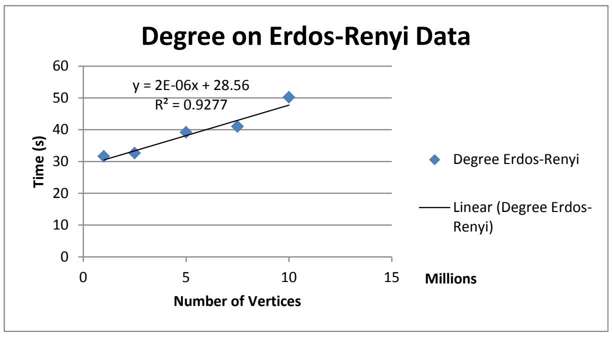
Figure 18: Time needed to compute different centrality measures using the computer cluster described in section 3.1.1. The input data was generated in accordance with the Random algorithm described in section 2.1.3.

4.3.2 Machine Learning Algorithms

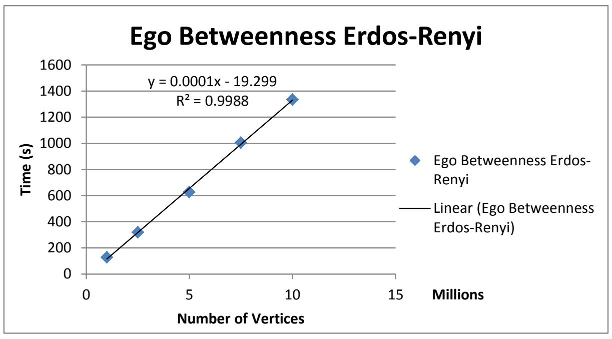
This sections is dedicated to the performance of the machine learning techniques used to create models for classifying subscribers of the telecom network. Three main categories of techniques have been considered, namely decision trees, neural networks and regression. The results for each category is presented in figure 21 below. The training set consists of 66 % of the subscribers in the community of 20737, mentioned in section 3.3.4. These 66% were picked randomly and the remaining subscribers were used as a validation set.

The best method in terms of F-value from the class of decision tree techniques was found to be the one labeled REPTree, with an F-value of 0.496. However, when studying the ability to correctly classify instances the methods ADTree and J48 share best results of 73.4931 %.

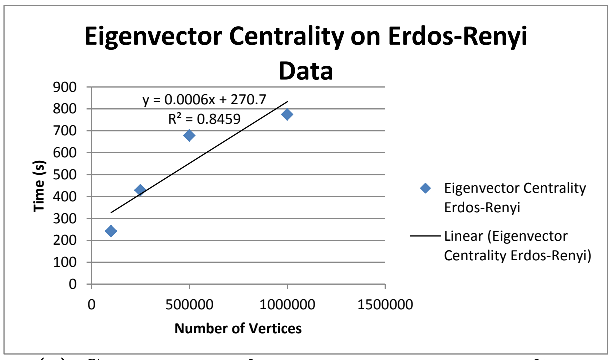
Less accurate results, with respect to F-value, are obtained for the regression algorithms. Both Logistic and SimpleLogistic have an F-value of just above 0.44. Studying the percent of correctly classified instances actually indicates higher accuracy than the decision



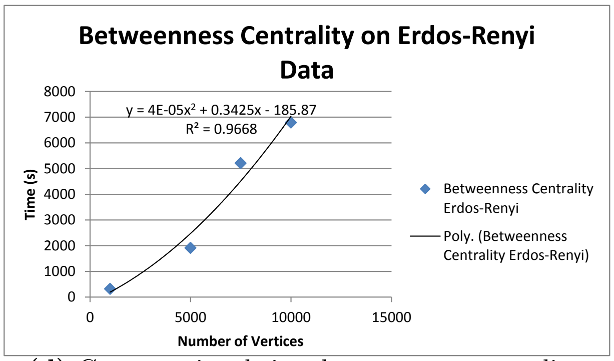
(a) Computational time degree centrality.



(b) Computational time ego betweenness centrality.

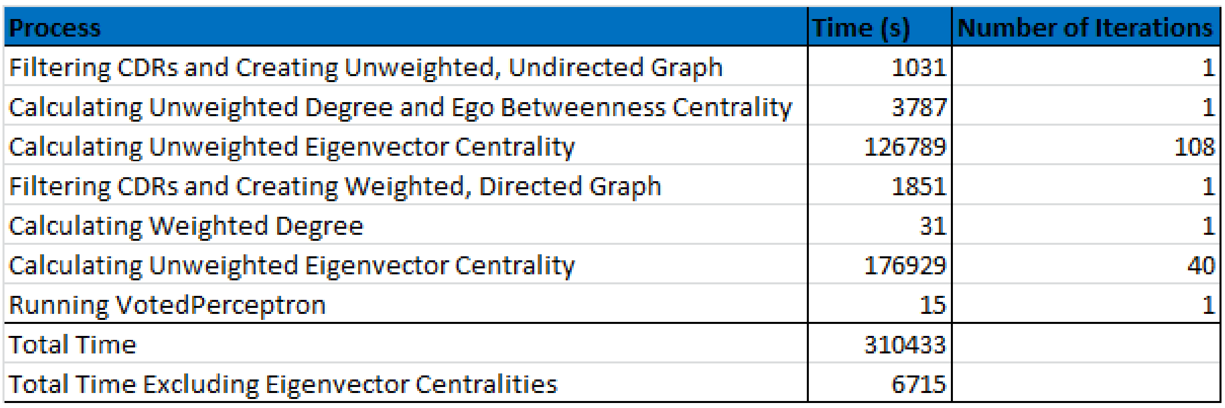


(c) Computational time eigenvector centrality.



(d) Computational time betweenness centrality.

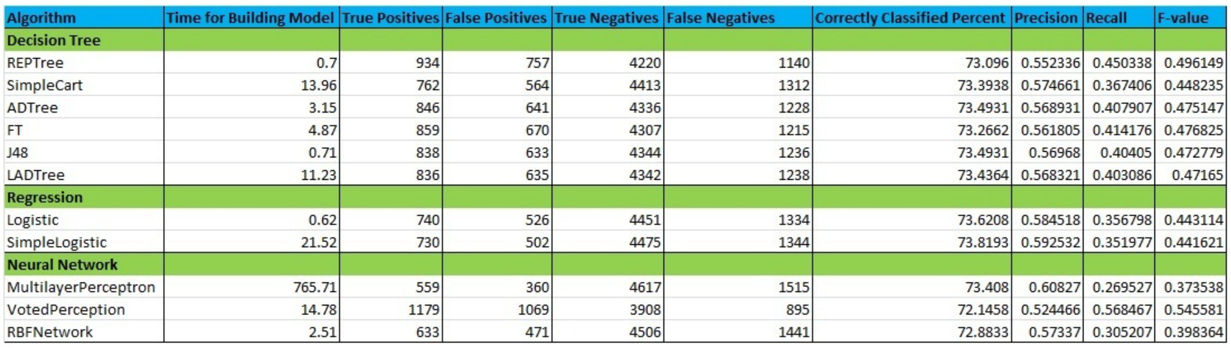
Figure 19: Time needed to compute different centrality measures using the computer cluster described in section 3.1.1. The input data was generated in accordance with the Erdõs-Rényi algorithm.



