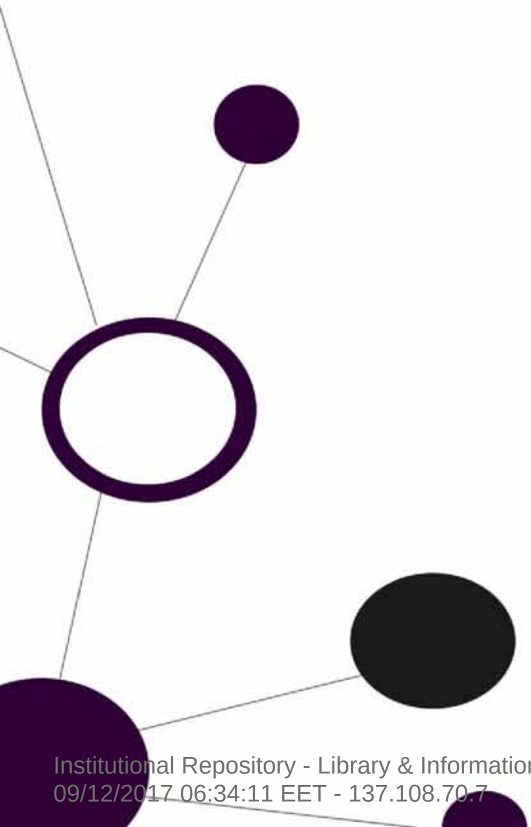
MapReduce-based Distributed *K*-shell Decomposition for Online Social Networks

Katerina S. Pechlivanidou
Dept. Electrical and Computer Engineering
University of Thessaly
Volos, Greece
June 2014
katpechliv@gmail.com



Advisor: Dimitrios Katsaros, Lecturer
co-Advisors: Antonios Argyriou, Lecturer
Athanasios Korakis, Lecturer
Dept. Electrical and Computer Engineering
University of Thessaly

Αποσύνθεση Online Κοινωνικών Δικτύων σε *k*-κελύφη βασισμένη στο προγραμματιστικό μοντέλο MapReduce

Κατερίνα Σ. Πεχλιβανίδου
Τμ. Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών
Πανεπιστήμιο Θεσσαλίας
Βόλος, 2014

Επιβλέπων: Δημήτριος Κατσαρός
Συνεπιβλέποντες: Αντώνιος Αργυρίου
Αθανάσιος Κοράκης

To my Family

Words of Thanks

My special gratitude goes to my supervisor Mr. Katsaros Dimitrios who provided me guidance throughout this work. I specially thank him for his confidence in my work and his willingness to help me anytime.

I wish to thank the members of CERTH (CEnter for Research and Technology, Hellas) for their overall support. Many thanks to the Technical Support team of the Department of Electrical and Computer Engineering of the University of Thessaly for their kind cooperation.

I heartily thank all of my dear friends for their continuous support during every academic and personal challenge.

I especially thank my family for their unconditional support, motivation and encouragement. Thanks to them I had the opportunity complete this Thesis.

Katerina S. Pechlivanidou

Department of Electrical and Computer Engineering
University of Thessaly
Volos, Greece
June, 2014

Part of this work is presented in the IEEE 10th World Congress on Services in proceedings of the 2nd International Workshop on Personalized Web Tasking (PWT) 2014 under the title: "MapReduce based Distributed K-shell Decomposition for Online Social Networks", authors Katerina Pechlivanidou, Dimitrios Katsaros, Leandros Tassiulas[1].

Περίληψη

Η ανάλυση των κοινωνικών δικτύων περιλαμβάνει μια ομάδα εργαλείων για την ανάλυση μεγάλων online κοινωνικών δικτύων. Ανάμεσα σε αυτά τα εργαλεία, η αποσύνθεση σε k-κελύφη ενός γραφήματος είναι μια τεχνική που έχει χρησιμοποιηθεί για την ανάλυση κεντρικότητας των κόμβων (centrality analysis), για την ανακάλυψη κοινοτήτων (communities discovery), για τον εντοπισμό κόμβων σημαίνουσας επιρροής, κλπ. Παρόλο που ο υπολογισμός της είναι σχετικά απλός, ο μεγάλος όγκος των γραφημάτων εισόδου και τα περιβάλλοντα στα οποία ο αλγόριθμος πρέπει να εκτελεστεί, δηλαδή στα μεγάλα κέντρα δεδομένων (datacenters) των κολοσσών του Διαδικτύου, καθιστούν όλους τους ήδη υπάρχοντες αλγορίθμους για την αποσύνθεση των δικτύων σε k-κελύφη ακατάλληλους. Σε αυτή την εργασία, αναπτύσσουμε για πρώτη φορά στη βιβλιογραφία έναν κατανεμημένο αλγόριθμο βασισμένο στο MapReduce για την αποσύνθεση ενός δικτύου σε k-κελύφη. Επιπλέον, παρέχουμε μια εφαρμογή και αξιολόγηση του αλγορίθμου χρησιμοποιώντας πραγματικά δεδομένα κοινωνικών δικτύων. Αναλύουμε τους συμβιβασμούς και την επιτάχυνση του προτεινόμενου αλγορίθμου και συνοψίζουμε τις αρετές και τις αδυναμίες του.

Τμήμα της παρούσας εργασίας παρουσιάζεται στην παρακάτω εργασία συνεδρίου: K. Pechlivanidou, D. Katsaros, L. Tassiulas, "MapReduce-based Distributed K-shell Decomposition for Online Social Networks", In Proceedings of the 2nd International Workshop on Personalized Web Tasking (PWT), In the context of the IEEE 10th World Congress on Services, Anchorage, Alaska, USA, June 27 – July 2, 2014[1].

Abstract

Social network analysis comprises a popular set of tools for the analysis of large online social networks. Among these tools, k-shell decomposition of a graph is a technique that has been used for centrality analysis of nodes, for communities discovery, for the detection of influential spreaders, and so on. Even though its computation is relatively simple, the huge volume of input graphs and the environments where the algorithm needs to run, i.e., large datacenters of Internet giants, makes none of the existing algorithms appropriate for the decomposition of graphs into shells. In this article, we develop for the first time in the literature, a distributed algorithm based on MapReduce for the k-shell decomposition of a graph. We furthermore, provide an implementation and assessment of the algorithm using real social network datasets. We analyze the tradeoffs and speedup of the proposed algorithm and conclude for its virtues and shortcomings.

Contents

Figure List

List of Algorithms

List of Tables

Chapter 1

Introduction

The tremendous advances in information technology combined with the omnipresent connectivity via wired or wireless networks have created a huge development and popularity of Online Social Networks (OSN). Facebook, Twitter, Instagram and LinkedIn are some of the most popular OSNs nowadays. The number of registered Facebook users recently exceeded one billion[1] and Twitter celebrated its seventh birthday with more than 500 million[2] users. Despite its short life, SnapChat's number of registered users is estimated to be several tens of millions. All such online social networks store and process tremendous amount of data mostly in form of pair wise interactions. Those pair wise information basically refer to a person's interactions and thus form networks (i.e., graphs). A social network is a theoretical construct that is useful in order to study almost any relationship such as the relationships between individuals, groups and organizations, and sometimes between entire societies [32].

The operational and business advantages of analysis and mining of these graphs are significant to the OSN owner. For example, the discovery and/or prediction of complex associations among interacting entities allow more efficient handling (storing, query processing) of the generated data. Also, the discovery of romantic relationships among persons in an OSN [6], permits the OSN owner to design marketing policies (e.g., product recommendations which generate revenue for the OSN).

The term "Social Network Analysis" (SNA) refers to the process and analysis of the properties of a social network and has its roots in the late '60s. The famous Milgram experiment, by Yale University psychologist Stanley Milgram, which quantified the "degree of separation", has been the cradle of SNA. In SNA, social relations are treated as units of network theory, meaning that individuals are represented as nodes and their relationship as edges. Since Milgram, the initial set of tools and algorithms for SNA has been extended so much that apart from the basic centrality metrics (like degree, closeness and shortest path betweenness) we have now spectral centrality and flow betweenness centrality measures, intelligent community discovery techniques based on modularity or other graph-theoretic concepts, algorithms for detecting influential spreaders or maximizing the spread of influence in a social network, methodologies for network sparsification to reduce its size, and so on.

Among all encountered concepts in SNA, *k*-shell decomposition of a graph seems to be appealing since it highlights the internal core structure of a network graph and reveals its hidden hierarchies. *K*-shell decomposition is a simple algorithm, originating from statistical mechanics for investigating graph properties, and in particular for discovering cohesive subgroups within a network. *K*-shell has been used in many intelligent and diverse concepts varying from detecting influential spreaders [6] and discovering communities [3] to Internet's structure analysis at the autonomous level [4], visualization purposes, and so on.

[1]http://news.yahoo.com/number-active-users-facebook-over-230449748.html

[2]http://www.telegraph.co.uk/technology/twitter/9945505/Twitter-in-numbers.html

Due to the significance of *k*-shell as a measure in SNA, many algorithms have been proposed for the computation of the *k*-shells in several different computational environments. More specifically, these environments range from single machine's main memory [5] and secondary storage [6] to small clusters that consist of few nodes [7]. *K*-shell decomposition algorithms have also been proposed for network graphs that are unweighted or weighted [8] and for static networks whose topology changes over time and is acquired in a streaming mode [9]. The most encountered and appealing in the majority of today's online social network algorithms are those referring to static or almost static network graphs, meaning those whose topology is changing very slowly compared to the time required to run analytics over the internet. Thus, we focus on networks that are static and unweighted and we propose an efficient implementation of *k*-shell decomposition of such network graphs.

1.1 Motivation

In this subsection the details of this work's motivation are presented. As mentioned earlier, modern online social networks consist of several millions of nodes and therefore any of the existing centralized *k*-shell decomposition algorithms that rely on single machine, exploiting solely the machine's main memory , is doomed to fail eventually due to lack of computational resources.

