Date of acceptance Grade

Instructor

Influence Maximization in Large Scale Graphs

Behrouz Derakhshan

Helsinki May 10, 2015

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Laitos — Institution — Department Department of Computer Science
Tekijä — Författare — Author Behrouz Derakhshan	
Työn nimi — Arbetets titel — Title Influence Maximization in Large Scale Graphs	
Oppiaine — Läroämne — Subject Computer Science	

Työn laji — Arbetets art — Level	Aika — Datum — Month and year May 10, 2015	Sivumäärä — Sidoantal — Number of pages 52 pages + 0 appendices

Tiivistelmä — Referat — Abstract

Maximizing influence in graphs, typically applied to Social Networks, is the problem of finding a set of nodes with the highest overall influence on the entire graph. In marketing domain for example, it is used to find the set of people who have the highest influence on their local communities. As a result, instead of blindly marketing a product to a large group of people, the product is marketed to this group of selected users, and they will in turn help spreading the word. The problem has been studied extensively, and several state of the art methods have been proposed. But all of these methods have one common flaw, none of them are scalable. Even on small graphs, current methods take extremely long amount of time and introduction of bigger data sets have rendered some of these methods completely useless. Over the past two decades, collection of data has become easier and a very common practice. This is mostly credited to the advancements in hardware and software technologies as well as the introduction of World Wide Web. To overcome issues related to big data sets, large scale data processing platforms have been developed to tackle scalability issues of problems similar to the influence maximization. Most notably are the two frameworks called Hadoop and Spark that contain many features for simple data processing, machine learning and graph processing. In this thesis work, some of the current influence maximization algorithms are implemented in these two frameworks, some new methods are proposed, experiments on graphs of different sizes are performed and the results are reported.

ACM Computing Classification System (CCS):
Information systems –> Graph-based database models,
Information systems –> MapReduce-based systems

Avainsanat — Nyckelord — Keywords
Graph, Influence maximization, Big Data, Hadoop, Spark

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

Over the past two decades with the introduction of social networks, the study of interactions between people have gained a lot of attention. Social networks, represented using a graph of relationships between people, carry a lot of information about the individuals. Sociologists have studied social networks to better understand what derives people to form friendships, groups and communities. Journalists and public figures have used social networks such as Twitter and Facebook to report the news and communicate with people. And last but not least, marketing experts have used social networks to better understand the market, and increase the benefit by performing targeted advertisement.

Influence maximization is one of the marketing applications in social networks. It emphasizes on the effect of individuals on each other and how utilizing it can increase a product's popularity. Domingo and Richardson [DpR01] were the first to study the spread of information in markets. They proposed a method to estimate the value of customers in terms of the network. Based on this method, increase in profitability can be easily determined when an item is advertised to a customer. Kempe et al. [KKT03] formally defined the problem of Influence Maximization. By studying how influence is propagated under two models: *Independent Cascade* and *Linear Threshold*, the problem is defined as finding an initial set of vertices(called *active set* or *seeds*), that will influence the most number of vertices. Influence maximization is proven to be an **NP-hard** problem. Kempe et al. provided a greedy algorithm for solving it but this solution suffered from scalability issues. Even on small graphs, finding the initial active set requires a substantial amount of time and resources. Following Kempe's work, many attempted to reduce the running time of the algorithm. Leskovec et al. [LKG07] and Chen et al. [CWY09] proposed methods that reduced the running time greatly. However, the proposed methods only worked well on small graphs and none of them can be applied to large scale graphs that are very common nowadays.

In the last decade a new field in computer science called **Big Data**, has emerged that involves working with massive amount of data. Given the widespread usage of internet, it is no surprise that a large amount of data is being generated daily. Amazon, Facebook and Twitter store information about their users, governments store data of their citizens, infrastructures, health care and economy. These are only a few examples, and many other organizations are collecting data, and storing it to help derive their businesses. Storage and processing of data at this scale requires an immense amount of effort. Big data technologies focus on building tools for efficient collection, storage, and processing of the data.

Probably the most notable big data frameworks are *Hadoop* and *Spark*. Hadoop is an open source project that is based on Google's MapReduce framework introduced by Dean and Ghemawat [DjG04]. It was originally developed to include the map reduce framework and a file system, called *Hadoop Distributed File System(HDFS)*, capable of catering terabytes or even petabytes of data reliably and efficiently. However, Hadoop now includes a wide array of frameworks, each one designed for special purposes. Some examples of these frameworks are Hive that provides a SQL interface to the data and Mahout which is a machine learning framework. Spark is another open source project. It approaches data processing differently from the traditional map reduce and tries to address some of the issues in Hadoop. Both Hadoop and Spark include a framework for processing data in graph format. They provide efficient storage and implementation of some of the well known graph algorithms, such as connected component and page rank. These frameworks are:

- *Giraph*, an open source project based on Google's Pregel framework by Malewicz et al. [MAB10]. It is built on top of Hadoop's map reduce
- *GraphX*, an open source project on top of Spark

The aim of this thesis is to study the problem of influence maximization, current state of the art solutions, as well as their advantages and disadvantages. The thesis also includes an investigation of applicability of running influence maximization algorithms on large scale graphs. It highlights why some of the earlier methods are not suitable in big data domain. And finally, it proposes solutions for solving influence maximization in large scale graphs.

The rest of this document is structured as follows. Section 2 presents the history of influence maximization problem, solutions that have been proposed and their advantages and disadvantages. In section 3 Big Data frameworks, their history and the current states of them are discussed. Two of the very common frameworks, i.e. Hadoop and Spark, are studied in detail. In section 4, an overview of large scale graph processing, followed by Giraph and GraphX details are presented. Section 5 studies the scalability issue of influence maximization. Several methods either novel or based on previous methods are proposed and the implementation details are discussed. In section 6 and 7 details of experiments and their results are explained. And finally, section 8 concludes the thesis and mentions future works and open problems related to influence maximization.

2 Influence Maximization

Effective marketing and advertisement methods have been a major application of Knowledge Discovery and Data mining. Prior to Domingo and Richardson [DpR01] the focus has always been on direct marketing. Contrary to traditional mass marketing, where a product was marketed to all the customers, in direct marketing the goal is to find the most potential and profitable costumers. While direct marketing is a great tool for increasing the profit, it only targets individuals and disregards the relationship between users. People's decision in purchasing an item is not only influenced by the product itself, but also by friends, colleagues and acquaintances. With introduction of Internet and social networks, gathering information about users became easier and more feasible. Social networks nowadays contain information about millions of people. They also contain information related to people's relationship with each other. Utilizing these information can make marketing more profitable, by targeting people that are more likely to promote the product to their acquaintances.

Domingo and Richardson [DpR01] discussed the use case of influence maximization in marketing, calling it *Viral Marketing*. They have modeled a market as a social network consisting of users and products. In such a network, probability of users buying products not only depends on the product itself but also on the neighbors of the user and the set of marketing actions being applied to the network. Hence the market can be formulated as join probability distribution. Through these a function was devised that calculates the global lift in profit given a marketing action was applied to some users. Starting from an initial configuration, using different optimization algorithms a local maxima for the function can be reached. They experimented on a collaborative filtering data set (MovieLens[1]) in which users have rated a list of movies. They have first constructed the network with the user ratings, by choosing similar users (calculating the Pearson correlation coefficient of users) as neighbors. They have performed experiments on the data set and compared their method with mass marketing and direct marketing. Their method has increased the profit the most. Due to non-linearity of the model, it did not scale well for bigger size networks, hence they have proposed a new linear model in another paper [DpR02]. Not only the new model decreased the computation time, they simplified the equation for calculating the network value of customers, and made it easier to incorporate more complex marketing action. They have experimented with Epin-

[1]https://movielens.org/

ions[2] data set, which is a website where users can rate items. It also has a feature called trusted users, where each user can select many other users as trusted sources of reviews. Domingo et al. [DpR02], hence, applied their model to the data set, by considering trusted users of a user as his neighbors. Their experiments showed that a continuous marketing action (a marketing action that can have unlimited values, such as a discount) performs much better than boolean marketing actions (one that can either be true or false, such as whether or not a discount should be given to a user with no regards for the amount of the discount).