Process	Time (s)	Number of Iterations
Filtering CDRs and Creating Unweighted, Undirected Graph	1031	1
Calculating Unweighted Degree and Ego Betweenness Centrality	3787	1
Calculating Unweighted Eigenvector Centrality	126789	108
Filtering CDRs and Creating Weighted, Directed Graph	1851	1
Calculating Weighted Degree	31	1
Calculating Unweighted Eigenvector Centrality	176929	40
Running VotedPerceptron	15	1
Total Time	310433	
Total Time Excluding Eigenvector Centralities	6715	

Figure 20: Time needed to perform the steps in the prototype. The data set used as input is based on a real telecommunication network with 2.4 million subscribers, observed during one month.

tree techniques. The highest value is obtained for the SimpleLogistic algorithm that manage to classifiy 73.8193 % of the instances in the validation set. This value is unmatched



Algorithm	Time for Building Model	True Positives	False Positives	True Negatives	False Negatives	Correctly Classified Percent	Precision	Recall	F-value
Decision Tree									
REPTree	0.7	934	757	4220	1140	73.096	0.552336	0.450338	0.496149
SimpleCart	13.96	762	564	4413	1312	73.3938	0.574661	0.367406	0.448235
ADTree	3.15	846	641	4336	1228	73.4931	0.568931	0.407907	0.475147
FT	4.87	859	670	4307	1215	73.2662	0.561805	0.414176	0.476825
J48	0.71	838	633	4344	1236	73.4931	0.56968	0.40405	0.472779
LADTree	11.23	836	635	4342	1238	73.4364	0.568321	0.403086	0.47165
Regression									
Logistic	0.62	740	526	4451	1334	73.6208	0.584518	0.356798	0.443114
SimpleLogistic	21.52	730	502	4475	1344	73.8193	0.592532	0.351977	0.441621
Neural Network									
MultilayerPerceptron	765.71	559	360	4617	1515	73.408	0.60827	0.269527	0.373538
VotedPerception	14.78	1179	1069	3908	895	72.1458	0.524466	0.568467	0.545581
RBFNetwork	2.51	633	471	4506	1441	72.8833	0.57337	0.305207	0.398364

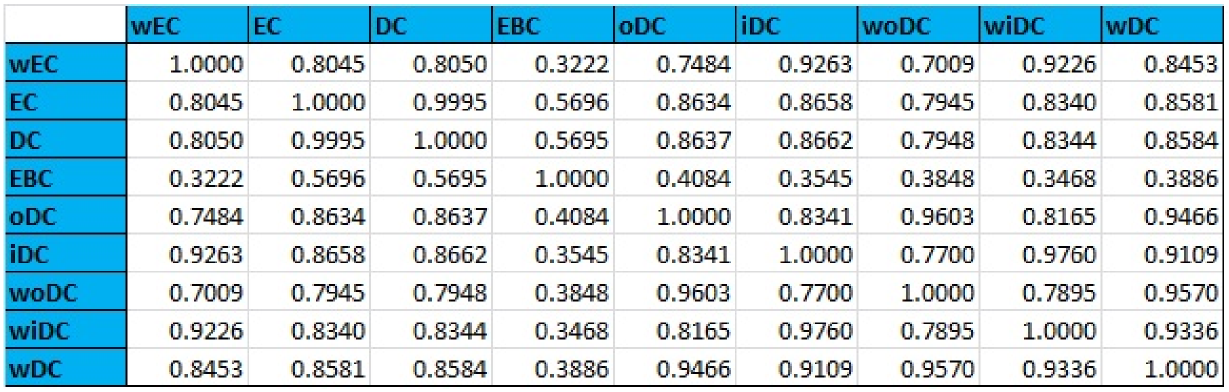
Figure 21: Performance, in terms of accuracy and computational time, of various machine learning algorithms.

among all machine learning methods tested.

The neural network algorithm VotedPerceptron is the one method obtaining highest F-value, about 0.546. This was the highest value found among any of the machine learning algorithms. On the other hand, the percent of correctly classified instances was a little bit lower than for other algorithms. Note that this number differs only slightly between the neural network methods studied.

4.4 Statistical Relations Between Metrics

Figure 22 shows correlations between the 9 social network analysis metrics calculated for the real CDR data set. The correlations are normalized, implying that the upper limit is 1.



	wEC	EC	DC	EBC	oDC	iDC	woDC	wiDC	wDC
wEC	1.0000	0.8045	0.8050	0.3222	0.7484	0.9263	0.7009	0.9226	0.8453
EC	0.8045	1.0000	0.9995	0.5696	0.8634	0.8658	0.7945	0.8340	0.8581
DC	0.8050	0.9995	1.0000	0.5695	0.8637	0.8662	0.7948	0.8344	0.8584
EBC	0.3222	0.5696	0.5695	1.0000	0.4084	0.3545	0.3848	0.3468	0.3886
oDC	0.7484	0.8634	0.8637	0.4084	1.0000	0.8341	0.9603	0.8165	0.9466
iDC	0.9263	0.8658	0.8662	0.3545	0.8341	1.0000	0.7700	0.9760	0.9109
woDC	0.7009	0.7945	0.7948	0.3848	0.9603	0.7700	1.0000	0.7895	0.9570
wiDC	0.9226	0.8340	0.8344	0.3468	0.8165	0.9760	0.7895	1.0000	0.9336
wDC	0.8453	0.8581	0.8584	0.3886	0.9466	0.9109	0.9570	0.9336	1.0000

Figure 22: Correlation between 9 different social network analysis metrics, calculated for an actual CDR data set generate from a telecommunication network.

The abbreviations of the metrics is as follows; w stands for weighted, o for out and i for in. If w is not present, the metric is unweighted. C represents centrality and D is degree, E is eigenvector and EB is ego betweenness. Note especially that ego betweenness centrality do not correlate strongly with any of the other metrics, the strongest being that to eigenvector centrality, amounting to about 0.57. Weighted degree centrality, on the other hand, has quite pronounced correlation to all other metrics, ego betweenness excluded. Additionally, a particularly great correlation of 0.9995 can be seen between unweighted degree centrality and eigenvector centrality.