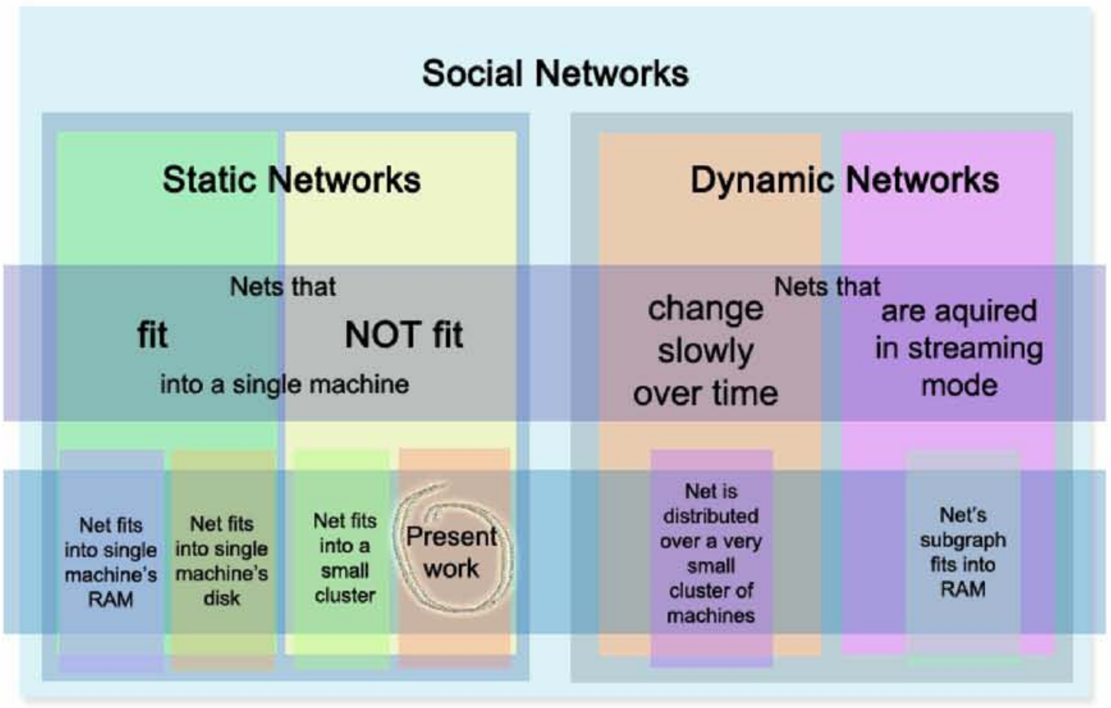

Social Networks
Static Networks
Dynamic Networks
Nets that
fit
NOT fit
into a single machine
Nets that
change slowly over time
are aquired in streaming mode
Net fits into single machine's RAM
Net fits into single machine's disk
Net fits into a small cluster
Present work
Net is distributed over a very small cluster of machines
Net's subgraph fits into RAM

Figure 1 Deficiencies of the existing approaches

Distributed solutions have also been proposed in order to overcome the aforementioned limitations. However, the computation of the *k*-shell decomposition that is implemented in a straightforward way includes a highly sequential process, deleting one node after another along with the incident edges. The latter indicates that developing a distributed version of the initially proposed *k*-shell decomposition algorithm may be a challenging task. Some of the solutions that are presented in [10] and [7] are well initiatives, but are still insufficient since they are able to run only on a small number of clustered machines. Considering the latter shortcoming, it is obvious that it is impossible for those algorithms to run on clusters of modern infrastructures which are maintained by Internet giants such as Google, LinkedIn and Facebook who operate huge datacenters and several thousands of machines. Clusters like these, are usually programmed by a high-performance middleware of MapReduce type [11]. Therefore, the *k*-shell decomposition of a network graph in a MapReduce format is necessary. The present work is exactly filling this gap. Figure 1 presents the categorization of the existing works that address the problem of *k*-shell decomposition of a graph highlighting the work's position in this ecosystem.

1.2 Contributions

Summarizing the above sections, the present work falls into the area of social network analysis and is dealing with the computation of the *k*-shell decomposition of a given network graph. The contributions that this work makes are the following:

- It develops a distributed algorithm for the computation of the *k*-shells of a given network graph.
- For the first time in the literature it presents a parallel and distributed algorithm for the *k*-shell decomposition based on Hadoop's MapReduce programming paradigm. It is therefore suitable for huge datacenter environments owned by modern Internet giants.
- It assesses the performance of the proposed algorithm proven with experimental results. For this purpose, real datasets where used. This work also analyzes various tradeoffs in its operation.

The rest of this work is organized as follows: In chapter 2 the related work is described; chapter 3 describes the necessary background and provides the description of the proposed algorithm along with an example of a network decomposed into its shells. In chapter 4, we present evaluation of the algorithm and finally chapter 5 concludes this work.

Chapter 2

Related Work

The present work tackles two very large areas: The first one is the data processing area that is based on MapReduce's programming model and the second one is the area of social/complex network analysis, especially referring to the *k*-core decomposition of a graph. It is imperative to briefly name some important works of the aforementioned areas and then analyze those who are the most relevant to the *k*-core decomposition of a network.

The MapReduce framework, since its debut in [11], along with its open source implementation in Hadoop[3], have been widely used in many fields concerning mostly huge or colossal data processing. MapReduce has contributed fundamental ideas in Information Retrieval and has been used for inverted index creation and Web page ranking [12]. It has also been used for document similarity discovery [13], for max-cover calculation [4], for data mining in [14] and [15], recently for bioinformatics data processing [16] and so on. Other directions of MapReduce enhancements include ways to extend it with database capabilities [17], to improve parallelization [18], energy consumption [19], and many others. Some of the major enhancements suggested to Hadoop that are described in [20] are related to data storage, processing and placement.

The roots of the literature focusing on the analysis of social networks are quite old [21] and very rich. Hundreds of articles have been in the light of publication during the last decade and several new topics, besides classic problems concerning the calculation of various centrality measures or detection of communities in social networks, have emerged. Such topics are falling into areas like databases, ad hoc networking [22], detection of influential nodes, evolution of social networks [23], and so on.

One of the major enhancements is that of detecting the most "central" or "influential" nodes in a social network. Among other more classic centrality measures, like degree, closeness and shortest-path betweenness, *k*-shell is appealing. *K*-shell was first proposed in [24] and afterwards it attracted the attention of data and network scientists. Some of the most successful application areas of *k*-shell are the Internet Topology modeling [25], the detection of influential spreaders [6] and the discovery of communities [3].

In a straightforward implementation of the *k*-core decomposition algorithm, we need to perform recursive deletions of all vertices and edges incident with them. Some efficient versions of the initial algorithm exist for various settings involving the type of network and the available computational resources. There are two main categories of algorithms; first there are the ones dealing with dynamic graphs (slowly or fast changing) and the other group of algorithms handles static networks (known in advance and not changing). The literature on *k*-shell decomposition for dynamic graphs is very limited and is able to handle only slowly changing graphs with the type of distributed algorithms described in [10] and those whose topology is acquired in a streaming mode [9].

[3] http://hadoop.apache.org

When dealing with static networks that fit entirely into the main memory, then the *k*-core decomposition is performed in O(number_of_edges) due to [5]. When moving to larger graphs that cannot be stored in the main memory but need additionally a secondary memory other proposed solutions like the one described in [6] can be used. For progressively larger network graphs that do not fit in the memory of a single machine the exploitation of a very small cluster can be used to decompose the network to its shells [6]. Figure 1 depicts both the most relevant work and the position of the present work in this ecosystem.

It is oblivious that none of the aforementioned solutions is suitable for the type of infrastructures that Internet giants like Google, Facebook, Yahoo!, LinkedIn and Twitter use. These internet companies own huge clusters that comprise of thousands of commodity machines and are usually programmed by MapReduce frameworks; therefore the need for a new solution has emerged. Hence, this work focuses on filling this gap.

Chapter 3

The Proposed Distributed Algorithm

In the following subsections we describe the necessary background, we provide the proposed algorithm for the *k*-shell decomposition suitable for enormously large networks and finally we discuss the obtained results from the experimental evaluation.

3.1 Background

In this section the background of MapReduce, the *k*-shell decomposition and all relevant areas are presented in details. For those who are not familiar with cloud computing, reading of subsections 3.1.2 and 3.1.3 are strongly recommended.

3.1.1 Definition of k-shell Decomposition

We begin describing a procedure for the decomposition of a graph into shells. The *k*-shell decomposition of a network graph is performed iteratively. The initial round of the decomposition involves removing all nodes that have degree equal to one, along with the unique incident link, and indexing these as *k*=1. As a next step we consider all nodes of the resulting graph of degree 1 also to have *k*=1 and are again pruned. The process is repeated until there are no nodes of degree 1. The same procedure is applied in next pruning rounds and therefore all nodes with i or fewer connections are iteratively removed. The group of the letter nodes is indexed as *k*=i. The final output of the *k*-shell decomposition is a single number for each node: its core assignment. Generally, if from a given graph we recursively delete all

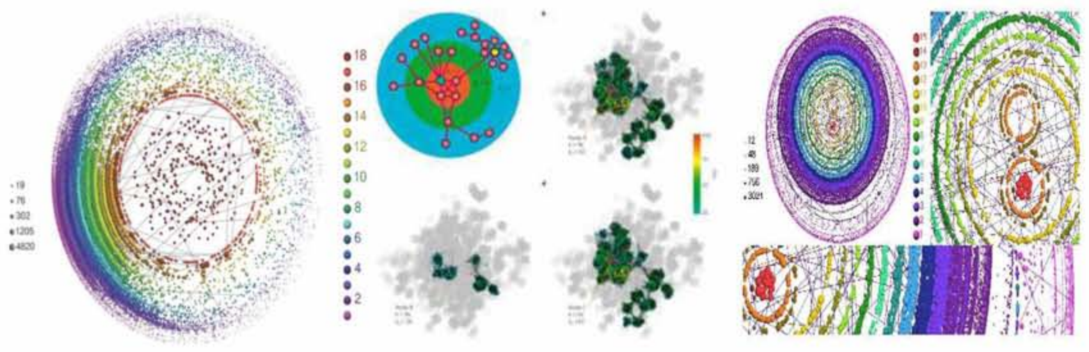


Figure 2 (Left) Visualisation of the social network of active users, based on k-core decomposition and components analysis [33], (Middle) shows the case when hub nodes of a network may not be good influential spreaders [34], (Right) depicts an application of k-shell for France's Internet domain [35]

vertices, and lines incident with them, of degree less than k, the remaining graph is the k-core[4].

To grasp the concept of k-shell decomposition let us decompose the sample graph shown in Figure 3a. Initially, k is set to 1. We now delete all nodes of degree 1; those can be seen in Figure 3b. As a next step we proceed with deleting all nodes arising with degree equal to 1 in the remaining graph (Figure 3c); the same is followed in case of Figure 3d until no node with degree 1 remains (Figure 3e). The next pruning round requires that k is increased. Therefore, k=2 in the next round (Figure 3e); k is then again increased (k=3) and all nodes with degree 3 are deleted. We can see that the maximum shell is 3 (3-coreness). The final result is presented in Figure 4.

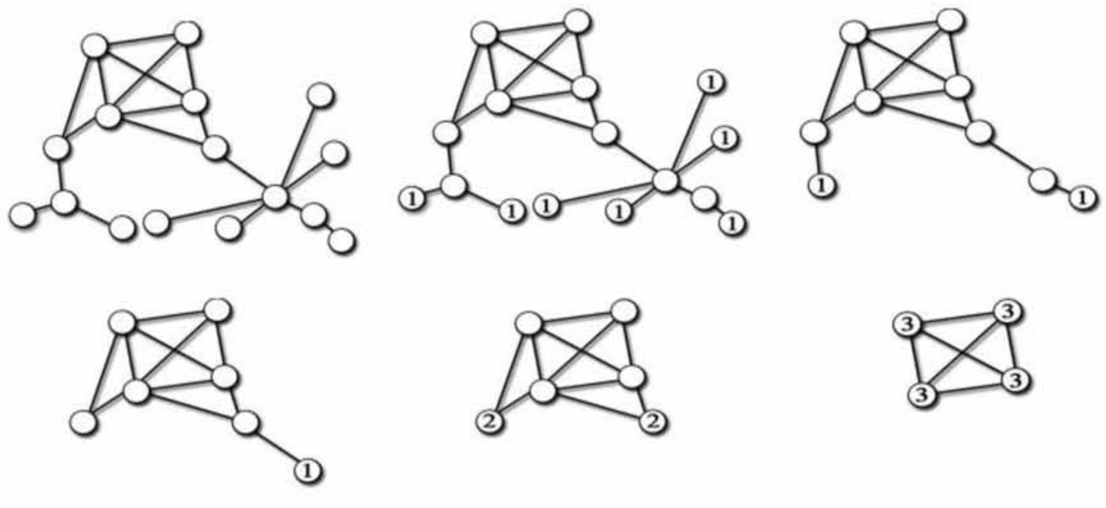


Figure 3 k-shell decomposition of a sample graph (from Top Left to Bottom Right): a) the sample graph, b) k=1, c) k=1, d) k=1 e) k=2, f) k=3

[4]We use the terms k-core and k-shell interchangeably.