These two papers, are considered the first works in influence maximization. Several other authors have used the concepts here to further improve and propose new methods for calculating the spread of nodes through a network.

2.1 Information Diffusion Models

With introduction of Social Networks, the topic of information diffusion has gain interest in research community. Information diffusion models describe how information and data are being propagated and transferred in social networks. Information diffusion is not only limited to social networks, for many years the topic has been studied to better understand for example how a disease is being spread among people or how contaminated water would affect a water pipe network. In influence maximization, two information diffusion models stand out among others.

- Linear Threshold Model:
  Consider a graph $G = (V, E)$, where all the nodes can be either active or non-active. Inactive nodes may become active as more of their neighbors become active but the reverse is not possible. In *Linear Threshold Model (LT)*, each node v is influenced by each of its neighbor u according to a weight $b_{u,v}$ such that $\sum_{u\, neighbor\, of\, v} b_{u,v} \leq 1$. The process starts with an initial set of active nodes and at the beginning of the process each node v will randomly chose a weight threshold $0 < \theta_v < 1$, this value determines the tendency of node v to become active. The diffusion process moves forward deterministically in a series of discrete time steps t and continues until no new node becomes active. In each time step t, all of the active nodes from step $t-1$ remain active and any non-active node v is activated if $\sum_{u\, activeneighbor\, of\, v} b_{u,v} \geq \theta_v$. θ_v is chosen randomly to indicate our lack of knowledge about node's influenceability.

[2]http://www.epinions.com/

- Independent Cascade Model:
  Independent Cascade Model (IC) is a stochastic undeterministic process. Same as linear threshold model, it starts with an initial set of active nodes and the process continues in a series of discrete time steps. In each time step t, all of the active nodes from step $t-1$ will remain active and any node that was activated in step $t-1$ tries to activate all of its neighbors. The probability of a node u activating one of its neighbors v is defined by a system property $p_{u,v}$. The process continues until no more new nodes can become active. Every node will try to activate its neighbors independently of other nodes, and if during one time step, a node v first becomes active by node u the rest of v's neighbors can skip the activation process. It is important to note that in each time step only nodes that became active in the previous time step will try to activate their neighbors. This means each activated node will only have one chance of activating its neighbors in the entire diffusion process. In classical Independent Cascade, the activation probability $p_{u,v}$ (probability of u activating v) is defined as a system property and it either takes one constant value or a value randomly chosen from a small set of values. There are different methods for choosing the activation probability, and some of them are described in section 2.3.

2.2 Influence Maximization Problem

Kempe et al. [KKT03] were the first to formulate Influence maximization as an optimization problem and proved that the problem is NP-hard. By studying how influence propagates through networks using the two diffusion models discussed earlier (IC and LT), they have formally expressed Domingos and Richardson's [DpR01] optimization problem in the context of the above models. Both diffusion models involve an initial set of nodes A, the influence of the set A denoted by $\sigma(A)$ is then defined as the number of nodes at the end of the process. Formally, the influence maximization problem can be defined as:
Given a graph $G = (V, E)$, find a subset $A \subseteq V$, that maximizes the function $f(A)$, which is the size of the spread function $\sigma(A)$ under either of the diffusion models subject to $|A| = k$ where k is the budget. Mathematically it is defined as:

- $A = \arg\max_{A \in \mathcal{P}_k(V)} f(A)$
- where $f(A) = |\sigma(A)|$ and $\mathcal{P}_k(V) = \{V' : V' \subseteq V, |V'| = k\}$

Finding the optimal set is NP-hard but they proposed an approximation algorithm for the problem which guarantees the solution to be a factor of $(1 - 1/\mathrm{e} - \varepsilon)$ of the optimal solution (slightly better that 63%). Approximate solution for these models is developed based on *submodular functions* by Nemhauser et al. [NEM78]. A function f is *submodular* if it satisfies a natural "diminishing returns" property. It means that the marginal gain of adding a node v to a set A is always greater than or equal to adding v to a super set of A .

$$f(A \cup \{v\}) - f(A) \geq f(S \cup \{v\}) - f(S), \text{ where } A \subseteq S$$

The greedy algorithm proposed by Kempe et al. [KKT03] is described in Algorithm 1. They have run experiments on co-authorship in physics theory section of arXiv [3]

Algorithm 1 Influence Maximization: Greedy Algorithm [KKT03]

```
Require: $G(V, E)$: Graph $G$, with Vertex set $V$ and Edge set $E$
Require: $k$: Initial Set Size
  S = $\emptyset$
  for i = 1 to k do
      for every $v \in V$ and $v \notin S$ do
          $\delta_v = \sigma(v)$
      end for
      $v^* = \arg\max_v \{\delta_v\}$
      $S = S \cup v^*$
  end for
  Output $S$: Set of initial active vertices
```

data set. The results show that their greedy algorithm performs better than other methods such as random selection, high-degree or central nodes. Both in estimating the spread and also in the greedy method, $\sigma(S)$ has to be calculated. But since both of the diffusion processes involve random elements(θ_v in Linear Threshold and activation of new nodes in Independent Cascade) to achieve a better estimate of the total spread, the models have to be run many times and the spread is calculated as the average of all the runs. The nature of the spread function makes the algorithm highly non scalable. Even on medium sized graphs running the simulation will take a lot of time, but on the other hand a good solution is only guaranteed if the simulation is run many times. The value of $\sigma(v)$ does not only depend on v itself, but it also

[3]www.arxiv.org

depends on the other active nodes. Hence in each iteration of the algorithm where a new node is added to the active vertices list, the value of $\sigma(v)$ has to be recalculated. This will greatly affect the performance of the greedy method.

Leskovec et al. [LKG07] in their Cost effective outbreak detection in networks, took upon the task of optimizing the greedy method. Although they did not work on solving the problem of maximizing influence, they have proven their method can be adapted to the problem of maximizing the influence using the greedy algorithm. They studied two problems from different domains which share some similarities with influence maximization. The first problem is finding the best placement of sensors in a water network, so that contaminated water can be detected as quickly as possible. The other problem is blog network, which is to find blogs that contains the maximum amount of news that cascades from different sources in shortest possible time. Their advanced greedy method, called *CELF*, utilizes the submodularity of the objective function, which reduces the amount of computation needed to be made in each iteration. Their method is 700 times faster than a simple greedy function. Their observation is that, if A is subset of B, the marginal gain of adding a node to B is always less than or equal to the marginal gain of adding the same node to A. Hence, instead of recomputing the gain for all the node in each iteration, we can go through the nodes in decreasing order of their spread and recompute their values. The change is usually very small and most of the times the value on top of the list stays on top. Algorithm 2 describes CELF method. As can be seen from the algorithm description, the difference between CELF and the normal Greedy method, is that in CELF, $delta_v$ is pre-calculated for all the vertices first, then starting from the biggest value, in each iteration δ_u is recalculated for vertex u if the value is still higher than the next vertex in the list, then node u is added to S otherwise the next vertex is chosen. As in the previous algorithm, $\sigma(v)$ is the expected spread of the node v under the IC model.