Apart from the test performed on the complete data set, one test were made on the smaller training set, corresponding to a community of the network. The community is the same as described in 3.3.4, consisting of 20707 subscribers. The test involved ego betweenness centrality and exact betweenness centrality, to determine the accuracy of the approximation ego betweenness centrality implies. The correlation of the two metrics was calculated to 0.3389.

5 Discussion of Results

The scope of this section is a discussion and analysis of the results obtained during the tests. First, the results of the comparison between Neo4j and Hadoop is investigated. This comparison was done on a single machine, thus a natural extension is to try the solution on a computer cluster. Initially, parts of the cluster are tested and following that the complete cluster. The analysis is presented in the same order. After that, a discussion of the statistical relation between the social network analysis measures obtained from the real telecommunication network are explored. Additionally, the results related to the prototype are looked at.

5.1 Neo4j Versus Hadoop

In section 4.1 experiments comparing the Hadoop framework and the graph database Neo4j on a single machine are performed. A collation of figure 12 and figure 13 hints that Neo4j is the faster solution, as it for instance manages to compute ego betweenness centrality more than twice as fast for a network of 100000 vertices. However, in the case of Neo4j more heap space are utilized as the network sizes are increased. This results in more time spent on garbage collecting, relatively. This is a quite slow process and affects the total time significantly for sufficiently large networks. Ultimately, if the machine is unable to clean the cache in a reasonable time, a heap space error is thrown and the process is stopped. Thus, there is a distinct limit to the size of the networks that can be used in Neo4j on a single machine.

The Hadoop framework, on the other hand, do not suffer from that type of problem. This is due to the fact that Hadoop deals with a job in smaller processes, called tasks. The tasks does not share heap space and hence they are not as prone to heap space problems as the Neo4j system is. It should be noted that the Hadoop framework do not scale linearly indefinitely on a single machine. For larger inputs the time spent on sorting the output of the map tasks grows and will eventually be a dominating part of the job.

5.2 Machine Quantity Impact on Computational Time

A decrease in computational time is experienced, both for calculating the degree centrality and the ego betweenness centrality on a 10 million vertex network, when the number of processors in the computer cluster is increased. Thus, it is reasonable to believe that larger computer clusters could handle jobs faster, and requirements on low computational time could potentially be met by running processes on larger clusters. Additionally, figure 17 seems to suggests that the increase of computational speed is more prominent for ego betweenness centrality than for degree centrality. An explanation could be that calculating degree centrality for a network of 10 million subscribers is a quite small job. Thus, staring up and preparing the Hadoop system for the job will be a significant part of the computational time and the actual processing phase will have a lesser impact. This would imply that Hadoop is more suitable for larger jobs.

In figure 17, one point deviates quite clearly from the others, namely the one representing calculations of degree centrality on a 30 processors system. This could possibly be due to the set-up of the job run in the test. It were using 30 map and reduce tasks. The cluster had been configured to run one task per processor and thus all those 30 tasks could be run in parallel on the 30 processors system. This was not possible for the smaller cluster systems tested and this may be a reason these smaller systems needed more time to complete their jobs.

5.3 Algorithm Scalability

In this section, the scalability of the four algorithms tested and presented in figure 18 and figure 19 will be discussed in some depth. As a general remark, it can be said that the results depicted in the mentioned figures are similar in terms of scalability of the algorithms tested. However, the networks considered in figure 19, Erdõs-Rényi graphs, are mostly of a smaller size than those that was generated using the Random algorithm. This is due to the fact that the Erdõs-Rényi algorithm demands much more time and this time is proportional to the number of vertices squared.

Degree centrality is, as described in section 2.1.4, a rather simple algorithm to perform and it can be shown to have a time complexity of $\mathcal{O}(\text{n})$. In a parallel environment like Hadoop it will only need a map task to be calculated, as shown in appendix A. Thus, information sorting is not likely to have a great impact on total computational time. A linear behavior of the relation between number of vertices, n, and time of computations is expected. This is verified by the results presented in figure 18a.

The fact that smaller networks are used as input for the Erdõs-Rényi tests has an impact on the result shown in figure 19a. It can be seen that the figure is rather flat for graphs below 5 million nodes. This is related to the slow starting phase of the Hadoop system, as merely initiating a task may take several seconds. Thus, a fair conclusion is that the Hadoop framework is more suitable for dealing with sufficiently large and complex jobs.

Ego betweenness centrality is also a quite simple algorithm to carry out. The computational complexity is dependent on the number of vertices in the network linearly. Additionally, the number of edges m has an impact on the time of execution. As argued for in section 2.1.4, the anticipated behavior of computational time is $\mathcal{O}(\text{n+m})$. A property of social networks involving human beings is that the number of relations per person is not affected by the size of the network to a great extent, especially for larger networks. This applies to telecommunication networks. Hence it is fair to assume that m is linearly proportional to n and the algorithm has a time complexity of approximately $\mathcal{O}(\text{n})$. Both figure 18b and figure 19b seems to strengthen this point.

A possible issue in the context of MapReduce-like frameworks is that of uneven distribution of degree. For each vertex, the number of neighbors (n_n) will determine the time and space needed to calculate ego betweenness centrality, as a matrix of size $(n_n+1)\times(n_n+1)$ must be created and manipulated for each vertex. Thus, as multiplication with $n\times n$ matrices has an asymptotic lower bound time complexity of $\mathcal{O}(n^2)$ [43], an extreme distribution of degree centrality may cause an even more extreme distribution in workload between the tasks of the MapReduce job. These tasks will require much more time to finish and a slower implementation of the algorithm is likely. For relatively even distributions of degree a linear trend in computational time is to be expected. Figure 18b shows such a behavior for networks where degree has been limited to 25.

In the case of eigenvector centrality, the algorithm is more demanding compared to those mentioned above. However, as noted in 2.1.4, the time required to complete the computations is of the order $\mathcal{O}(n+m)$. The logical conclusion found in the case of ego betweenness centrality is applicable in the case as well, thus the time complexity might be approximated by $\mathcal{O}(n)$. This finding is strengthened by the results depicted in figure 18c.

The figure 18d shows a considerably different behavior than the other figures in figure 18. This can be attributed to the characteristics of the algorithm for calculating betweenness centrality. Brandes' algorithm, described in section 2.1.4, has a computational time that follows the behavior $\mathcal{O}(nm)$. In line with the arguments above regarding the correlation between n and m in telecom networks, it can be assumed that the time complexity can be rewritten as $\mathcal{O}(n^2)$. As mentioned in section 4.2.2, this is exactly the result gathered.