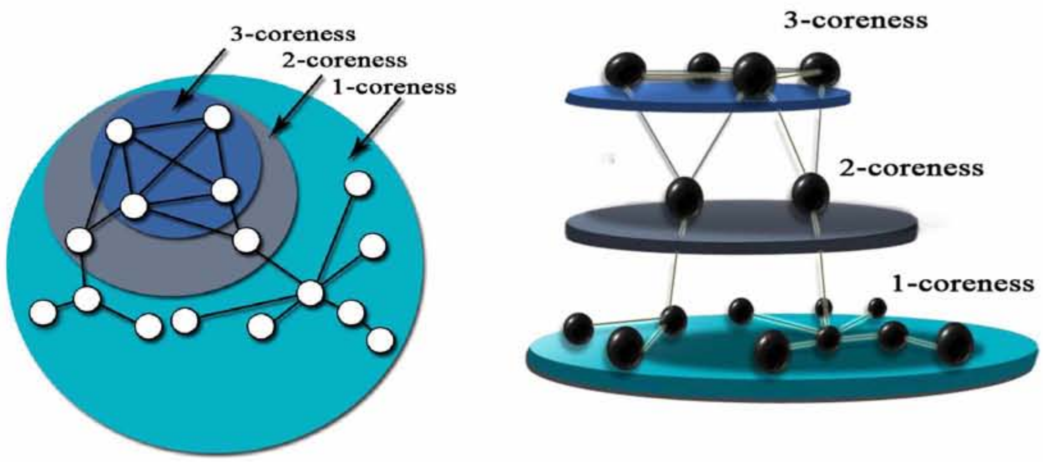


Figure 4 Illustration of *k*-cores retrieved from sample graph

3.1.2 A bit of Map Reduce

MapReduce is a programming model designed for processing very large datasets with a parallel and distributed algorithm over a cluster of machines. The cluster comprises of one master node and several slave nodes. The MapReduce program consists of a sorting and filtering procedure, namely Map(), and an aggregation operation called Reduce(). The MapReduce paradigm was initially proposed in [11] and follows the functional programming model (e.g. Lisp).

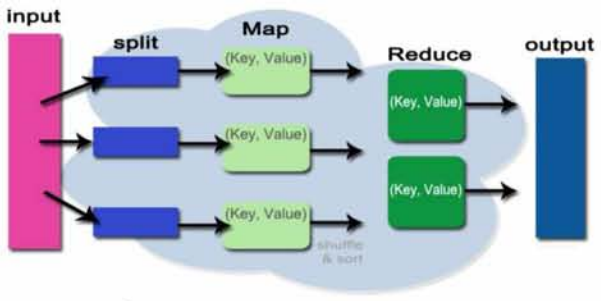


Figure 5 The MapReduce concept

For those who are not familiar with MapReduce we will now describe the internal procedures that take place during a MapReduce job. Initially, the owner of the cluster has to move all data into the distributed file system (HDFS when Hadoop is used). The first thing the master undertakes is to split the input file into independent chunks and distribute them to the slaves which at this point are called Mappers. Concurrently, the master assigns Map tasks to some or all slave nodes of the cluster. During the Map phase each worker reads pair wise (i.e., Key-Value pairs) the chunk he was handed. Afterwards, the Shuffle and Sort phase that takes place is a default procedure to ensure that the input for the Reducer is sorted by Key. Once the sorting is performed some nodes are appointed as Reducers and apply the Reduce method on the lexicographically sorted input. It is imperative to mention that the Reducers are usually fewer than the Mappers in a MapReduce job or sometimes do not even exist (i.e., Map-only jobs). Finally, intermediate results are merged into one final output.

3.1.3 The Hadoop middleware

Hadoop is a software library, or to be more specific a middleware, that allows parallel and distributed processing of large datasets over a cluster of low-cost machines. It uses the MapReduce programming model to process the input files. One of Hadoop's greatest advantages is that it was natively designed to detect and handle system failures at the application layer and that it delivers high-availability without relying on the hardware level. Therefore, Internet giants like Yahoo!, Google and Facebook have been widely using Hadoop for Big Data analysis.

The Hadoop system mainly consists of two components: The Hadoop Distributed File System (HDFS) and the MapReduce framework. Input and output data are written from and to HDFS. Each node in a Hadoop cluster plays a specific role according to its position. The components of Hadoop's architecture seen in Figure 6 are described below in more details.

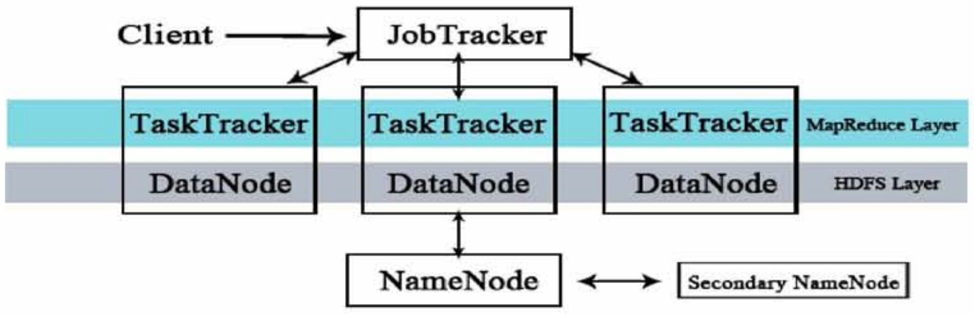


Figure 6 Hadoop's architecture

JobTracker The JobTracker farms out Map and Reduce tasks to specific nodes. Ideally, nodes that receive the task should have the data or should be at least in the same rack. The major problem of the JobTracker is that it is a single point of failure in the cluster meaning that should it not work properly all currently running jobs are stopped [31].

TaskTracker A TaskTracker is a node of the cluster that accepts tasks that cannot exceed a maximum threshold (i.e., the number of slots) from the JobTracker. A task can be a Map, a Reduce or a Shuffle operation [30].

DataNode (DN) A DataNode is a node of the cluster that stores data in the HDFS. For the HDFS to be consistent and functional the cluster should consist of more than one DataNode. Each DataNode stores data replicas [29].

NameNode (NN) A NameNode is the centerpiece of the HDFS. It stores the directory tree of all files in the HDFS but does not store the actual data. It also tracks where all files across the cluster are stored. Unfortunately, like the TaskTracker the NameNode is a single point failure for the Hadoop cluster [28] . To support the proper functionality of the NameNode the Secondary NameNode performs periodic checkpoints.

3.2 The MR-SD algorithm

After providing the necessary background on *k*-shell decomposition and Hadoop's MapReduce we are now ready to describe in this subsection the proposed algorithm, namely The MR-SD Algorithm, and how it maintains the decomposition of a network into its cores. Algorithm 2 and Algorithm 3 describe processes running on the server and client side respectively.

The MR-SD Algorithm, standing for ***M**ap**R**educe-based **S**hell **D**ecomposition*, consists of chained MapReduce jobs that manage to retrieve the *k*-cores of a given network recursively. Our Hadoop cluster consists of five nodes, one master and four slaves. At this point let as denote with node$_c$ a node of our cluster; node$_c$ can either be a slave node or the master. The intuition behind The MR-SD Algorithm is that each node$_c$ is responsible for pruning the input graph and retrieving the *k*-cores, and only a part of the available nodes$_c$ makes the final aggregations.

Based on the above idea, our proposed algorithm is split into two computational parts:

Part 1: The Job's progress-termination

This part of the MapReduce job is responsible for the job set up; this includes setting input and output paths, defining the Mapper and Reducer classes and initializing variables as well. This stage of the algorithm is highly important since it is responsible to increase the coreness value between the pruning phases and decide if the decomposition reached its primary endpoint.

Part 2: The pruning part of the Decomposition (MR-SD)

In this phase of the algorithm the graph is distributed over the collection of the nodes$_c$ as the data set is being split into independent chunks and handed out to worker nodes. Each node$_c$ is entrusted now to perform some of the Map tasks, some of the Reduce tasks or even some of both aforementioned task types during each job run. The goal for the nodes$_c$ at the Map step is to determine if a node of the network graph has *k*-coreness and hand out this information to the Reducers without paying significant additional communication cost. On the other hand, the Reduce step requires that the Mappers have provided the appropriate information in order to apply the changes to the network graph.

We formalize the algorithm in pseudo-code and describe our approach. The initial step of our proposed algorithm is finding out the one-hop neighborhood of each node of the given network graph. The MR-SD Algorithm achieves this by distributing an independent and random chunk of the actual graph across the cluster. Algorithm 2 dictates that a Map task is now forced. At this point, each node is considered as the output Key and each of its neighbors as a single Value. Each time a Reducer receives a Key-Value pair it groups all sorted pairs by Key (*K*) and considers the collection as the one-hop neighborhood of the Key (*K*). Algorithm 2 can be seen as the preparation stage of the actual decomposition procedure.