Chen et al. [CWY09] proposed several different methods for solving the problem of influence maximization. The first group of algorithms, were improvements upon Kempe et al. [KKT03] and Leskovec et al. [LKG07] greedy methods. In their greedy method, they have modified the simulation process. Instead of trying to activate nodes based on the edge probability one by one, they have first created a new Graph G' which is sampled from the original graph based on the edge probability. Now the process of estimating the spread of initial set S is just finding the number of reachable nodes in G' from S, this process is repeated many times and the result is the average for all of the G' sampled Graphs. Although this method is much faster

Algorithm 2 CELF Algorithm [LKG07]

```
Require: $G(V, E)$: Graph $G$, with Vertex set $V$ and Edge set $E$
Require: $k$: Initial Set Size
  S = $\emptyset$
  for i = 1 to k do
    set $\delta_v = \sigma(v)$ for $v \in V \setminus S$
    while TRUE do do
      $u = \arg\max_{v \in V \setminus S} \delta_v$
      $\delta_u = \sigma(S \cup \{u\}) - \sigma(S)$
      if $\delta_u \geq max_{v \in V \setminus (S \cup \{u\})} \delta_v$ then
        $S = S \cup \{u\}$
        break
      end if
    end while
  end for
  Output $S$: Set of initial active vertices
```

than the original greedy method, it cannot compete with CELF [LKG07] method, since in their method only in the first iteration the spread for all the nodes are calculated and from the second iteration onward, due to submodularity only the spread for a few of the nodes are recalculated. Hence, they have combined their new Greedy method with CELF, where in the first iteration they use the sample graphs to calculate the spread and in the following iterations CELF is used. The second class of algorithms they have proposed are a set of heuristics for finding the seed set that maximizes the influence. Prior to their work, heuristics have been considered inferior to the Greedy method and while that is true for many of the heuristics, such as degree, random, distance and etc., their proposed heuristic called *DegreeDiscountIC* managed to achieve almost the same spread with the greedy method while outperforming the greedy methods by orders of magnitude. Details of the algorithm DegreeDiscountIC is explained in section 5.

Other notable heuristic method is of Saito and Kimura [KmS06]. They have investigated the scalability issue of IC model's simulations. They have argued that currently IC model requires many iterations of simulation and each iteration took a lot of processing time and power. They have proposed two natural special cases of the IC model that provide a good estimate of the spread without the need for running the simulations. They define the Shortest-Path Model(SPM) which is a special case of the IC model in which each node v has the chance to become active only at step $= d(S, v)$, where if S is the initial set, $d(S, v)$ is the distance of S to node v. This means that each node is activated only through the shortest paths from the initial active set. The other model they have proposed named SP1M, which is each node v has the chance to become active only at steps $t = d(S, v)$ and $t = d(S, v) + 1$. Through these two models they have proposed methods for calculating the spread of an initial active set S. They have proved that the result achieved using this method is also within a bound of the best solution. In their experiments, they have assigned a uniform probably to each edge with two different values $p = 0.1$ and $p = 0.01$. The experiments showed that the estimation of the spread for IC model increases as p increases while the processing times for the SPM and SP1M hardly changes.

Cheng et al. [CSH13] have addressed both the accuracy and scalability of the solutions using the greedy methods. They have pointed out the main bottleneck in methods based on Kempe et al. [KKT03] are in the Monte Carlo simulation steps. They have proven that the objective function is not entirely sub-modular due to the randomness in the simulations, and hence to compensate for that many Monte Carlo simulations have to be run in each step of the algorithm which greatly reduces the performance. They proposed a new method called *StaticGreedy* which computes a series of snapshots at the beginning of the algorithm and reuses those for every step of the greedy algorithm. A snapshot is a Graph $G' = (V, E')$, which is a subgraph of G where an edge $E(u, v)$ is remained with the probability $p(u, v)$ and deleted otherwise. Before the main steps of the algorithm many snapshots are produced and averaged to estimate the spread function. Two main differences between this method and the previous ones are:

- Monte Carlo simulations are conducted on static snapshots, which are sampled before the greedy process of selecting seed nodes
- The same set of snapshots are reused in every iteration to estimate the influence spread of the active set, which explains the meaning of ”static”

They have performed experiment and compared their methods with the normal Greedy methods. None of the other greedy methods are any match with Static-Greedy in terms of speed and scalability.

Zhuo et al. [ZZG13] propsed a new Greedy algorithm called *Upper Bound based Lazy Forward algorithm(UBLF)*. They have tried to bound the number of Monte Carlo simulation calls, similar to CELF algorithm [LKG07]. CELF increased the speed of

Algorithm 3 UBLF Algorithm [ZZG13]

```
Require: $G(V, E)$: Graph $G$, with Vertex set $V$ and Edge set $E$
Require: $PP$: Propagation probability matrix of $G$
Require: $k$: Initial Set Size
  S = $\emptyset$ and $\delta = (I - PP)^{-1}.1$
  for i = 1 to k do
      set $I(v) = 0$ for $v \in V \setminus S$
      while TRUE do do
          $u = \arg\max_{v \in V \setminus S} \delta_v$
          if $I(u) = 0$ then
              $\delta_u = \sigma(S \cup \{u\}) - \sigma(S)$
              $I(u) = 1$
          end if
          if $\delta_u \geq max_{v \in V \setminus (S \cup \{u\})} \delta_v$ then
              $S = S \cup \{u\}$
              break
          end if
      end while
  end for
  Output $S$: Set of initial active vertices
```

the original algorithm by a factor 700 on average case, but still on its first iteration it has to calculate the spread of every vertex. While that is reasonably fast and will give good result on small networks, it will not perform well on medium sized graphs let alone large scale graphs. To that extent, even running the simulations for all the vertices once is not feasible. Zhuo et al. [ZZG13] provided a mathematically proven upper bound for spread function of any initial set. In UBLF, instead of running simulations for each vertex in the first iteration, they find the upper bound which can be found in constant time. Having the initial values for all the nodes, they used the same technique as CELF. They started calculating the spread for vertices in order of their initial estimated spread, if the estimated spread was still bigger than the rest of nodes, the node was chosen and added to the seed set. Computation continues until desired number of initial seeds are achieved. Algorithm 3 describes UBLF. The matrix PP is constructed from the graph, where each value (u, v) in the matrix corresponds to the edge $e = (u, v)$; value of the edge that connects u to v. S consists of the seed set, the output of the algorithm i.e. nodes with highest influence propagation and I is the *identity matrix.* Before starting the iterative process, vector δ is calculated. In each iteration of the algorithm, vertices with the maximum δ are chosen, if Monte Carlo simulation is not yet run for them, the algorithm will run the simulation, and if the value of δ_u is still greater than the node with second highest δ then u is added to to the set S . The process continues for k iteration, where k is the desired size of the output. $\sigma(v)$ is the expected spread of v under the IC model. Experimental results shows that UBLF manages to reduce more than 95% Monte Carlo simulations needed and is at least 2 to 5 times faster than CELF method.