To summarize, the algorithms degree, ego betweenness and eigenvector centrality all manage to scale linearly. The performance is found to be much better for the two former algorithms however, they scale further and requires a lot less time to be executed on networks of comparable size. The reason for this is only partly the fact that the algorithms are simpler. Additionally, the algorithm for calculating the eigenvector centrality is of an iterative nature. In essence, this implies that several MapReduce jobs must be performed in order to obtain the final result. More specifically, one job per iteration is called for, where the output of the previous job will be input to the subsequent. In terms of the Hadoop framework, this involves storing the output of a job on the distributed file system and later reading and interpreting it as input for another job. The writing/reading procedure is quite time consuming and costly when it comes to hard drive space. As mentioned in section 2.2.1, this is one of the main drawbacks with the Hadoop framework. This issue will be a limiting factor for the the other three algorithms as well.

As a final remark it can be noted that the result for the tests on the Erdõs-Rényi graphs are following the same behavior as those for the Random graphs in the same network size range. This implies that it is reasonable to expect the results to be similar for larger network sizes as well.

5.4 Relations of Social Network Analysis Metrics in Telecommunication Network

Correlation tests have been made for every pair of the nine SNA metrics calculated on the graph created from the real CDRs described in section 3.1.2. Several interesting results of this correlation study are apparent in figure 22. Most notable is the immensely high correlation of 0.9995 between unweighted degree centrality and unweighted eigenvector centrality. Hence, these two metrics provides almost the same information regarding the subscribers in the telecommunication network. Other strong correlations is found between in- and out-degree centrality and their weighted counterparts, where values of 0.976 and 0.9603 have been calculated, respectively. Furthermore, weighted degree centrality is correlated to a high extent with all other metrics except ego betweenness centrality. Ego betweenness centrality is the one metric that seems to be the least related to the others, having its top correlation value of only 0.5696, with unweighted eigenvector centrality.

5.5 Prototype

Most of the steps defined in the prototype, as of section 3.3, can be performed in around an hour or less per step, according to figure 11. The notable exceptions are calculations of eigenvector centrality, both weighted and unweighted, which required more than 24 hours each to be completed. This seems to further indicate that iterative jobs generally require a lot of time in a Hadoop implementation.

This problem could be avoided by making use of the result discussed in section 5.4. The correlation table shown in figure 22 suggests an extremely strong correlation between unweighted degree centrality and unweighted eigenvector centrality at 0.9995. This implies that unweighted eigenvector centrality do not add more information regarding individual subscribers, compared to degree centrality. As the computational time of degree centrality is much lower, this is the preferred algorithm to implement in a prototype as that in figure 11. An other strong correlation is found between weighted eigenvector centrality and in-degree centrality. Thus, in-degree could be viewed as a good representation of weighted eigenvector centrality. All in all, both unweighted and weighted eigenvector centrality could be omitted from the implementation of the prototype. That would result in a total computational time of the could be reduced from several days to 6715 seconds, or about 1.87 hours.

5.6 Machine Learning Algorithms

The last part of the prototype, labeled the Influence Weight Estimation, has been studied separately. The results, presented in section 4.3.2, shows that the algorithms tested has about the same ability to classify the subscribers correctly. This measure is not always the most important, however. A more significant value for representing the accuracy of a decision model is the part of influential users that are actually found, defined as the recall in section 3.3.4. The VotedPerceptron is the algorithm managing to obtain the best recall, finding 56.8 % of all influential subscribers. VotedPerceptron is the only algorithm of a recall above 50 %. Recall might be a misleading measure of accuracy as well, due to the fact that simply classifying all subscribers as influential would ensure correctly classifying all influential subscribers. The F-value is a more precise way of favoring both finding influential subscribers and additionally avoiding discover influence where there is none to be found. Like the recall measure, VotedPerceptron did achieve the best F-value of 0.546. A more thorough description of the VotedPerceptron algorithm can be found in [44].

Although some accuracy is acquired, there are some improvements to be made. The probable main reason for the somewhat low accuracy is the lack of a clear relation between the attributes of the subscribers, the SNA metrics, and the classifying method used to create the training set, ranking with respect to betweenness centrality. This relation could be strengthened in two ways; either by generating the training set to resemble the attributes more or add more attributes, preferably such attributes that is correlated to the training set creation model. The training set used to test the prototype is not necessarily a good representation of influence and a different creation model could very well be used. For instance, an information spreading model, as in [45], could be used to determine the influence of a certain subscriber in a sub-graph of the network. As for adding more attributes, there is an ever increasing set of SNA metrics that aims at representing influence within the network. A few examples is PageRank [25], LineRank [46], Eccentricity Centrality [47] and Shapley Value [48].

An other thing to note is that the the short times to generate the decision models. This is partly due to the smallness of the training set, being less than 1 % of the complete social network. However, most algorithms for machine learning scales linearly or faster with respect to number of examples in the training set. Issues with accuracy may arise as the training set is so small and may not be a good representation of the entire graph. With the current betweenness centrality model for generating the training set, a larger training set increases the generation time drastically. A model based on some other concept of influence may more easily be used for creating larger training sets.

6 Conclusions and Further Work

This part of the thesis is a summary of the most important results of this study. In addition to this, implications and possible future studies is discussed.

Social network analysis has been around for about a century, but it is only lately that the trends has gone towards analysis of really large networks. With the increasing importance of the Internet and growing possibilities for people to contact each other, giant social networks tend to occur to a greater extent than before. To be able to analyze the networks in terms of SNA, novel technology for handling large data sets must be utilized. This study has examined the potential of using Hadoop, a framework for parallelization, to perform algorithms related to SNA. The scalability of various algorithms has been tested and a prototypical solution for finding vertices with wanted characteristics has been developed. Specifically, solutions handling telecommunication networks have been investigated.

In tests involving only a single machine with specifications described in 3.1.1, Hadoop showed promises of an ability to handle very large data sets and to scale computational time of algorithms in a linear fashion. Following that, more machines were added, forming a computer cluster for deployment of the Hadoop platform. Test yielded results indicating that for sufficiently large jobs, Hadoop manages to use much of the potential of every machine and obtain a speed increase proportional to the computational power increase. For full scale tests, a cluster consisting of 11 computers were used to try a number of SNA algorithms as well as other jobs. It turned out that algorithms that scales linearly in a non-distributed environment can be implemented in Hadoop with maintained scalability, in most cases. However, jobs involving many iterations or a large amount of information transmission between computers in the cluster tend to be processed less rapidly than jobs avoiding these traits.

A complete solution for analyzing telecommunication networks and detecting subscribers of interest have been developed. Starting from Call Detail Records generated for charging purposes, a social network is created. This network is analyzed to find an array of a total of nine different values for each subscriber, indicating that particular subscriber's influence within the network. A supervised learning method is ultimately used to determine a model for detection of relevant subscribers.