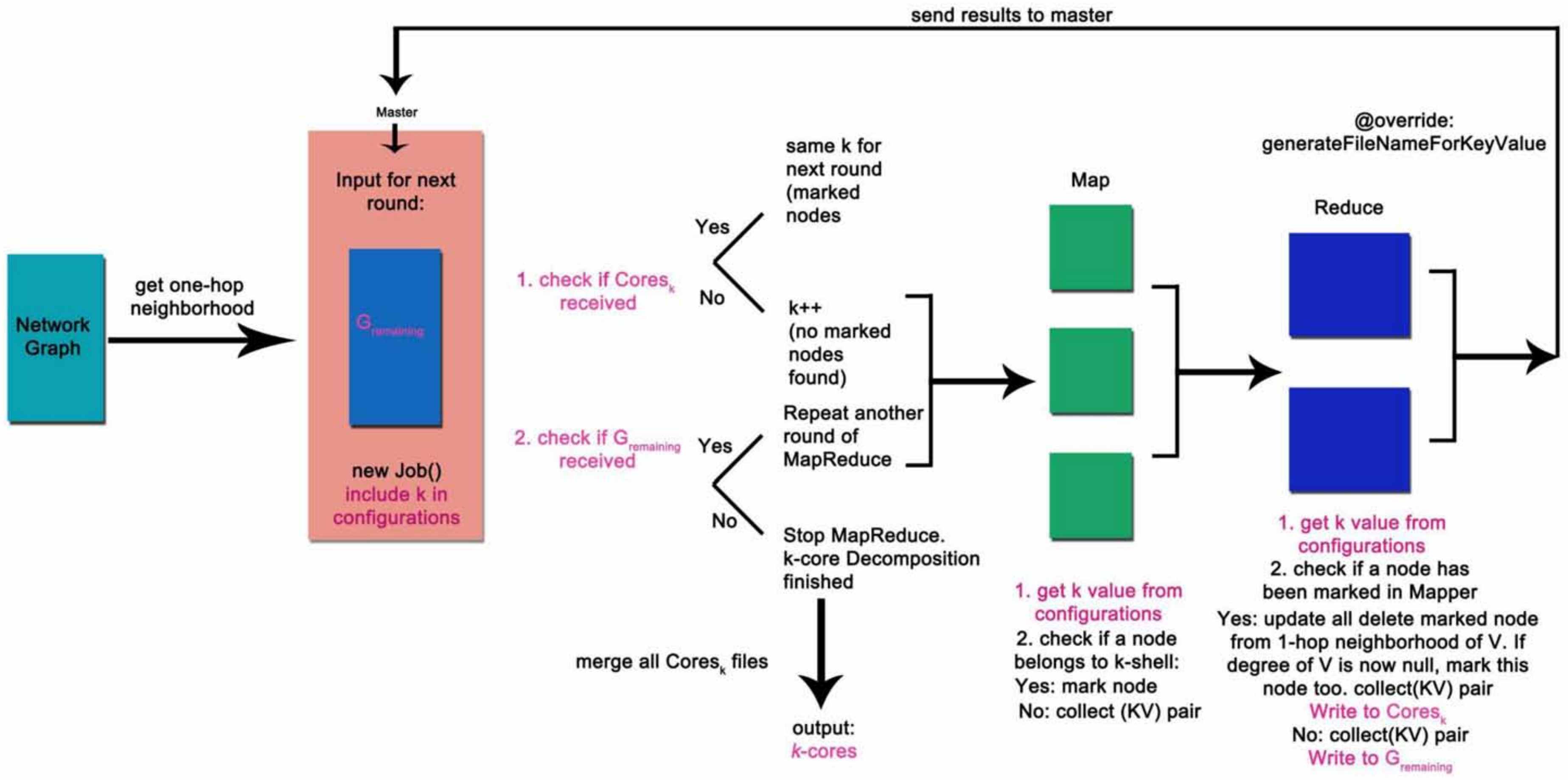

send results to master
Master
Input for next round:
$G_{remaining}$
new Job()
include k in configurations
Network Graph
get one-hop neighborhood
1. check if $Cores_k$ received
Yes
same k for next round (marked nodes
No
k++ (no marked nodes found)
2. check if $G_{remaining}$ received
Yes
Repeat another round of MapReduce
No
Stop MapReduce. k-core Decomposition finished
merge all $Cores_k$ files
output: *k*-cores
Map
1. get k value from configurations
2. check if a node belongs to k-shell:
Yes: mark node
No: collect (KV) pair
@override: generateFileNameForKeyValue
Reduce
1. get k value from configurations
2. check if a node has been marked in Mapper
Yes: update all delete marked node from 1-hop neighborhood of V. If degree of V is now null, mark this node too. collect(KV) pair
Write to $Cores_k$
No: collect(KV) pair
Write to $G_{remaining}$

Figure 7 The Workflow of the MR-SD Algorithm

The intuition behind The MR-SD Algorithm algorithm is that each node$_c$ is can only get a random part of the network graph for processing and that the coreness value k must be disseminated efficiently. Therefore, the master of the cluster is required to update k between the pruning rounds. The master node is also in response for the propagation of k so that all nodes$_c$ see the same instance of k in each Job. Thus, he sets k initially to 1, defines the required configurations of the MapReduce job and announces k to each slave. At this point computations have to be distributed across the cluster.

In order to control the progress and detect termination of the decomposition the master node maintains the following:

- k represents the value of the k-coreness we currently examine; $k \in Z, k > 0$.
- $Cores_k$ is a variable representing the set of nodes that should be included in the current k-shell.
- $G_{remaining}$ is a variable representing the remaining network graph after a pruning round of The MR-SD Algorithm.
- G_{in} is a variable that contains the initial network graph provided by the user.

Algorithm 1: computeOneHopNeighborHood(NetworkGraph G) A Map-Reduce pair to compute the one-hop neighborhood for each node in G

```
Mapper:
on map do
    for each KV pair do
        K ←nodeId
        V ←neighbor;
        collect(K, V);
end map

Reducer:
on reduce do
    for each KV pair do
        collect(K,V);
end reduce
```

Algorithm 1 computation of one-hop neighborhood

Algorithm 2: Driver Routine executed to coordinate the job and detect termination of the k-core decomposition process

```
on initialization do
    configure(Job);
    $G_{remaining}$ ← computeOneHopNeighborhood($G_{in}$);
    k ←1;
end initialization

repeat until each node has k-coreness
    configure(Job);
    <$Cores_k$, $G_{remaining}$> ← MR-SD($G_{remaining}$);

    if $Cores_k$ not received then
        $k$ ← $k$+1;
    if $G_{remaining}$ not received then
        $k$-cores ← mergeIntermediate($Cores_k$);
return $k$-cores;
```

Algorithm 2 Driver routine, auxiliary algorithm

The MR-SD algorithm : A Map-Reduce pair that implements the pruning phase of the *k*-shell decomposition process

```
Mapper:
 on map do
     $k \leftarrow$ get($k$);
     for each $KV$ pair do
         degree $\leftarrow \|V\|$;
         if degree $\leq k$ then
             node $\leftarrow$ mark(node);
             for each $v \epsilon V$ do
                 collect($v$, attachedInfo);
                 collect(node, $k$);
         else
             for each $v \epsilon V$ do
                 $V \leftarrow V + v$;
                 collect($K$, $V$);
 end map
Reducer:
 on reduce do
     $k \leftarrow$ get($k$);
     for each $KV$ pair do
         if attachedInfo received from Mapper then
             for each attachedInfo received do
               oneHopNeiborhood  $\leftarrow \{V\}$ – attachedInfo;
         degree $\leftarrow \|V\|$;
         if degree == 0 then
             mark(node);
             Cores$_k \leftarrow$ collect($K$,$k$);
         else
             $V \leftarrow$ oneHopNeiborhood;
             G$_{remaining} \leftarrow$ collect($K$,$V$);
 end reduce
```

Algorithm 3 The MR-SD Algorithm

Each slave must first retrieve the *k* value from job's configurations before starting a Map task; Hadoop's MapReduce framework provides the suitable tools to implement this approach. Since the master is the only node$_c$ responsible for updating *k* and only at the end of each MapReduce job, i.e. a Map and a Reduce task have both to terminate first, we guarantee that the *k* value a slave receives is always up-to-date and consistent for all nodes$_c$. Right after the announcement of *k*, node$_c$ counts the one-hop neighbors (i.e., it finds out its degree) of a node which are stored in *V* arriving from previous MapReduce job. Once this information is available, the slave has to check if the degree of node *K* under consideration has fallen below the *k* threshold or is equal to *k*. Should this occur, node *K* is marked so that the Reducer can include the node in the current *k*-shell and prune *K* from the remaining network graph. Additional information is then attached (*attachedInfo* variable in the The MR-SD Algorithm algorithm) to all nodes that are one-hop neighbors of *K* so that the Reducer can exclude the current node from their neighborhood. However, if the latter check indicates that the degree of node *K* exceeds the *k* limit then the node's id is collected along with its one-hop neighbors; the node id is the *Key* and the one-hop neighbors the *Value* of the *KV* pair collected and send to the Reducer.

The Reduce phase of the The MR-SD Algorithm algorithm is now ready to start computations. At the beginning of each Reduce task, node$_c$ follows the same protocol as the

Mapper; *k* value becomes available to the worker node by retrieving it from job's configurations using the appropriate tools provided by Hadoop's MapReduce framework. Each time a Reduce is performed, a Reducer receives (sorted by *Key*) data as *KV* pairs. From previous phase of The MR-SD Algorithm *K* represents the node's id and *V* stores its one-hop neighborhood. At this point three possible scenarios can appear and we examine them below:

1. node *K* <u>was marked</u> in preceding Map task
2. node *K* was <u>not marked</u> by previous Mapper
3. node *K* comes coupled with <u>additional information</u> meaning that one or more neighbors are going to be deleted in this pruning round

<u>Scenario 1</u>

In scenario 1, a Reducer receives a Key *K* that represents a node that has been marked during the preceding Map, i.e., the degree of the node has shrunk below *k*. Beyond any doubt this node should be included in the current *k*-shell by the Reducer. In this case, we collect (*K*, *k*) as the *KV* pair for this round.

We should clarify now that we have two output files in the Reduce phase. This arises from the need to have one file (namely the $Cores_k$ file) containing the nodes that are deleted in each round as they built gradually the current *k*-core (here the marked nodes) and another to store the remaining network graph ($G_{remaining}$). This approach ensures that each node is included exactly one time in a *k*-shell and is not further examined after its deletion. Moreover, splitting the output into two separate files produces obviously a smaller input file for the next round and that saves both the Mapper and the Reducer from examining the initial enormous data set constantly.

<u>Scenario 2</u>

In scenario 2, we handle the situation where the node that was examined in the preceding Map phase of The MR-SD Algorithm does not meet the requirements to be included in the current *k*-shell, i.e., its degree is still above the *k* limit. Even if this seems to be a quite basic collection of *KV* pairs, there has actually some further work to be done. The Reducer must now check for additional information attachment on the *KV* pair received. Should the received node come with no additional information attached then the *KV* pair is simply collected.

<u>Scenario 3</u>

For scenario 3, we designed part of the algorithm that handles the case where there is additional information attached (*attached_info*) to the *KV* pair received. The information is associated with the node that has to be removed from the one-hop neighborhood of node *K*. Thus, in scenario 3 we exclude the node from the one-hop neighborhood; the degree is now reduced by one for each neighbor appearing in the attached info. A possible situation arises at this point. The node whose neighborhood we previously pruned can now remain with no neighbor. In this case the node is considered of *k*-coreness and is therefore collected along with the *k*-core value. Should there at least one neighbor after the pruning exist, the node id and the neighbors ids are collected as *KV* pair and sent back to the master in the $G_{remaining}$ output file.

Progress & Termination

At this point we need to prove how the The MR-SD Algorithm in cooperation with the auxiliary algorithms, Algorithm 1 and Algorithm 2, manage to force correctly another pruning round and detect termination when the *k*-cores are formed entirely. Like before we will examine separately each situation that might appear.