2.3 Learning Influence Probabilities

Another aspect of Influence Maximization problem is the assignment of probabilities to edges, which denotes the probability of a node influencing its neighbor through an edge. In both IC and LT models, different strategies are applied. In one method, fixed probabilities are assigned to all the edges. Other methods, includes assigning probabilities to edges chosen randomly from the set 0.1, 0.01, 0.001. Another strategy, typically referred to as the *Weighted Cascade* uses the inverse of the in-degree of each vertex as the influence probability. That means, an already activated node u can activate node v with probability of $1/d_v$, provided there is an edge between the nodes. Of all the methods discussed before, none of them truly capture the behavior of the network. It is obvious that in a social network people are affected by their friends differently. Saito et al. [SNK08] were the first to study how to predict the edge probabilities based on the past behavior of the users. Based on the propagation made in IC model, they introduced a probabilistic model and using *Expectation Maximization* they tried to find the best assignment of probabilities to edges.

Goyal et al. [GBL10] have also tackled the problem of forming the social graph whose edges have the probability of a user influencing the others based on a log of actions. Based on the *General Threshold* model that simultaneously generalizes the Linear Threshold and Independent Cascade models and an action log of past behaviors of the users, formatted as $Actions(User, Action, Time)$ where each entry indicates, an action made by a user at a specific time. Graph $G = (V, E, T)$ is constructed based on the relationship of users in the graph, where T indicates the time a specific edge was added to the graph (i.e. a relation was formed between two users). Hence, the objective function is defined as $p : E \rightarrow [0, 1] * [0, 1]$ assigning both directions of each edge (u, v) in E the probabilities : $p_{v,u}$ and $p_{u,v}$. Using simple counting methods, they have scanned the data set, and estimated the edge probability $p_{v,u}$ based on the action of node v and the subsequent action made by u. If these two actions happened within a predefined time, the probability of v influencing u was increased. They have also proposed more advanced methods of estimating the probability $p_{v,u}$ which is outside of the scope of this thesis. In another paper, Goyal et al. [GBL11] have also tackled the problem of influence maximization directly. Instead of first learning the edge probabilities and then finding the optimum seed set, they have proposed a method to directly find the optimal seed set based on the action log.

3 Big Data

Information and data has always been gathered for analysis, improvements, and predictions in different branches of science. It has helped governments to better govern countries, it has helped companies to better understand their costumers, and it has helped researchers and scientists to solve problems. With introduction of World Wide Web[4] exchange and collection of data has become a lot simpler. Big firms and companies started to collect users' information, browsing behavior and other related data. Petabytes of data are being stored and exchanged everyday, and these data contain valuable information that is being used for solving a lot of problems using data mining and machine learning methods. Processing this amount of data, has never been an easy task and prior to introduction of *Big Data* technologies, this processing and analyzing has been always done using expensive server systems that are able to process millions to billions of bytes of data in a second. Google's revolutionary papers has drastically changed the field. Ghemawat et al. [GGL03] introduced *The Google's Distributed File System*; a file system built for storing petabytes of data. It is scalable, meaning more storage units can be easily added, and it is being deployed on commodity hardware, meaning it does not require any sophisticated and expensive piece of hardware to use. Dean et al. [DjG04] introduced *MapReduce*, a framework for processing large amount of data, again it is scalable and it can be deployed on commodity hardware. Although the source code of either of these two technologies were never released by Google, but based on the publications **Doug Cutting** created the open source project *Hadoop* which implements both MapReduce (Hadoop's MapReduce) and Google File System (Hadoop Distributed File System or HDFS for short). Since then, Hadoop has become Big Data standard and several new technologies and project have been added to it.

3.1 Hadoop

Hadoop consists of two major parts. *MapReduce* which is a parallel programming paradigm. It is designed to run on distributed systems with many commodity computers. It involves three main steps, namely *Map*, *Reduce* and *Shuffle*. The first two are defined by the user. Map operations are run in parallel and they process the data individually, data is divided into different portions and each one is processed by a mapper. The result of the mappers are send to a Reduce task(or several) where

[4]http://webfoundation.org/about/vision/history-of-the-web/

the results are combined. A shuffle operation is usually performed on the result of the Map tasks so that similar data are all send to the same node. Algorithm 4 is a simple algorithm for counting words in a series of documents. The popularity of

Algorithm 4 Word Count in Hadoop

```
function MAP(String name, String document)
    //name: document name
    //document: document contents
    for each word w in document do
        emit (w, 1)
    end for
end function

function REDUCE(String word, Iterator partialCounts)
    //word: a word
    //partialCounts: a list of aggregated partial counts
    sum = 0
    for  each pc in partialCounts do
        sum += ParseInt(pc)
        emit (word, sum)
    end for
end function
```

Hadoop's MapReduce framework rises from the fact that many complexities of the system is being handled automatically by the framework. In most typical use cases, the user only needs to provide the method for Mappers and Reducers. Framework will take care of compiling and distributing the code to all of the nodes, and running them in parallel. As previously mentioned it is fault tolerant, it means if during the processing a task fails, the framework will restart the task, and if a node encounters a problem Hadoop will move the computation to another node. It is scalable, which means the program will try to utilize as many nodes and resources as possible based on the size of the input data and amount of available resources, and programmers do not need to concern themselves with that while writing the code. There are also many advanced options that users can do to further optimize the program, but they are generally not required for simple tasks. The other great feature of MapReduce and Hadoop is that it is available in many different languages; Java, Python, C++, Scala and many more. Figure 1 shows how the word count program will be executed on a cluster.

Newer versions of Hadoop and MapReduce employs more sophisticated resource manager and scheduler called *YARN* developed by Vavilapalli et al. [VVM13]. YARN essentially provides container for running different applications and it is not limited to MapReduce. Spark, Storm and even simple Java programs can be run on YARN. YARN is more or a less cluster manager program and overviews the resource and allocates resources to different jobs being submitted to it. It has become available as part of Hadoop 2.0 technology stack.

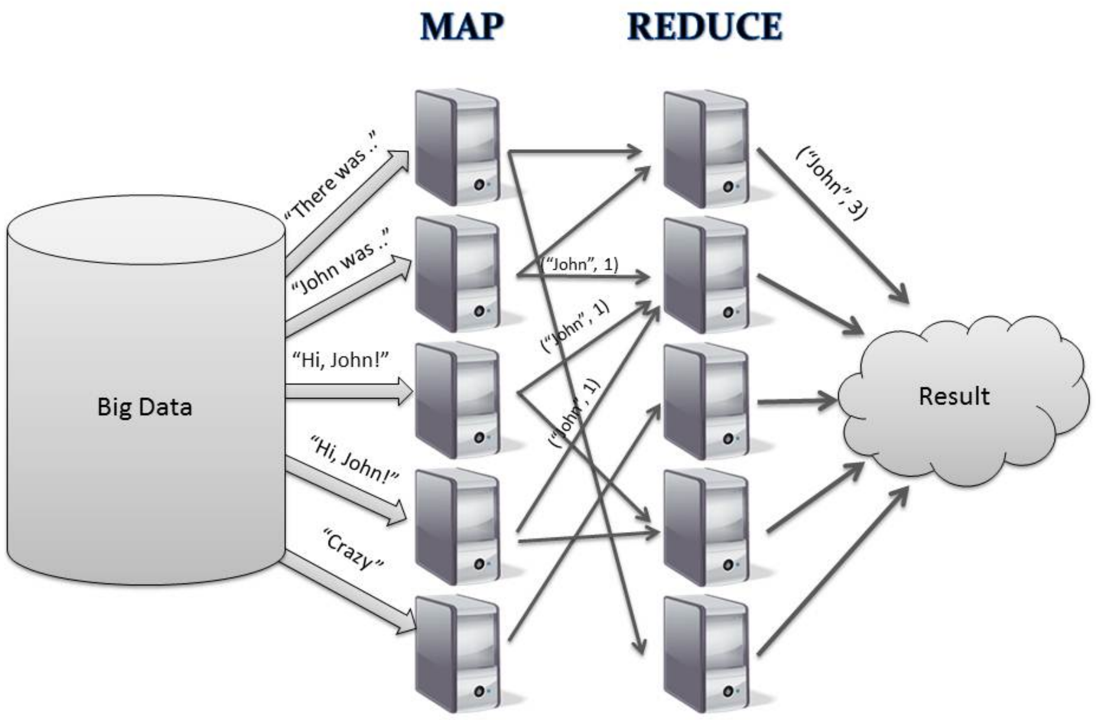

Figure 1: MapReduce paradigm [5]