The prototype was tested but the execution was quite slow. A thorough analysis showed the two iterative SNA algorithm implemented, unweighted and weighted eigenvector centrality, was responsible for most of the processing time. Studying correlations between different SNA metrics reveals strong correlations between unweighted degree centrality and unweighted eigenvector centrality as well as between in-degree centrality and weighted eigenvector centrality. Thus, the both eigenvector centralities could very well be omitted without much loss of information. The time needed to perform all parts of the prototype, eigenvector centralities excluded, on a real telecom network of 2.4 million subscribers is just below 2 hours, a reasonable time.

The last part of the prototype, the supervised learning, has been studied in further detail. A small community, consisting of 20737 subscribers, of the network was chosen as a training and validation set. Accuracies of predictions of about 73 % were common and the algorithm called VotedPerceptron managed to find 56.8 % of the influential subscribers in the validation set. Moreover, VotedPerceptron obtained the best F-value of 0.546.

6.1 Further Research

Early in this study, Hadoop was compared to a graph database called Neo4j. For various reasons, Hadoop was chosen as the system to investigate further. However, a deeper study of the Neo4j application is sure to uncover interesting results. Especially, Neo4j should be tested in distributed mode on a computer cluster. An other system that were not tested at all in this work is Spark, which is a framework for parallelization that aims at dealing with iterative job in a more effective way than Hadoop. Testing that framework could be a relevant study as well. Another possible way of speeding up the calculation of the social network analysis metrics considered in this study is to implement a clever partitioning of the input data. This partitioning should ensure that vertices that are closely related ends up in the same map task. This could reduce the amount of data that needs to be sent between computers as well as improve the performance of any combiner in use. An investigation of this can be found in [34] for web graphs, but if an effective and computational partition function could be defined, this could be tried in the case of telecommunication networks as well.

This study has been limited to examination of merely four groups of centrality measures; degree, eigenvector, betweenness and ego betweenness. Extended tests on additional algorithms, for instance PageRank [25], LineRank [46], Eccentricity Centrality [47] and Shapley Value [48], may determine the scalability of these algorithms and provide information of their patterns in telecommunication networks. This might be helpful in terms of the machine learning algorithms as well, as it is more likely to find relations to the training set classification if a larger number of attributes is used. Either, more attributes could be given in the input examples in the training set of one machine learning algorithm, or the attributes could be split to form inputs to several models. In the latter case, the different models would ultimately be combined to yield an improved result.

To obtain more knowledge of telecommunication networks in general, further SNA studies should be performed on other real CDR sets. Especially, correlations between SNA metrics may be studied to learn more about the structure of telecom network. If a strong correlation between degree centrality and eigenvector centrality could be verified in other networks, this might be used as a rule of thumb in the future. That kind of information could prove very useful to telecom operators.

Lastly, when it comes to machine learning, algorithms should be tested using a larger training set. The one used in the test of this study was rather small, less than 1 % of the complete graph. This could possibly improve the accuracy of the algorithms. To be able to generate training sets of larger sizes, a different model for classifying the subscribers should be developed. Furthermore, unsupervised learning might be tested, for example k-mean clustering to see if there are any obvious groups within the network. If there is, then perhaps one could be identified as the influential users. For instance, a supervised learning part could be introduced, asking the operator to name a group of desirable influential users and the group finding most of them can be regarded as the influential group.

References

[1] Yongmin Choi, Hyun Wook Ji, Jae-yoon Park and Hyun-chul Kim, A 3W Network Strategy for Mobile Data Traffic Offloading. *Communications Magazine*, 118-123, 2011.

[2] Bob Emmerson, M2M: The Internet of 50 billion devices. *WinWin by Huawei*, 19-22, 2010.

[3] Christine Kiss, Andreas Scholz and Martin Bichler, Evaluating Centrality Measures in Large Call Graphs. *Proceedings of the 8th IEEE International Conference on E-Commerce Technology*, 2006.

[4] Muhammad Usman Khan and Shoab Ahmed Khan, Social Networks Identification and Analysis Using Call Detail Records. *Proceedings of the 2nd International Conference on Interaction Sciences: Information Technology, Culture and Human*, 192-196, 2009.

[5] Angela Bohn, Norbert Walchhofer, Patrick Mair and Kurt Hornik, Social Network Analysis of Weighted Telecommunications Graphs. *ePub Institutional Repository*, 2009.

[6] Eunice E. Santos, Long Pan, Dustin Arendt and Morgan Pittkin, An Effective Anytime Anywhere Parallel Approach for Centrality Measurements in Social Network Analysis. *2006 IEEE International Conference on Systems, Man and Cybernetics*, 2006.

[7] Oliver A. McBryan, Trends in Parallel Computing. *DTIC Online*, 1990.

[8] Jeffrey Dean and Sanjay Ghemawat, MapReduce: Simplified Data Processing on Large Clusters. *Proceedings of OSDI '04: 6th Symposium on Operating System Design and Implemention*, 2004.

[9] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski, Pregel: A System for Large-Scale Graph Processing. *PODC '09 Proceedings of the 28th ACM symposium on Principles of distributed computing* , 2009.

[10] Apache Hadoop, retrieved 15 November 2011, `http://hadoop.apache.org/`.

[11] Neo4j: NOSQL For the Enterprise, retrieved 15 November 2011, `http://neo4j.org/`.

[12] J. Scott, Social Network Analysis. *Sage*, 1988.

[13] L. C. Freeman, The Development of Social Network Analysis: A Study in the Sociology of Science. *Booksurge Llc*, 2004.

[14] C. Kadushin, Who Benefits from Network Analysis: ethics of social network research. *Social Networks*, 27:139-153, 2005.

[15] A. A. Nanavati, S. Gurumurthy, G. Das, D. Chakraborty, K. Dasgupta, S. Mukherjea and A. Joshi, On the Structural Properties of Massive Telecom Call Graphs: Findings and Implications. *Proceedings of the 15th ACM international conference on Information and knowledge management*, 435-444, 2006.

[16] M. E. J. Newman and Juyong Park, Why social networks are different from other types of networks. *Physical Review E*, 68:036122, 2003.

[17] Fredrik Hildorsson, Scalable Solutions for Social Network Analysis. *Diva portal*, 2009.

[18] Albert-László Barabási and Réka Albert, Emergence of Scaling in Random Networks. *Science*, 286:509-512, 1999.

[19] Jurij Leskovec , Deepayan Chakrabarti , Jon Kleinberg , Christos Faloutsos, Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. *in PKDD*, 133-145, 2005.

[20] Duncan J. Watts, Steven H. Strogatz, Collective dynamics of 'small-world' networks. *Nature*, 393:440-442, 1998.

[21] David Eppstein and Joseph Wang, A Steady State Model for Graph Power Laws. *International Workshop on Web Dynamics*, 2002.

[22] P. Erdõs and A. Rényi, On The Evolution of Random Graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences*, 1960.

[23] M. E. J. Newman, Analysis of Weighted Networks. *Physical Review*, 70, 056131, 2005.