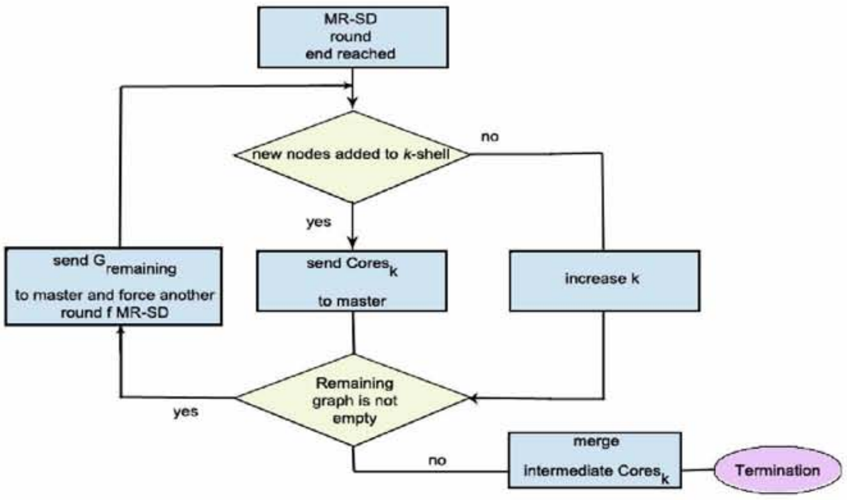

MR-SD round end reached
new nodes added to k-shell
no
yes
send $G_{remaining}$ to master and force another round f MR-SD
send $Cores_k$ to master
increase k
Remaining graph is not empty
yes
no
merge intermediate $Cores_k$
Termination

Figure 8 Flowchart of Progress and Termination

Let us assume we have reached the end of a round of The MR-SD Algorithm; this means that each node of the input has been examined and all *KV* pairs have been received and processed. The master who is the only responsible for the progress of *k*-core decomposition and the termination detection of the The MR-SD AlgorithmAlgorithm receives now either both output files or just one of them. To be more specific:

Case 1: Both $Cores_k$ and $G_{remaining}$ exist

If one or more nodes were added to the current *k*-shell during the Map and Reduce tasks, the master receives at the end of the job the $Cores_k$ file which includes all node ids that have been deleted previously. The master also receives $G_{remaining}$ file including the remaining network graph. In this case, *k* stays the same for another round of MR-SD.

Case 2: Only $G_{remaining}$ recieved

If only the output file $G_{remaining}$ is received, then *k* value has to be updated because no other node has been added to the current *k*-shell. Now, the master must increase *k* value and force another pruning round.

Case 3: Only $Cores_k$ recieved

Finally, if only the $Cores_k$ file is gathered by the master then termination is detected since no other nodes are left for examination. All intermediate files that have been generated during previous rounds are now merged into one final output file.

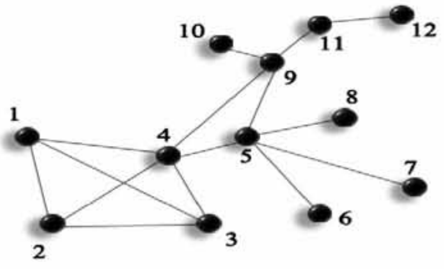


Figure 9 An example network graph

In order to grasp the way our proposed algorithm works, we conclude with an example of a pruning round performed with the The MR-SD AlgorithmAlgorithm and present the network graph used for this purpose in Figure 9.

The objective of this example is to calculate the *k*-cores of the network in Figure 9. No cloud is needed for this specific graph since it is small in size and obviously *k*-cores can be retrieved visually, but it is suitable for our example in order to understand the concept of auxiliary Algorithm 2 and The MR-SD Algorithm.
Clearly one can directly realize that the first job run should exclude nodes 6, 7, 8, 10 and 12 from the given network graph and include them in the *1*-shell.

Figure 10 depicts the following description of the algorithm: The master sets initially *k* value to 1 and the $G_{remaining}$ is split into independent chunks; assume that the calculation of one-hop neighborhood preceded the initialization. It is imperative to mention that $G_{remaining}$ is in the first round the description of the network graph shown in Figure 9 and the input file the user provides.

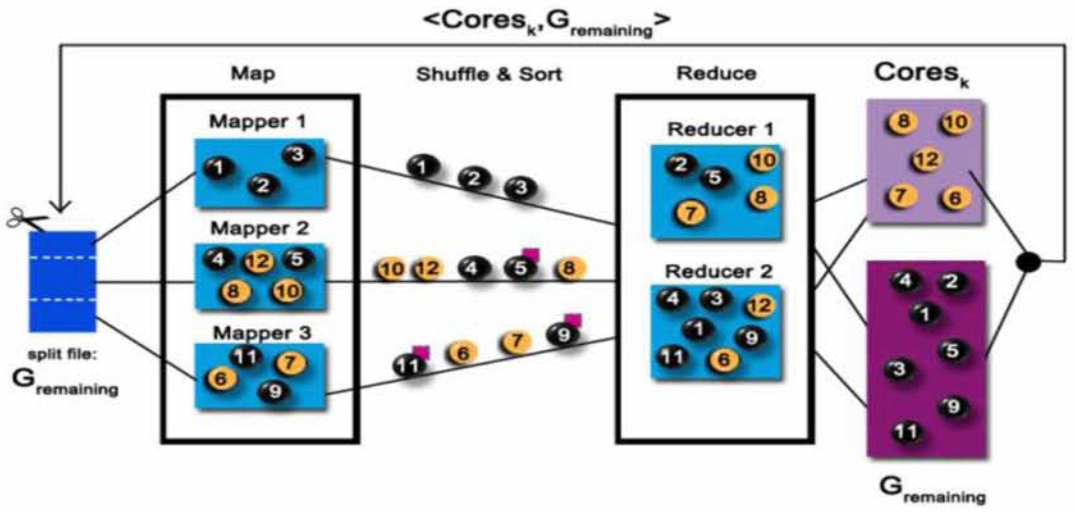


Figure 10 An example running of the proposed algorithm The pink boxes represent the attached_info and the yellow nodes are the marked nodes

<u>Map Phase</u>

In Map phase, all nodes Mapper 1 reads from his chunk of the input file have degree greater than 1. Therefore Mapper 1 collects simply the *KV* pair. Unlike Mapper 1, Mapper 2 and Mapper 3 find nodes with degree equal to one; nodes 8, 10 and 12 and nodes 6 and 7 respectively. Both Mappers mark the nodes so that the forthcoming Reducers include them in *1*-shell. Additionally, they attach information to nodes 5, 9 and 11 to let the Reducers know that they have to delete nodes 6, 7 and 8 from one-hop neighborhood of node 5, node 10 from one-hop neighborhood of node 9 and node 12 from one-hop neighborhood of node 11. The *attached_info* is represented in Figure 9 as pink boxes "glued" to the nodes.

Reduce Phase

In reduce phase, when the Reducers receive *KV* pairs they fist check if the Key, that represents the node's id, was marked previously. In case of nodes 1, 2 and 3 and 4 the Reducers simply collect the *KV* pairs as they appear since they were not marked. On the other hand, for nodes 5, 9 and 11, all nodes that are mentioned in the additional attached information are now deleted from their one-hop neighborhood; here nodes 6, 7 and 8 are removed from node's 5, node 10 from node's 9 and node 12 from node's 11 one-hop neighborhood. Reducers now also check if any of the nodes 5, 9 or 11 has degree less or equal than 1. Reducers collect the (*Key,k*) as *KV* pair for nodes 6, 7, 8, 10 and 12.

Now that all nodes have been examined and both the Map and the Reduce have finished results are sent back to the master. Since some nodes have been deleted during the job, Cores_k containing those nodes is sent to the master and as there is part of the network graph that has not been examined yet, $G_{\text{remaining}}$ is also sent. Consequently, *k* is not increased for another round and the algorithm obviously is not terminating in this round.

Chapter 4

Experimental Evaluation

In this section we provide both the obtained results and their experimental analysis. We describe the Online Social Networks we used to test and assess the performance of our proposed algorithm on real-world network graphs. However, it is imperative to first describe our cluster infrastructure briefly.

4.1 The evaluation platform

Our algorithm allows retrieving the *k*-cores of a network graph in a distributed and parallel way over a collection of connected machines that are using commodity hardware. We tested our proposed algorithm on a cluster which consists of five nodes, one master node and four slave nodes. Each node is equipped with a disk space of 42GB, a 12GB RAM and is an 8-core Intel CPU-based blade running CentOS. The network switch which connects our network storages supports a 10-gigabit Ethernet connection. During each experiment there was no significant interference from other workloads.

4.2 The experimental setting

As mentioned earlier, the The MR-SD Algorithm is the first algorithm in the literature for *k*-shell decomposition that is based on the MapReduce framework. Therefore, there are no competitors in order to compare any efficiency results. However, it is important to notice that we used the algorithms reported in [8] and [26] to decompose our test OSNs, but they run quickly out of memory (in case of the large ones) and never terminated. Thus, we do not present results for them in this work.

4.2.1 The Datasets

Data sets				
Experiment	**Social Network name**	**Number of nodes**	**Number of edges**	**Number of Jobs**
1	**Autonomous systems AS-733**	6474	13895	61
2	**DBLP collaboration network**	317080	1049866	360
3	**Autonomous systems by Skitter**	1696415	11095298	1305
4	**LiveJournal online Social Network**	3997962	34681189	3363
5	**Orkut online social network**	3072441	117185083	5918
6	**Amazon product co-purchasing network**	334863	925872	87
7	**Deaseasome**	7533	22052	118
8	**Protein Interaction Network in budding Yeast**	2361	7182	74

Table 1 Real Online Social Networks used for the evaluation

The eight real Online Social Networks we used for the experimentation part of this work are shown in Table 1. In order to present the size of the networks we also show some of their characteristics. Stanford University and Gephi [27] provide generously the above datasets; datasets 1-6 can be found on https://snap.stanford.edu/ and datasets 7 and 8 can be found here: http://wiki.gephi.org/index.php/Datasets. The networks we used to evaluate the proposed algorithm are ranging from small sized (a few thousand nodes) to very large sized (a few millions of nodes); most of them were also used in [3]. We performed separate experiment for each dataset.

4.2.2 The Performance Measures

In order to assess the MR-SD algorithm we chose three quantities as performance measures of efficiency and we present them below:

Average CPU time spent (msec) stands for the time spent solely by the CPU to perform the computations. All results presented in Table 2 are averaged over all MapReduce tasks.

Average Total Execution Time (sec) stands for the total execution time for the *k*-shell decomposition of the input network. It is different from the previous metric since it also takes into account the communication among the master and the slaves, the time required to store intermediate and final results, and any other additional latency that may occur. The obtained results are averaged over all MapReduce tasks.

Average Total committed heap usage (Bytes) is the memory footprint of the algorithm. It shows the amount of heap memory that is required during the experimentation phase.

4.3 The obtained results

In this subsection we present the obtained results. We also try to capture how our proposed algorithm behaved during the experimentation part. First, we present the averages of the three aforementioned measured described in subsection 4.2.2 and then the plots that concern their precise distribution. For clarity of presentation purposes, we provide the results concerning each dataset in a different plot. Table 2 depicts the average values of CPU time, total execution time and memory footprint per Map job, per Reduce job and per dataset.