Hadoop Distributed File System (HDFS) is a distributed file system, designed to run on commodity hardware. It provides high throughput access to application data and it is fault tolerant. Though initially designed for Hadoop's MapReduce framework, it has been used in many different applications and frameworks.

It is fault tolerant, meaning if a node in a cluster fails, the data is not destroyed.

[5]http://dme.rwth-aachen.de/en/research/projects/mapreduce
[6]http://hortonworks.com/hadoop/hdfs/

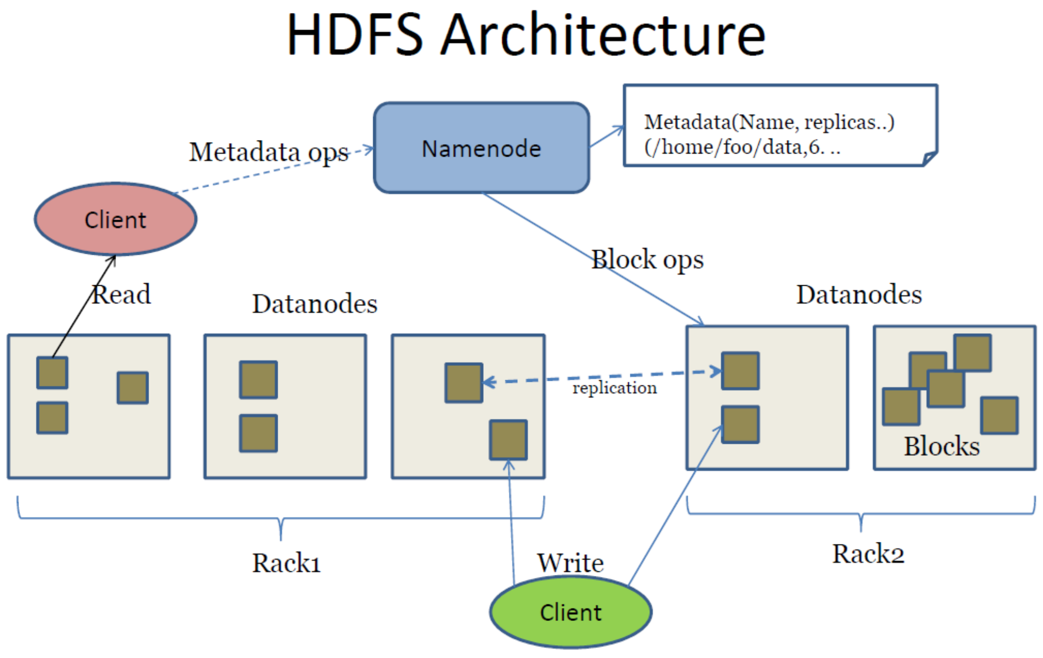

Figure 2: HDFS Architecture [6]

HDFS employs a feature called *Data Replication*, which indicates the number of times each object is being replicated. Figure 2 shows the high level architecture of HDFS. *NameNode* keeps track of where all the same instances of the data is stored and in case a node fails it knows where to look for it. HDFS emphasizes that moving computation is much cheaper than moving data, it is true since HDFS is designed to store big data sets, terabytes or even petabytes of data, hence in conjunction with MapReduce, instead of moving the data around, the computation is done on the node containing the data. It is designed for high throughput rather than low latency so it is not ideal for streaming applications, although several improvements are being made to further decrease the latency and making it suitable for real time streaming applications. Similar to MapReduce it is design to work across heterogeneous hardware and software platforms. This means that nodes in the cluster do not have to be identical and even running the same operating system. Hadoop runs on JVM, as a result of this it is operating system agnostic.

3.2 Spark

Apache Spark was originally developed by Zaharia et al. [ZCF10] in AMPLab at UC Berkeley. The main idea was to solve MapReduce's problems. In contrast to MapReduce's 2 stage program (Map and Reduce), Spark employs an iterative paradigm. It allows users to load data into memory and perform many operations on the data, this is ideal for a lot of Machine learning problems. The underlying frameworks work on entities called Resilient Distributed Datasets (RDDs) introduced by Zaharia et al. [ZCD12].

RDD is a read only and partitioned collection of data. It can be created from storage or other RDDs. There are two types operation on RDDs, *transformations* and *actions*. Transformations are operations that will create a new RDD from the existing one, for example *filter* is a transformation operation. The power of RDDs lie in their internal structure, they do not need to be materialized all the time. If an RDD is created using a transformation, it will hold information about how the transformation was made and only apply them to the data once the user requests that through an action or caching of the data set. Actions are operations that will materialize the data set. Users can also ask the RDD to be stored in memory if they know they are going to use it many times. These characteristics of RDD makes them the perfect tool for performing iterative applications. Many machine learning algorithms require iterative calls to the same data set. This is one of MapReduce's deficiencies since every algorithm and job has to be written in terms of Map and Reduces. Spark with the help of RDDs allows many of the well known machine learning algorithms to be implemented easily. With success of Spark, it has become part of the Hadoop framework. Recent versions can now run on top of Yarn and they have integration with HDFS and other Hadoop components.

A typical Spark program consists of a driver program that launches multiple workers. Worker programs are run on data nodes. Upon execution of a program, Spark will try to send the computation to where the data is, hence minimizing the network traffic. Spark tracks where the RDDs and their partitions are. Figure 3 shows a the general overview of how a Spark application is executed.