[24] James H. Fowler, Who is the best connected congressperson? a study of legislative cosponsorship networks. *Paper presented at the American Political Science Association 2005 annual meeting*, 2005.

[25] Sergey Brin, Rajeev Motwani, Lawrence Page and Terry Winograd, What can you do with a Web in your Pocket?. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 21:37-47 1998.

[26] Ulrik Brandes, A Faster Algorithm for Betweenness Centrality. *Journal of Mathematical Sociology*, 25(2):163-177, 2001.

[27] Martin Everett and Stephen P. Borgatti, Ego network betweenness. *Social Networks*, 27:31-38, 2005.

[28] Santo Fortunato and Claudio Castellano, Community Structure in Graphs. *ArXiv e-prints*, 0712.2716, 2007.

[29] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte and Etienne Lefebvre, Fast unfolding of communities in large networks. *Journal of Statistical Mechanics*, P10008, 2008.

[30] Hadoop wiki, WordCount Example. retrieved 14 November 2011, `http://wiki.apache.org/hadoop/WordCount`.

[31] Course page by Jimmy Lin, Data-Intensive Information Processing Applications. retrieved 7 November 2011, `http://www.umiacs.umd.edu/~jimmylin/cloud-2010-Spring/syllabus.html`.

[32] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica, Spark: Cluster Computing with Working Sets. *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010.

[33] Sangwon Seo, Edward J. Yoon, HAMA: An Efficient Matrix Computation with the MapReduce Framework. *2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, 2010.

[34] Jimmy Lin and Michael Schatz, Design Patterns for Efficient Graph Algorithms in MapReduce. *MLG '10 Proceedings of the Eighth Workshop on Mining and Learning with Graphs* , 2010.

[35] Carola Lange, Harry M. Sneed and Andreas Winter, Comparing Graph-based Program Comprehension Tools to Relational Database-based Tools. *Proceedings of the 9 th International Workshop on Program Comprehension Tools*, 2001.

[36] Ian H. Witten and Eibe Frank, Data Mining: Practical Machine Learning Tools and Techniques. *Academic Press*, 2000.

[37] J. R. Quinlan, Induction of Decision Trees. *Machine Learning*, 81-106, 1986.

[38] J. R. Quinlan, C4.5: Programs for Machine Learning. *Morgan Kaufmann Publishers*, 1993.

[39] Tom M. Mitchell, Machine Learning. *McGraw-Hill*, 1997.

[40] Wikipidia article, Sigmoid Function. retrieved 3 October 2011, `http://en.wikipedia.org/wiki/Sigmoid_function`.

[41] David W. Hosmer, Stanley Lemeshow, Applied logistic regression. *John Wiley & Sons*, 2000.

[42] Simon Haykin, Neural Networks: A Comprehensive Foundation. *Prentice Hall PTR*, 1994

[43] Wikipidia article, Matrix Multiplication. retrieved 1 November 2011, `http://en.wikipedia.org/wiki/Matrix_multiplication`.

[44] Y. Freund, R. E. Schapire, Large margin classification using the perceptron algorithm. *11th Annual Conference on Computational Learning Theory, New York, NY,* 209-217, 1998.

[45] Maksim Kitsak, Lazaros K. Gallos, Shlomo Havlin, Fredrik Liljeros, Lev Muchnik, H. Eugene Stanley and Hernán A. Makse, Identification of influential spreaders in complex networks. *Nature Physics* 10:888-893, 2010.

[46] U Kang, Spiros Papadimitriou, Jimeng Sun, Hanghang Tong, Centralities in Large Networks: Algorithms and Observations. *SIAM / Omnipress* 119-130, 2011.

[47] Katarzyna Musiał, Przemysław Kazienko and Piotr Bródka, User Position Measures in Social Networks. *SNA-KDD '09 Proceedings of the 3rd Workshop on Social Network Mining and Analysis* 2009.

[48] Ramasuri Narayanam, Y. Narahari, A Shapley Value Based Approach to Discover Influential Nodes in Social Networks. *Nature Physics* 8:130-147, 2011.

[49] U Kang, Charalampos E. Tsourakakis and Christos Faloutsos, PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations. *Proceedings of the 2009 Ninth IEEE International Conference on Data Mining (ICDM 2009)*, 229-238, 2009.

A Parallelization of Social Network Algorithms

One of the greatest challenges of analyzing graphs by applying parallel computing is to make the algorithms work in parallel. The fact that many analytic tools needs access to the whole graph causes problems as information regarding the graph structure will be distributed on the different nodes of the computer cluster. For that reason, many graph analysis algorithm involves sending important information between the nodes and in the end gathering all relevant and necessary data in one place. Specifically, the map phase generally consists of a vertex sending some sort of information to its neighbors. In the reduce phase, the information from every neighbor is gathered by a vertex and processed into the output of the iteration.

This usually implies using an iterative algorithm, which may result in increased time complexity. Specific algorithms for some of the metrics introduced earlier, namely degree centrality, eigenvector centrality, ego betweenness centrality and betweenness centrality, is described in further detail. The solutions that is explained all assumes that the graph is available as a adjacency list. The list can be in the form of an ordinary text file where each row in the list contains the identification number, henceforth referred to as id, of the vertex and an array of the vertex outgoing neighboring vertices, both their id and the weight of the relation. This is the general case as such an adjacency list always can be constructed provided the graph structure is known.

A.1 Degree Centrality

The graphs considered in the real user case of a telecommunication network are directed, as implied by the form of the adjacency list. The array on each row of the adjacency list contains information regarding incoming edges of the vertex. This is sufficient to calculate out-degree centrality. Thus, this calculation is performed with only a map step on a MapReduce-like framework, such as Hadoop. The map method simply counts how many neighboring vertices that are listed in the array corresponding to a vertex. The weights could also be included to obtain a weighted out-degree centrality by simply summing the weights in the array.

To calculate the in-degree implies a slight increase in complexity. In order to do the computation properly information has to be sent form the destination vertex to the originating vertex of each edge. This is done in the map step. A message is sent, containing the destination vertex id and the weight of the edge. In the Hadoop environment, this means setting the key to be the originating vertex of an edge, and the id of the destination vertex and the weight of the edge as the value. Each vertex do this process for all its incoming edges. When Hadoop sorts the output of the map part by key all messages to a specific vertex will be collected in the same reduce process. Thus, the reduce step will simply be an aggregation of the messages sent from the map phase, either by number for regular in-degree centrality or by weight for the weighted form of the metric.