Average CPU time spent (msec)			
	Map	**Reduce**	**Total (Job)**
Autonomous systems AS-733	2061.31	2715.74	4777.05
DBLP collaboration network	7696.08	5768.33	13464.42
Autonomous systems by Skitter	61377.72	61982.36	123360.08
LiveJournal online Social Network	79122.92	28758.76	107881.68
Orkut online social network	505243.72	304598.03	809841.74
Amazon product co-purchasing network	9803.56	6339.20	16142.76
Deaseasome	2223.05	2718.64	4941.69
Protein Interaction Network in budding Yeast	1559.32	2256.49	3815.81

Average Total Execution Time (sec)			
	Map	**Reduce**	**Total (Job)**
Autonomous systems AS-733	99.95	9.13	16.18
DBLP collaboration network	104.79	10.30	19.33
Autonomous systems by Skitter	158.55	71.48	90.83
LiveJournal online Social Network	159.53	33.63	56.77
Orkut online social network	311.50	299.78	339.79
Amazon product co-purchasing network	200.13	10.75	20.06
Deaseasome	175.80	8.87	16.26
Protein Interaction Network in budding Yeast	156.41	8.82	15.00

Average Total committed heap usage (Bytes)			
	Map	**Reduce**	**Total (Job)**
Autonomous systems AS-733	379584512	189792256	569376768
DBLP collaboration network	381046693	192004460	573051153
Autonomous systems by Skitter	397844599	199193282	597037881
LiveJournal online Social Network	486663048	201195325	687858373
Orkut online social network	1695512575	157003260	1852515835
Amazon product co-purchasing network	376372495	191817092	568189587
Deaseasome	379584512	189792256	569376768
Protein Interaction Network in budding Yeast	379584512	189792256	569376768

Table 2 The Average resource consumption (Top) Average CPU time, (Middle) Average total execution time, (Bottom) Memory footprint

4.3.1 Average execution time and memory footprint

It is obvious from Figure 11 and Figure 12 that the execution time depends on the number of edges of the graph under consideration. On the other hand, the number of nodes seems to not influence the obtained results in a great extend. This is expected since the number of edges is related to the density of the network graphs; the denser the networks the greater the number of The MR-SD Algorithmrounds that are required to decompose the graph.

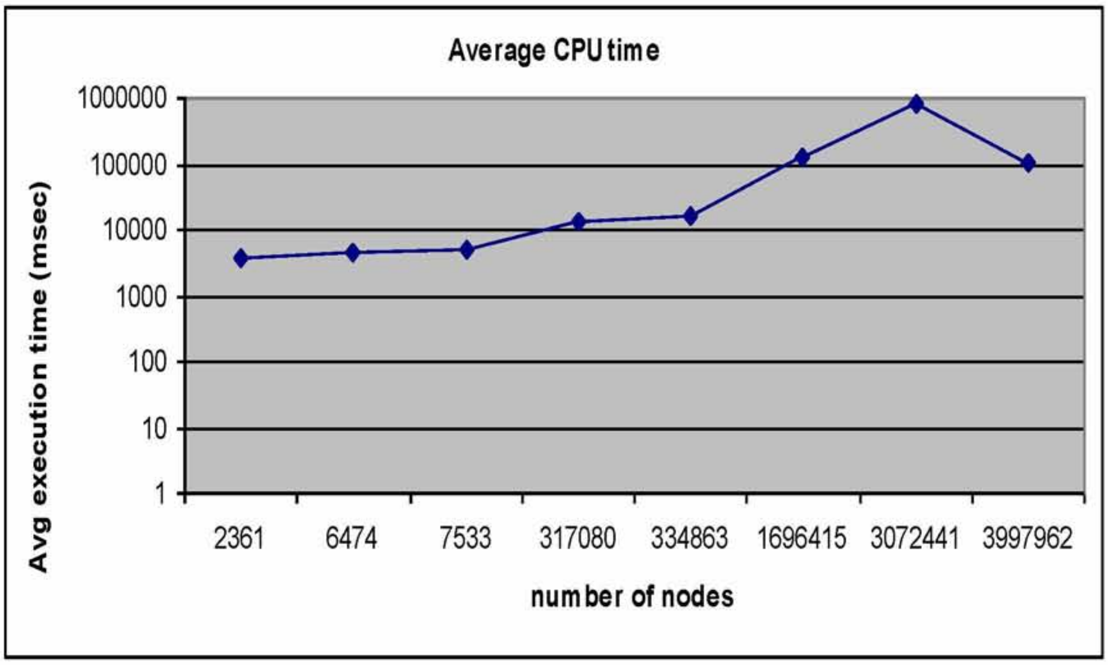


Figure 11 Average CPU time vs. number of nodes

Figure 11 and Figure 12 confirm that a great number of edges are more related to a large time and that the number of nodes has only a small effect on the average CPU time. Results can be seen in Figure 12 (averages) and Figure 15 (per experiment).

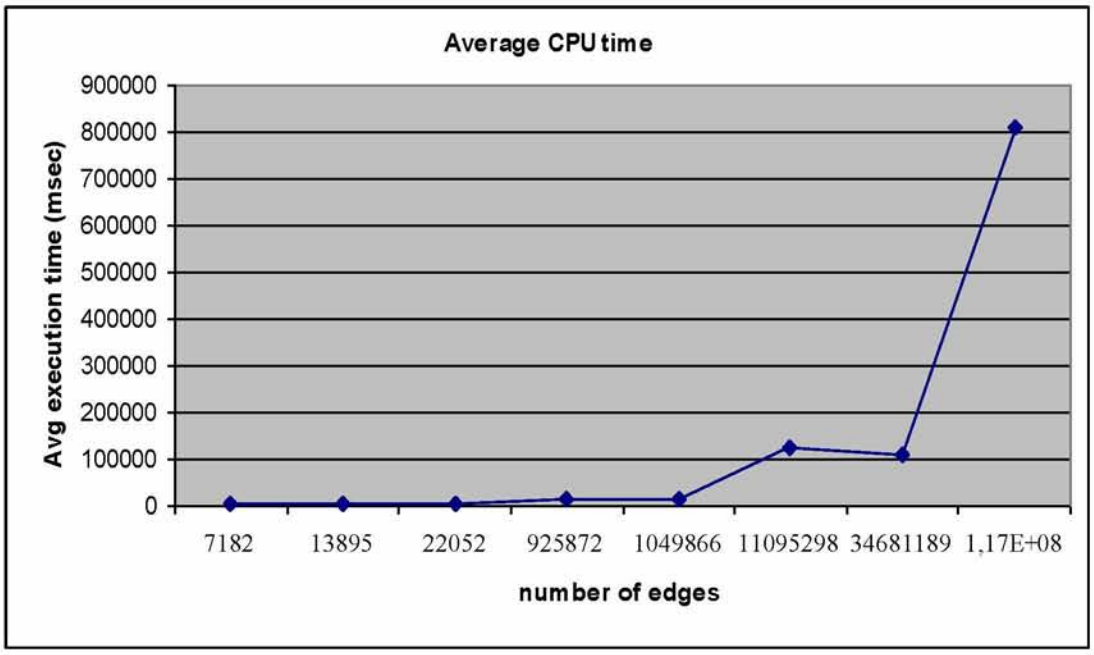


Figure 12 Average CPU time vs. number of edges

4.3.2 Impact of the number of VMs

As mentioned previously the cluster we setup in order to evaluate our algorithm consists of five nodes and in some of our experiments the datasets tested were extremely large. This has consequences on the performance of each worker node. More specifically, the results showed that when running heavy jobs, i.e., really large datasets, on a small cluster some of the nodes$_c$ are obliged to run more than one Map task; the decomposition of the social network in experiment 5 highlights this case. In this experiment the maximum number of the Map tasks exceeded 30 at the beginning and therefore more than 7 Map tasks where assigned to each slave during the first rounds of Algorithm 2 and Algorithm 3; this actually resulted a greater time. For graphs with size ranging from small to medium no significant workload has been noticed. Generally, the MR-SD approach of *k*-shell decomposition manages every time, even with great workload, to retrieve the *k*-cores. A better performance can be achieved when running MR-SD on a larger cluster since Map and Reduce tasks can be distributed more efficiently. Results can be seen in Figure 12 (averages), Figure 13, Figure 14 and Figure 16.

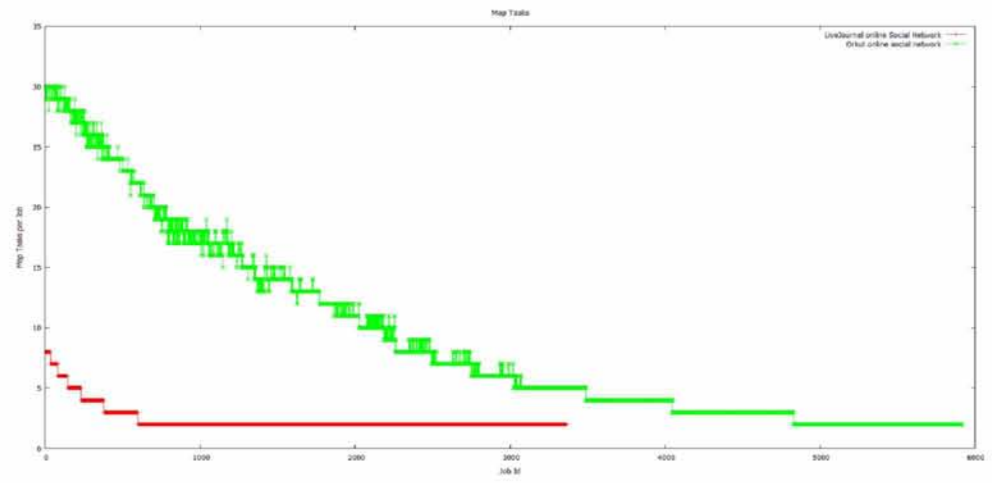


Figure 13 Map tasks during experiments 4 and 5

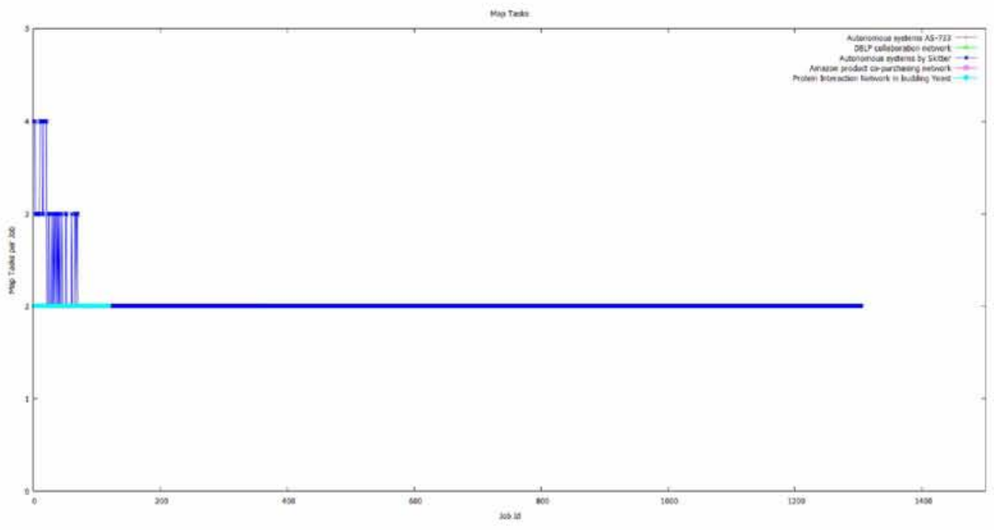


Figure 14 Map tasks during experiments 1, 2, 3, 6 and 8. Most of the Map tasks overlap with the Map tasks of experiment 8.