Users can control RDDs behavior by persisting it to memory or disk. Tests performed on Spark showed that it is 100 times faster than MapReduce and it is able to run applications that are not even possible to run using MapReduce. Spark is fault tolerant by means of RDDs. Driver always keeps track of the changes made to the RDDs, and they can be recreated from previous stages. So if a node con-

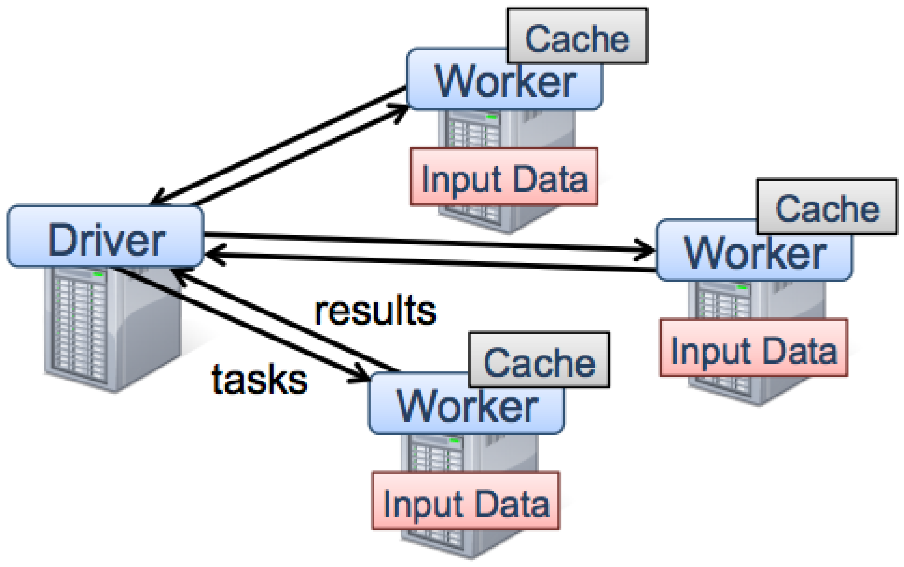

Figure 3: Spark Program, Zahaeria et al. [ZCF10]

taining some partitions of RDDs is lost, the partitions can be recreated using these information.

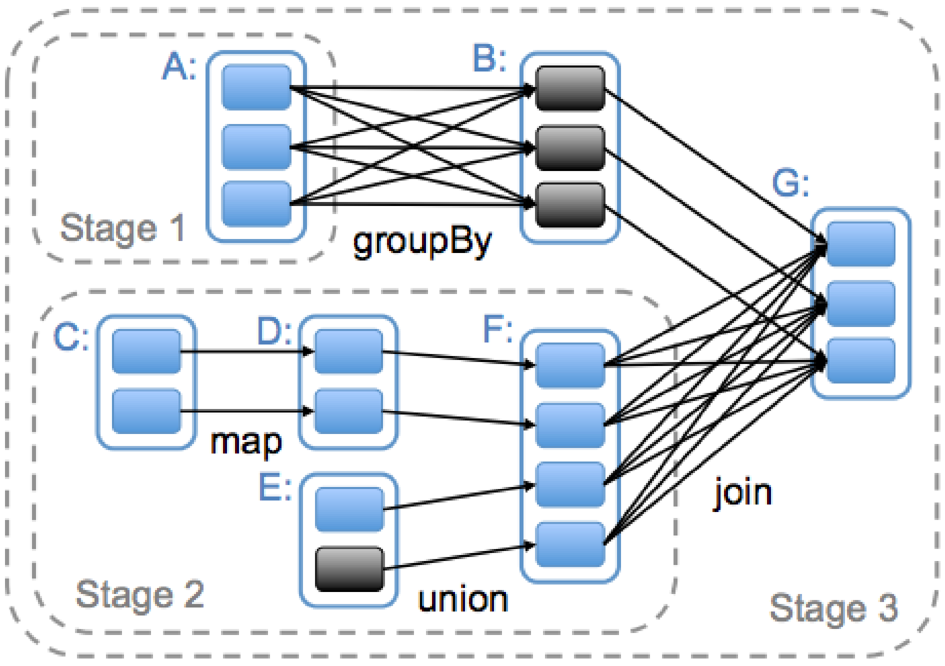

Figure 4: Spark stages and RDDs Zahaeria et al. [ZCD12]

Figure 4 shows how Spark computes job stages. Boxes with solid outlines are RDDs(A, B, ..., G). Partitions are shaded rectangles, in black if they are cached. To run an action on RDD G, the scheduler builds stages at wide dependencies and pipelines narrow transformations inside each stage. In this case, stage 1 does not need to run since B is cached, so we run 2 and then 3. If any of the stages fail, the computation will restart from the last healthy RDD. Also if a node containing some specific partitions of an RDD fails, the RDD will be transferred to another node and the computation will be restarted.

4 Large Scale Graph Processing Framework

Large data sets come in many different formats. They are not limited to documents, texts, or key value pairs. One of the more common representation of data is *graph*. Not only they contain information about individual entities, but graphs also store information about the relationship between these entities. They have been used in many different fields of science to represent data. The most prominent example of a graph data set is the *World Wide Web*, which is a collection of digital pages on Internet and each page may have links to several other pages.

In this section, two of the widely used graph frameworks designed specifically for handling large scale graphs, are discussed in detail. Both of these frameworks have been used to implement the methods discussed in section 6. Interestingly, one common computation model used in many of these graph frameworks is based on a model introduced by Valiant [Val90] in 1990, called *Bulk Synchronous Parallel* (BSP). A BSP computation model consists of three components:

- A component capable of local processing
- A network for transferring messages
- A hardware which synchronizes the components

BSP programs are run in a series of global steps, called *supersteps*. In each superstep, first, every node runs the task assigned to them locally, and then they send messages over the network to other components. Before the next superstep begins, a barrier is set in place that synchronizes all of the nodes and makes sure all the computation and network transfers of the current superstep are finished. Malewicz et al. [MAB10] developed a system called *Pregel* at Google, which is based on the BSP model and is designed for implementing graph related algorithms. Pregel works by using vertices of the graphs as computation nodes. In each superstep, all vertices perform local computations. During this phase they only have access to the data internal to the specific vertex which includes the vertex properties and the messages that were sent to the vertex in the previous superstep. After the computation step is done, all vertices may forward a series of messages to other vertices. The messages are usually send through edges, i.e. only vertices connected by an edge communicate with each other, but that is not necessarily the case all the time. Pregel also takes care of the synchronization barrier and makes sure a new superstep does not begin until the current superstep is finished on all of the vertices.

4.1 Giraph

Apache Giraph is an iterative graph processing framework, built on top of Apache Hadoop. It is the open source version of Pregel and is now being actively developed by several companies. It is being used in many real case scenarios. For example, Facebook is using it to analyze the graphs formed by users and their connection with their friends[7]. In a typical Giraph program, input graph is usually stored and read from the HDFS. Once the program starts, Giraph will read the graph and partition the vertices across several nodes. Users can decide how to partition the vertices based on the type of problems they are trying to solve. Giraph by default uses a *Hash Partitioner* to partition the vertices. Generally it is a good idea to provide a custom partitioner that works best with the specific problem that is being solved. For example in most of the Giraph programs, in the communication phase, messages are sent along the edges of the graph, so if vertices that are connected to each other are partitioned to the same computation node, the network overhead will be reduced significantly. Figure 5 shows how computation is done in Giraph.