Below is a list showing the steps of the algorithm in the unweighted case of out-degree centrality.

```
1. class MAPPER
2.   method MAP(id n, node N)
3.     DC ← |N.ADJACENCYLIST|
4.     COLLECT(n, DC)
```

A.2 Eigenvector Centrality

Eigenvector centrality is given as the eigenvector of the adjacency matrix corresponding to its largest eigenvalue. As has been mentioned earlier, this particular eigenvector can be found through the iterative power method, as stated in equation (3). As the networks considered are all very large, they yield very large adjacency matrices. However, these matrices all tend to be very sparse with each row containing only a few non-zero elements, usually less than a hundred. Thus, ignoring the zeros could save a lot of time. A way of performing a matrix-vector multiplication using the Hadoop framework can be found in [49]. This approach involves sending data regarding edges and their weights, as well as the current value of a vertex eigenvector centrality. This is repeated iteratively until all the elements of the eigenvector is changed between iterations by less than a specific threshold. Each iteration is divided into two MapReduce jobs, which will be described in greater detail.

As a pre-processing step to the algorithm each vertex has to be assigned a starting eigenvector centrality. This is usually set to one. Also, if not already done, the weights of the edges of each vertex have to be normalized. This can for instance be done with respect to the total weight of the outgoing edges of each vertex. The normalization has to be done in order to maintain the value of the 1-norm of the eigenvector, which influences the stopping condition.

After the pre-processing the actual algorithm takes place. The map step is simply sending messages with information to the appropriate vertices. For each vertex a single message is sent containing the current eigenvector centrality of that vertex. This message is sent to the vertex itself. Furthermore, for each outgoing edge a message is sent to the originating vertex, specifying the id of the destination vertex and the product of the eigenvector centrality and the weight of the edge. When the reducer receives a bundle of messages it first sorts them depending on if they hold facts concerning the current value of the eigenvector centrality or the incoming edges of the vertex. Following that, all products sent in the map tasks are summed and the result is the new eigenvector centrality.

For stopping condition purposes, this is compared with the old value in order to determine if the change is significant or if the value seems to converge. Finally, information about the new eigenvector centrality and the edges of a vertex is put together to resemble the input of the first job. Thus, the output of this step can be used as input in the next iteration of the algorithm.

```
1. class MAPPER
2.  method MAP(id n, node N)
3.   EC ← N.EC
4.   COLLECT(n, EC)
5.   for all Node Neighbor ∈ N.ADJACENCYLIST
6.    w ← N.ADJACENCYLIST.weight(Neighbor)
7.    COLLECT(Neighbor.ID,[n,w])
```

```
1. class REDUCER
2.  method REDUCE(id n, [$w_1$,$w_2$,...,$w_n$])
3.   for all $w_i$ ∈ values
4.    if IsEC($w_i$)
5.     EC ← $w_i$
6.    else
7.     $weight_i$ ← $w_i$.Weight
8.     $Neighbor_i$ ← $w_i$.Id
9.   for all i ∈ [1,...,|$weight_i$|]
10.   newPartEC ← $weight_i$×EC
11.   COLLECT($Neighbor_i$,[n,$weight_i$,newPartEC])
```

A.3 Ego Betweenness Centrality

In order to calculate ego betweenness centrality for a whole graph, the ego network needs to be constructed for each vertex. In addition to a list of neighbors, which is readily available from the adjacency list, information concerning the neighbors of all neighbors must be accessible. With that data, the adjacency matrix of the ego network can be constructed and the ego betweenness centrality can quickly be calculated with it [27].

To accomplish this in the parallel environment of Hadoop, a single MapReduce job is needed. First, during the map step, each vertex sends a message to each of its neighbors. The message contains a list of all the sending vertex neighbors. In the reducer, a vertex receives neighbor lists from every neighbor. By comparing these and determining which of the neighbors that knows each other, the adjacency matrix of the ego network can be constructed. Implementing the simple matrix multiplication and element aggregation described in [27] ultimately gives the value of the ego betweenness centrality.

The algorithm can be represented in list form in the following way:

```
1. class MAPPER
2.  method MAP(id n, node N)
3.   list $\leftarrow$ N.ADJACENCYLIST
4.   for all Node Neighbor $\in$ N.ADJACENCYLIST
5.    nId $\leftarrow$ Neighbor.ID
6.    COLLECT(nId,[n,list])
```

```
1. class REDUCER
2.  method REDUCE(id n, [$l_1$,$l_2$,...,$l_n$])
3.   for all $l_i \in$ values
4.    $id_i \leftarrow l_i$.n
5.    $list_i \leftarrow l_i$.list
6.   M $\leftarrow$ CreateEgoAdjacencyMatrix(id,list)
7.   EBC $\leftarrow$ CalculateEgoBC(M)
8.   COLLECT(n,EBC)
```

A.4 Betweenness Centrality

As has been mentioned before, ego betweenness centrality is just an approximation of the metric betweenness centrality. Whereas a calculation of ego betweenness centrality merely requires information about nearest neighbors, computation of exact betweenness centrality needs the complete network. These computations can be very time consuming for large graphs, although Brandes has proposed a faster algorithm [26]. To perform this algorithm, information has to be sent in several steps, making the algorithm iterative. The algorithm starts with a source vertex sending a message to each of its neighbors, containing the original source of the message, the distance from the sender to that source, number of shortest paths between the source and the sender and a list of predecessors on these shortest paths. In each step of the iteration this message is past on until it has reach every vertex in the network.

In terms of the Hadoop framework, most of the algorithm takes place in the reduce phase, whereas the mapper simply passes on the message received from the reducer. More specifically, the number of shortest path from each source vertex is calculated in the reduce phase. Additionally, new messages are sent to neighbors if messages originating from a specific source have not been received earlier.

At this point a process of calculating the contribution of betweenness centrality from vertices to every other vertex is initiated. This is done by starting with the vertices furthest away and moving towards the source. At each step the contribution of betweenness centrality, by Brandes called dependency, to all predecessors of the current vertex is updated according to the formula

$$\delta(v) = \delta(v) + \frac{\sigma(v)}{\sigma(w)}(1 + \delta(w)) \tag{20}$$

where v is the predecessor vertex, w is the vertex considered at the moment, $\delta(x)$ is the dependency or contribution to betweenness centrality of vertex x and $\sigma(x)$ is the number of shortest path from the source to vertex x.

As a final step, the contributions are summed to yield the total betweenness centrality score for each vertex. The complete procedure is presented in a more concise form in the table below. The input key, called message, contains the information described above.