Finally, in Figure 15, results can be seen per experiment. x-axis in each plot depicts the total committed heap usage per job (in MBytes), and the y-axis depicts the total CPU time spent per job (in msecs). In general, the Reduce tasks cost far less in computation time than the respective Map tasks, which is expected since the Reducers perform mainly aggregation operations.

4.3.3 Impact of network density

As mentioned earlier, the performance of the MR-SD algorithm depends on the network density, i.e., the number of network edges. Figure 12 shows that the network density impacts exponentially the execution time. This is expected since the number of jobs increases with the number of edges. For example, some millions edges or even more edges, result a significant greater execution time than the graphs with a small number of edges. Table 2 includes in details the performance of MR-SD for each network. In particular, the numbers of the obtained results highlight the impact of the network density when comparing the results of CPU time of dense networks (e.g., experiment 5) with those of more sparse social networks (e.g., experiment 8).

4.3.4 Impact of the machine load

Processing a very large sized network is a difficult procedure until a large number of nodes are pruned. This indicates that during the first pruning rounds of MR-SD, a greater resource demand is observed. Indeed, both CPU time measured and heap size requirements are considerably greater. Figure 15 and Figure 17 clearly depict that the workload demands for large network graphs require an increasing processing power while the heap size remains actually almost stable during this interval. The same is observed for networks with a size ranging from small to medium although to a much lesser extent. One especially interesting outcome is that Reduce tasks tend to be more time consuming in the first 20% to 45% of the pruning rounds than in the last ones. In fact, the execution time seems to follow a decreasing exponential trend for very large graphs. However, for networks with a small or medium size we observe a rather flat execution time during the Reduce phase.

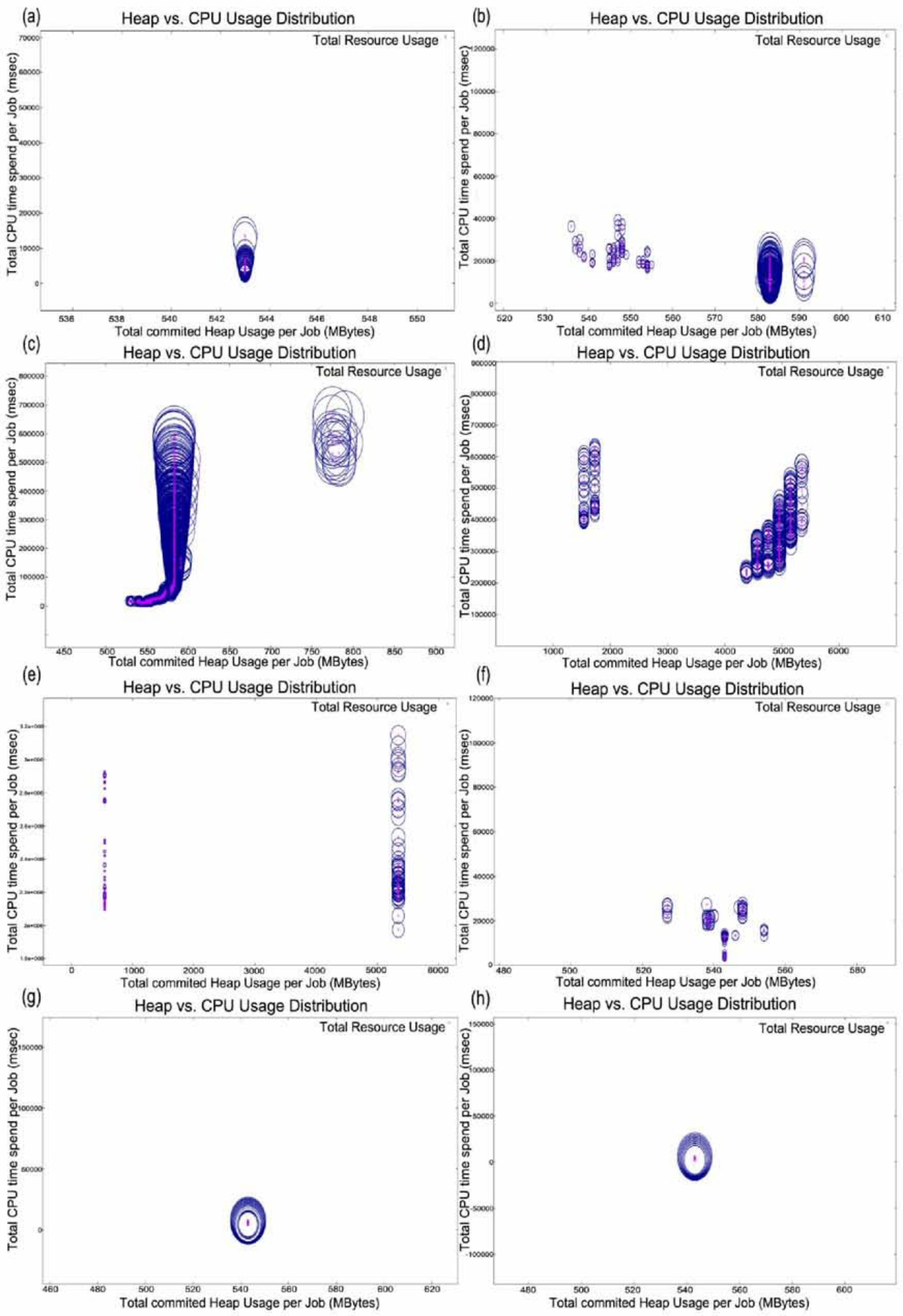


Figure 15 Heap vs. CPU Usage Distribution. The size of the radius of each circle is relative to the square root of the ratio of CPU and Heap usage multiplied by a constant number c. a) Experiment 1, c=0.1, b) Experiment 2, c=0.1, c) Experiment 3, c=0.8, d) Experiment 4, c=5, e) Experiment 5, c=5, f) Experiment 6, c=0.2, g) Experiment 7, c=2, h) Experiment 8, c=2

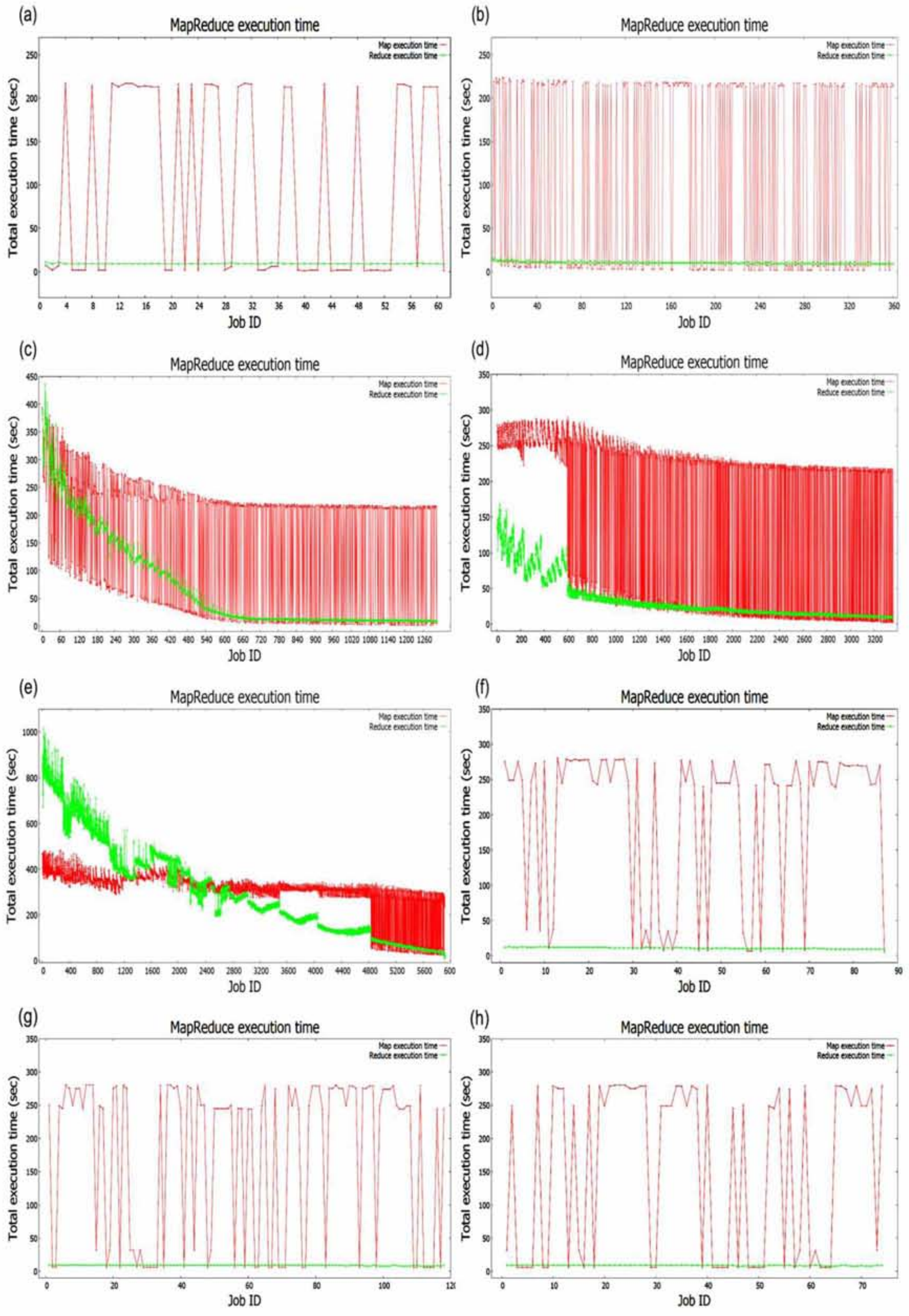


Figure 16 Map and Reduce task execution time (red is used for Map tasks and green for Reduce tasks) a) Experiment 1, b) Experiment 2, c) Experiment 3, d) Experiment 4, e) Experiment 5, f) Experiment 6, g) Experiment 7, h) Experiment 8