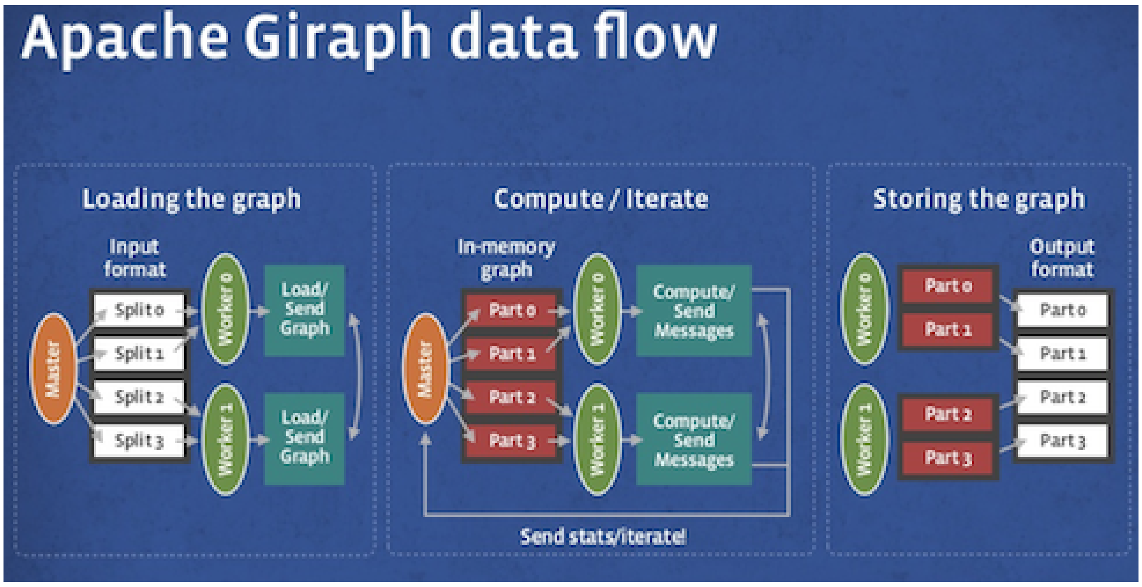

Figure 5: Giraph Work Flow [8]

Each Giraph cluster has a master node and one or more worker nodes. Master node has the responsibility of reading the graph and partitioning it based on the given partitioner, coordinate the job and the supersteps. Master node is also responsible

[7]https://code.facebook.com/posts/274771932683700/large-scale-graph-partitioning-with-apache-giraph/

[8]http://giraph.apache.org/

for the synchronization of vertices and making sure that a new superstep does not begin until all the vertices have finished their computation. Each worker node contains several vertices and run the actual program, and coordinate with the master and other nodes through a set of messages send along the edges or directly to other vertices. Vertices in a Giraph program can be in either of the two states; *Active* or *Not Active*. Generally the program continues while all the vertices are active. Vertices themselves are responsible for setting their status after they have finished computing. If a vertex receives a message from other vertices, its state will automatically change to Active. The figure 6 shows supersteps involved with computing the single source shortest path for a simple graph.

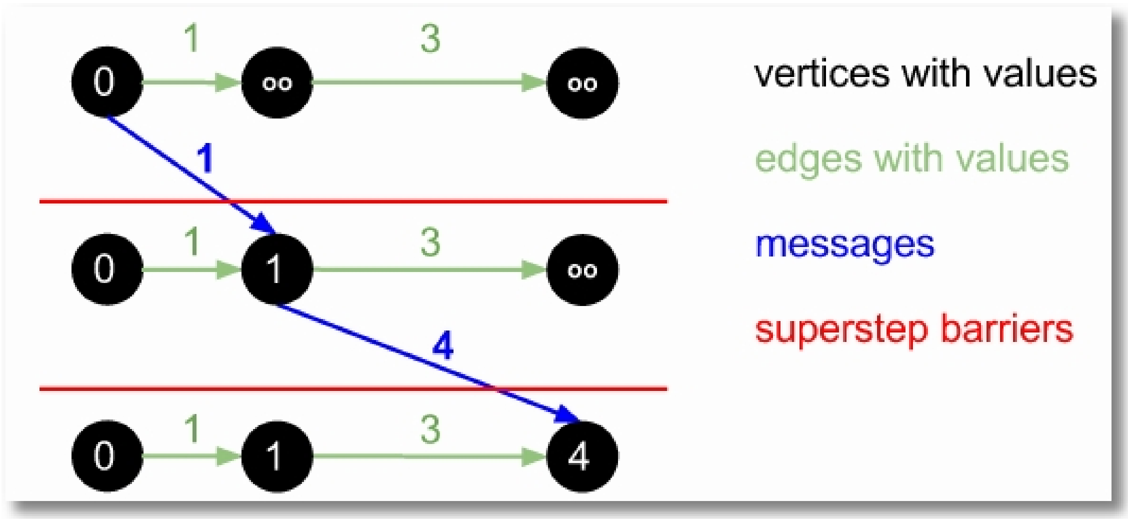

Figure 6: Single Source Shortest Path [9]

The input graph consists of 3 vertices, and 2 edges. The goal of the program is to find the shortest path from every vertex to the left most vertex. Red lines separate supersteps and in the first superstep, the value of all the vertices except for the source is assigned to infinity. At the end of each superstep, distance upper bounds (vertex value + smallest edge value) is send to the destination of the edges and vertices's values are set to the smallest message they have received. The execution stops when all the vertices have the distance from the source vertex.

Although Giraph has many advantages over MapReduce, it is still has some limitations. In one Giraph job many iterations can be done, but if graphs should go through several transformations, that would require multiple Giraph jobs, which means that the data has be written to HDFS after each job. This adds a lot of IO

[9]http://giraph.apache.org/intro.html

overhead. A lot of the Machine learning and graph problems are iterative in nature, which will cause Giraph to have severe performance issues in some of these cases.

4.2 GraphX

GraphX is a light API written on top of Spark. It uses the powerful RDD abstraction as the basis of its computation, by storing vertices and edges in separate RDDs. Similar to Spark, it works well for iterative algorithms and it comes packaged with many of the well known graph algorithms such as, PageRank, Label Propagation, Connected Components and others. Several important components of GraphX are:

- VertexRDD: an RDD object consists of a vertex id and a data type. It carries some extra features in comparison to the traditional RDDs, such as ensuring vertex ids are unique and optimized joins among different vertex groups. It offers several methods for mapping values, filtering vertices(while preserving their id) and finding difference of two vertex groups.

- EdgeRDD: similar to VertexRDD, it is a subclass of RDD, which organizes the edges into blocks based on some predefined partitioning strategies. It supports three extra methods; *mapValues* which maps the values while preserving the structure, *reverse* which reverse the edge directions while preserving the attributes, and *innerJoin* which joins two EdgeRDD objects, using the source and destination ids as keys.

The key feature of GraphX comes from the way vertices and edges are stored. GraphX employs different partitioning strategies for storing vertices and edges. While each one have their own advantages and disadvantages, the overall goal is to reduce the amount of communications along the edges by having all the vertices close to each other in one computation node to avoid network traffic. The details of GraphX and how it behaves under the hood are beyond the scope of this document. Xin et al. [XRG13] described GraphX in detail and GraphX website[10] includes more information about the internal architecture of the system. GraphX also provides a Pregel like API, that users should only provide the vertex program and the communication strategy. This is one of main advantages of GraphX over Giraph and other similar graph processing frameworks. A GraphX's graph is like any other object

[10]https://spark.apache.org/graphx/

in Spark environment, and all the transformations and actions available in Spark package can be applied to vertices and edges of the graph.

5 Big Data and Network Influence Maximization

The original greedy method proposed by Kempe et al. [KKT03] provided a reliable solution to the problem of maximizing influence in graphs. Several other researchers have worked on finding algorithms that find better solutions in terms of accuracy, but most of the researches done in recent years were involved around improving the efficiency of the original method. CELF [LKG07], CELF++ by Goyal's [GLW11] and Zhuo et al. [ZZG13] methods have considerably increased the speed of the original Greedy method. Cheng et al. [CSH13] proposed the StaticGreedy method which combined with CELF further improved the speed of the greedy method. However, none of the proposed methods are scalable and they can only operate on small to medium sized graphs. The bottleneck for the greedy method lies in the function it is trying to optimize. As discussed in the previous sections, the spread function under the Independent Cascade (and Linear Threshold) model cannot be determined exactly due to the edge probabilities. Hence, Monte Carlo method is being used which consists of running the IC model many times (up to 10000 for accurate result) and then averaging over all the simulations.