Forward steps:

```
1. class MAPPER
```

```
 method MAP(id n, message M)
  COLLECT(n, M)
```

```
class REDUCER
 method REDUCE(id n, message M)
  Map<id,message> mmap
  for all $M_i \in$ values
   if ISNODE($M_i$)
    COLLECT(n,$M_i$)
    neighbors $\leftarrow$ $M_i$.NEIGHBORS
   else
    if mmap.CONTAINS($M_i$.SOURCE)
     if $M_i$.DISTANCE < mmap( $M_i$).DISTANCE
      mmap.REPLACE($M_i$)
     else if $M_i$.DISTANCE = mmap( $M_i$).DISTANCE
      mmap($M_i$).ADDSIGMA()
      mmap($M_i$).ADDPREDECESSOR()
    else
     mmap.ADD($M_i$)
  for all $M_i \in$ mmap
   if key.NOTVISITED($M_i$.SOURCE)
    for all id neigh $\in$ neighbors
     COLLECT(neigh,$M_i$)
```

Backward step:

```
class MAPPER
 method MAP(id n, message M)
  COLLECT(n, M)
```

```
class REDUCER
  method REDUCE(id n, message M)
    Map<id,message> mmap
    SortedMap<id,dist> smap
    for all $M_i \in$ values
      mmap.ADD($M_i$)
      smap.ADD($M_i$.DISTANCE)
    for all $M_i \in$ DECENDINGKEY(smap)
      for all predecessor $\in$ mmap($M_i$).PREDECESSOR
        predecessor.UPDATEDEPENDECY($M_i$)
      COLLECT($M_i$,$M_i$.DEPENDENCY)
```

B Technical Specifications

The appendix gives the technical specifications of the computer cluster used for running Hadoop jobs in detail.

B.1 Master

B.1.1 Memory

MemTotal: 49552620 kB
MemFree: 137976 kB
Buffers: 653984 kB
Cached: 45043532 kB
SwapCached: 3476 kB
Active: 2456940 kB
Inactive: 45109760 kB
Active(anon): 1486008 kB
Inactive(anon): 383332 kB
Active(file): 970932 kB
Inactive(file): 44726428 kB
Unevictable: 0 kB
Mlocked: 0 kB
SwapTotal: 79068152 kB
SwapFree: 78985744 kB
Dirty: 108 kB
Writeback: 0 kB
AnonPages: 1867048 kB
Mapped: 19708 kB
Shmem: 72 kB
Slab: 813908 kB
SReclaimable: 772240 kB
SUnreclaim: 41668 kB
KernelStack: 5440 kB
PageTables: 10748 kB
NFS_Unstable: 0 kB
Bounce: 0 kB
WritebackTmp: 0 kB
CommitLimit: 103844460 kB
Committed_AS: 4174320 kB
VmallocTotal: 34359738367 kB
VmallocUsed: 221772 kB
VmallocChunk: 34333852988 kB
HardwareCorrupted: 0 kB

HugePages_Total: 0
HugePages_Free: 0
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 2048 kB
DirectMap4k: 8432 kB
DirectMap2M: 50313216 kB

B.1.2 CPU

The master has 16 identical processors with the following specifications:

processor : 0
vendor_id : GenuineIntel
cpu family : 6
model : 26
model name : Intel(R) Xeon(R) CPU X5550 @ 2.67GHz
stepping : 5
cpu MHz : 2667.163
cache size : 8192 KB
physical id : 0
siblings : 8
core id : 0
cpu cores : 4
apicid : 0
initial apicid : 0
fpu : yes
fpu_exception : yes
cpuid level : 11
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts rep_good xtopology nonstop_tsc aperfmperf pni dtes64 monitor ds_cpl vmx est tm2 ssse3 cx16 xtpr pdcm dca sse4_1 sse4_2 popcnt lahf_lm ida tpr_shadow vnmi flexpriority ept vpid
bogomips : 5334.32
clflush size : 64
cache_alignment : 64
address sizes : 40 bits physical, 48 bits virtual
power management:

B.1.3 Hard Drive

Filesystem	Size	Used	Avail	Use %	Mounted on
/dev/cciss/c0d0p1	201G	34G	158G	18 %	/
none	24G	220K	24G	1 %	/dev
none	24G	0	24G	0 %	/dev/shm
none	24G	76K	24G	1 %	/var/run
none	24G	0	24G	0 %	/var/lock
none	24G	0	24G	0 %	/lib/init/rw
none	201G	34G	158G	18%	/var/lib/ureadahead/debugfs
/dev/cciss/c0d1p1	1.4T	1.2T	155G	89%	/disk1

B.2 Slave

B.2.1 Memory

MemTotal: 2056872 kB
MemFree: 36192 kB
Buffers: 78484 kB
Cached: 1160780 kB
SwapCached: 4700 kB
Active: 535788 kB
Inactive: 1278264 kB
Active(anon): 412164 kB
Inactive(anon): 162636 kB
Active(file): 123624 kB
Inactive(file): 1115628 kB
Unevictable: 0 kB
Mlocked: 0 kB
SwapTotal: 6025208 kB
SwapFree: 5964948 kB
Dirty: 116 kB
Writeback: 0 kB
AnonPages: 573900 kB
Mapped: 12660 kB
Shmem: 12 kB
Slab: 148108 kB
SReclaimable: 134652 kB
SUnreclaim: 13456 kB
KernelStack: 1552 kB
PageTables: 3724 kB
NFS_Unstable: 0 kB
Bounce: 0 kB
WritebackTmp: 0 kB

```
CommitLimit: 7053644 kB
Committed_AS: 1703480 kB
VmallocTotal: 34359738367 kB
VmallocUsed: 276524 kB
VmallocChunk: 34359455932 kB
HardwareCorrupted: 0 kB
HugePages_Total: 0
HugePages_Free: 0
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 2048 kB
DirectMap4k: 8032 kB
DirectMap2M: 2088960 kB
```

B.2.2 CPU

The slaves each have 2 processor with the following specifications:

```
processor : 0
vendor_id : GenuineIntel
cpu family : 15
model : 6
model name : Intel(R) Pentium(R) D CPU 3.00GHz
stepping : 4
cpu MHz : 2400.000
cache size : 2048 KB
physical id : 0
siblings : 2
core id : 0
cpu cores : 2
apicid : 0
initial apicid : 0
fpu : yes
fpu_exception : yes
cpuid level : 6
wp : yes
flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush
dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall lm constant_tsc pebs bts pni dtes64 monitor
ds_cpl vmx est cid cx16 xtpr pdcm lahf_lm tpr_shadow
bogomips : 5985.07
clflush size : 64
cache_alignment : 128
address sizes : 36 bits physical, 48 bits virtual
```

power management:

B.2.3 Hard Drive

Filesystem	Size	Used	Avail	Use %	Mounted on
/dev/sda1	87G	4.2G	78G	6 %	/
none	1001M	216K	1000M	1 %	/dev
none	1005M	0	1005M	0 %	/dev/shm
none	1005M	44K	1005M	1 %	/var/run
none	1005M	0	1005M	0 %	/var/lock
none	1005M	0	1005M	0 %	/lib/init/rw
/dev/sdb1	459G	263G	173G	61 %	/disk2
/dev/sda3	367G	269G	80G	78 %	/disk1