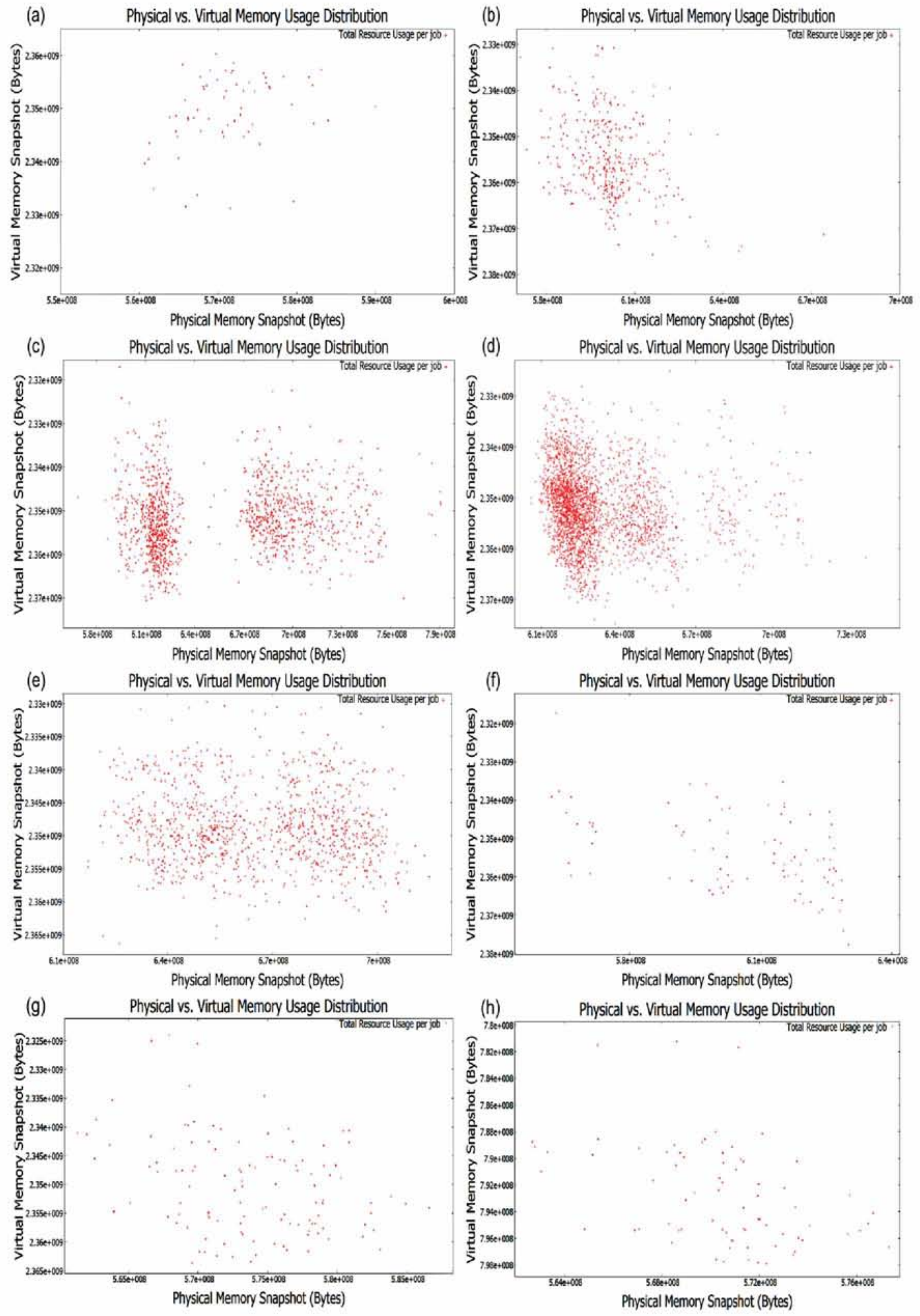


Figure 17 Virtual Memory vs. Physical Memory (MB) Usage a) Experiment 1, b) Experiment 2, c) Experiment 3, d) Experiment 4, e) Experiment 5, f) Experiment 6, g) Experiment 7, h) Experiment 8

Chapter 5

Conclusions & Future Work

The *k*-shell decomposition of an online social network is a significant task involved in social network analysis. It has found many application scenarios in many fields. For example, it can be used for the discovery of influential spreaders, for community detection, for analysis of the Internet structure, for visualization purposes, and so on. In the past many *k*-shell decomposition algorithms have been proposed. However, none of the already existing algorithms is suitable when the input graph is thought to be BigData, i.e. enormously large, or when the computations take place in huge Hadoop-based clusters such as those deployed by current Internet giants (Google, Yahoo, LinkedIn). In this work, motivated by the latter unsuitability and lack of existing solutions to deal with this, we designed a MapReduce-based distributed *k*-shell decomposition algorithm for social networks, namely MR-SD. We addressed the challenges involved in the design of a parallel and distributed version of a graph decomposition technique, which is highly sequential in its nature, and provided an effective and efficient algorithm able to scale to millions of graph nodes and edges. For the implementation of MR-SD we used the Hadoop middleware, and assessed its performance for eight real social networks of varying size and density. We investigated the performance of the algorithm in terms of pure CPU time, of total execution time and memory footprint. We recognized its virtues and suitability for modern distributed environments.

As a future work, we plan to work on some aspects of the MR-SD algorithm in order to gain some additional speedup. In this direction we will try to study areas that involve the communication among slaves so as to optimize it and also to develop a variation of the algorithm for annotated networks, e.g., weighted. We will also work on how to best distribute tasks across the cluster. Finally, we plan to perform experimentation of our proposed algorithm on a MapReduce environment of a major cloud service provider in order to highlight its scalability and achieve further speedup.

6 References

[1] K. Pechlivanidou, D. Katsaros, L. Tassiulas, "MapReduce-based Distributed K-shell Decomposition for Online Social Networks", *In Proceedings of the 2nd International Workshop on Personalized Web Tasking (PWT), In the context of the IEEE 10th World Congress on Services*, Anchorage, Alaska, USA, June 27 – July 2, 2014

[2] P. Basaras, D. Katsaros, L. Tassiulas, "Detecting influential spreaders in complex, dynamic networks", *IEEE Computer magazine*, vol. 46, no. 4, pp. 26-31, 2013

[3] H. Aksu, M. Canim, Y.-C. Chang, I. Korpeoglu, O. Ulusoy, "Multi-resolution social network community identification and maintenance on big data platform", *Proceedings of the IEEE International Congress on Big Data (BigData)*, pp. 102-109, 2013.

[4] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, E. Shir, "A model of Internet topology using *k*-shell decomposition", *Proceedings of the National Academy of Sciences*, vol. 104, no. 27, pp. 11150-11154, 2007.

[5] V. Batagelj and M. Zaversnik, "An O(m) Algorithm for Cores Decomposition of Networks", University of Ljubljana, Department of Theoretical Computer Science, Ljubljana, Slovenia, *Preprint series*, vol. 40, 2002. Available at http://arxiv.org/abs/cs.DS/0310049

[6] J. Cheng, Y. Ke, S. Chu, M.T. Ozsu, "Efficient core decomposition in massive networks", *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, pp. 51-62, 2011

[7] A. Montesor, F. de Pellegrini, D. Miorandi, "Distributed *k*-core decomposition", *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 2, pp. 288-300, 2012

[8] A. Garas, F. Schweitzer, S. Havlin, "A k-shell decomposition method for weighted networks", *New Journal of Physics*, vol. 14, 2012

[9] A.E. Sariyuce, B. Gedik, G. Jacques-Silva, K.-L. Wu, U.V. Catalyurek, "Streaming algorithms for k-core decomposition", *Proceedings of the VLDB Endowment*, vol. 6, no. 6, pp. 433-444, 2013

[10] D. Miorandi and F. de Pellegrini, "K-shell decomposition for dynamic complex networks", *Proceedings of the Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks (WiOpt)*, pp. 488-496, 2010

[11] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters", *Proceedings of USENIX Symposium on Operating System Design and Implementation (OSDI)*, pp. 137-150, 2004.

[12] J. Lin and C. Dyer, "Data-Intensive Text Processing with MapReduce", *Synthesis Lectures on Human Language Technologies*, Morgan & Claypool Publishers, 2010.

[13] P. Pantel, E. Crestan, A. Borkovsky, A.-M. Popescu , V. Vyas, "Web-scale distributional similarity and entity set expansion", *Proceedings of Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 938-947, 2009.

[14] S. Papadimitriou and J. Sun, "DisCo: Distributed co-clustering with Map-Reduce: A case study towards petabyte-scale end-to-end mining", *Proceedings of the IEEE Interational Conference on Data Mining (ICDM)*, pp. 512-521, 2008.

[15] W. Zhao, H. Ma, Q. He, "Parallel k-means clustering based on MapReduce", *Proceedings of the International Conference on Cloud Computing (CloudCom)*, LNCS, vol. 5931, pp. 674-679, 2010.

[16] G. Sudha-Sadasivam and G. Baktavatchalam, "A novel approach to multiple sequence alignment using Hadoop data grids", *Proceedings of the Workshop on Massive Data Analytics on the Cloud (MDAC)*, 2010.

[17] A. Abouzeid, K. Bajda-Pawlikowski, D. Abadi, A. Silberschatz, A. Rasin, "HadoopDB: An architectural hybrid of Map/Reduce and DBMS technologies for analytical workloads," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp.922-933, 2009.

[18] Y. Tao, W. Lin, X. Xiao, "Minimal MapReduce algorithms", *Proceedings of the 2013 ACM International Conference on Management of Data (SIGMOD)*, pp. 529-540, 2013.

[19] P. Paraskevopoulos and A. Gounaris, "Optimal tradeoff between energy consumption and response time in large-scale MapReduce clusters". *Proceedings of the Panhellenic Conference on Informatics (PCI)*, pp. 144-148, 2011.

[20] A. Kala Karun and K. Chitharanjan, "A review on Hadoop — HDFS infrastructure extensions", *Proceedings of the IEEE Conference on Information & Communication Technologies (ICT)*, pp. 132-137, 2013.

[21] S. Wasserman and K. Faust, "*Social Network Analysis: Methods and Applications*", Cambridge University Press, 1994.

[22] D. Katsaros, N. Dimokas, L. Tassiulas. "Social network analysis concepts in the design of wireless ad hoc network protocols", *IEEE Network magazine*, vol. 24, no. 6, pp. 23-29, 2010.

[23] M. Giatsoglou and A. Vakali, "Capturing social data evolution using graph clustering", *IEEE Internet Computing* vol. 17, no. 1, pp. 74-79, 2013.

[24] S.B. Seidman, "Network structure and minimum degree", *Social Networks*, vol. 5, no. 3, pp. 269–287, 1983.

[25] S. Carmi, S. Havlin, S. Kirkpatrick, Y. Shavitt, E. Shir, "A model of Internet topology using *k*-shell decomposition", *Proceedings of the National Academy of Sciences*, vol. 104, no. 27, pp. 11150-11154, 2007.

[26] R.-H. Li, J.X. Yu, R. Mao, —Efficient core maintenance in large dynamic graphs‖, *IEEE Transactions on Knowledge and Data Engineering*, to appear, 2014. Available at http://arxiv.org/abs/1207.4567

[27] Bastian M., Heymann S., Jacomy M. (2009). Gephi: an open source software for exploring and manipulating networks. International AAAI Conference on Weblogs and Social Media.

[28] http://wiki.apache.org/hadoop/NameNode

[29] http://wiki.apache.org/hadoop/DataNode

[30] http://wiki.apache.org/hadoop/TaskTracker

[31] http://wiki.apache.org/hadoop/JobTracker

[32] http://en.wikipedia.org/wiki/Social_network

[33] http://www.nature.com/srep/2013/131018/srep02980/images_article/srep02980-f6.jpg

[34] http://www.nature.com/nphys/journal/v6/n11/carousel/nphys1746-f1.jpg

[35] http://4.bp.blogspot.com/-_BuuVu4MTAo/ToGwMHlc15I/AAAAAAAAsWo/XObAuT