Currently, there are companies and organizations that have graphs of millions to billions of nodes with even more edges, and none of the optimization methods discussed above can be applied to any of these graphs. Even on the most powerful clusters running the random process for one iteration takes a lot of time, essentially rendering these methods completely useless. The other group of methods discussed in the previous sections were heuristic methods. They can range from very simple methods such as finding the nodes with highest in or out degree to more sophisticated ones such as *Single Discount* and *Degree Discount* discussed by Chen et al. [CWY09] or SPM and SP1M proposed by Saito and Kimura [KmS06]. The advantages of heuristic methods are that they are very fast and they can scale to graphs of any sizes, however the accuracy of the solution is not as good as the greedy method.

In the rest of this section, first the literature on influence maximization for big data and then details of the methods implemented are discussed. Some of methods are fairly simple heuristics and some are based on the methods discussed in previous sections with tweaks that make it possible to implement them on large scale graphs. Changes have to be made to the original algorithms in order to implement the methods using graph frameworks built for processing large graphs. These frameworks are imposing certain set of restrictions that will make applying some classes of algorithms more difficult. Bulk Synchronous Processing is the base of all the large scale graph frameworks where they are using the method of *Think Like a Vertex*, which means each vertex is a virtual computation abstraction. Although this has many benefits but when it comes to classical IC and LT models, some difficulties will rise. In BSP model, usually algorithms that only need local data are implemented a lot simpler. Howevern, in influence maximization we need global access to the network to find out how many new nodes were affected by adding a node to the initial seed. The whole process is a randomized simulation that should be run hundreds or thousands of times to get a better and more accurate result. Large scale data processing usually suffers in this area, because initializing jobs given huge amount of data whether it is in graph format or others, takes some time and that is something that should be avoided. The main challenges when implementing the methods using BSP models are hence, finding ways to avoid the iterative and simulation like implementation and trying to generalize the whole simulation cycle into one BSP job.

5.1 Monte Carlo Simulation

On the baseline of most of the methods, lies the *Monte Carlo Simulations*. It represent the Independent Cascade propagation process. Given an initial seed, it will calculate the expected spread, by running the process thousands of times and report the average spread for the initial active set. The distributed method differs from the original implementation in the way the propagation is spread through vertices. In single machine computation, for each vertex in the graph, if the vertex is active it will try to influence all of its neighbors only once and the process continues until no new vertices become active. In distributed implementation, all of the vertices will try to activate their neighbors at once.

Each vertex carries information about its state. As can be seen from the algorithm, in each iteration of the simulation, every active vertex will try to activate its neighbors, and then change its state to *tried* so it doesn't try to activate again in the next iterations. If a vertex is activated, its state is changed to *active* and it is added to the active set. The process continues until all of the vertices in active set A have tried activating the neighbors and no new vertex is added. The final result is the average of all the spreads for every simulation iteration. This is a time consuming procedure because of the number of iterations in the simulation. But this algorithm is not being used in any of the main methods for finding the initial seeds and it is rather for estimating the spread of a seed. I am only using the procedure to have

Algorithm 5 Monte Carlo Simulation for Independent Cascade Model

```
Require: $G(V, E)$: Graph $G$, with Vertex set $V$ and Edge set $E$
Require: $S$: initial active vertices
Require: $R$: # of simulation iterations
  $s_i = 0$ for all $0 < i < R$
  for $i = 1$ to $R$ do
      $A = S$
      for each $v \in A$ do
          if $v_{state} \neq tried$ then
              for for each $u \in v_{neighbors}$ do
                  try to activate $u$
                  if $u_{state} = active$ then
                      $A = A \cup \{u\}$
                  end if
              end for
              $v_{state} = tried$
          end if
      end for
      $s_i = |A|$
  end for
  $sp = avg(s)$
  Output $sp$: Spread of the initial set $S$ through out graph $G$
```

a baseline of how good an initial seed is and running it even with few number of iterations should give an acceptable estimate.

5.2 Single Cycle Influence Maximization

Based on Kempe et al. [KKT03] Independent Cascade method, *Single Cycle Influence Maximization* is a novel method, which is a simpler but a less accurate version of the original algorithm. Implemented using the BSP paradigm, it finds the vertices with maximum influence spread in one iteration, by simulating the Independent Cascade process for all the vertices at once.

Each vertex contains information about the number of nodes it has affected, and the list of nodes it is affected by. The algorithm is written as point of view of a vertex, that means every vertex executes these steps until the master decides the

Algorithm 6 Single Cycle Influence Maximization

```
if superstep == 0 then
    add your own id to list of influencedBy an increase vertex value by one
    try to activate all neighbors by sending its own id
else
    receive message from every one (message is the id of the vertex)
    for each message received do
        if it is already in your influencedby list then
            vote to halt
        else
            activate all neighbors by using the message
            add to influecedBy list
            notify source vertex
            vote to halt
        end if
    end for
end if
```

computation has ended. The program ends either when all of the vertices have changed their state to not active or a limit for number of supersteps is reached. Each vertex sends its own id at the first superstep to all of its neighbors, the reason the vertex id is being sent and not just a simple boolean value is that each vertex has to know who has activated it and store their id and send a message back to them. In the first iteration, when vertices activate their neighbors, any successful activation will be carried on with the affected vertex. For example, if vertex v_2 was activated by vertex v_1 and now it is trying to activate vertex v_3 it will send both its own id v_2 and id of the vertex that activated it v_1. Subsequently when v_3 tries to activate its neighbors it will send its own id as well as v_1 and v_2. When a vertex is activated it will send a message back to the source of the activation, informing it that it has been activated and the source can now increment its total spread by 1. At the end of the procedure, each vertex has a value indicating its expected spread. The top k vertices are reported as the best initial seed. While this method is fast, it has some draw backs:

- Activation of a new node is done through only one single random process, based on edge weight.

- All nodes are performing the activation process independent of other nodes. Depending on network structure this could have some effects on the final result
- Since each vertex is sending it's own id and id of vertices that it has been affected by, it might create network traffic.

Point one indicates that the result may vary between different runs since we are only running the simulation once. In the original greedy algorithm, in each iteration one vertex is added to the current initial set and the spread is recalculated, then the vertex that increases the spread the most is chosen to be added next. But here all of the vertices are operating in silos, disregarding the network effect. Consider an example where two neighboring vertices, individually has the highest spread among all vertices. Algorithm 6 will report both of them in the final result, but in reality having two neighboring nodes is not a very good idea, since nodes affected by one are usually affected by the neighbor as well. The last point about network traffic, in worst case scenarios this might be extremely inefficient and in some cases bring the whole cluster down. However, it is hardly the case in real big graphs, since first of all most of the big graphs are extremely sparse and the propagation probabilities are so small that a vertex id hardly has to travel more than a few nodes. Testing on larger graphs however revealed another flaw of this method. Since vertices have to store the list of other vertices that have activated them in memory, in big graphs that usually leads to the program crashing because of memory being full. In section 6 this issue is explained.

5.3 Token Based Implementation

Another novel method based on the Single Cycle algorithm, where the simulation aspect of the Independent Cascade model is taken into account. In this algorithm, each vertex, instead of trying to activate its neighbors only once, it will try many times. Vertices will keep many tokens with unique ids. Each token can be tracked back to the vertex itself. In each superstep a vertex will try to activate its neighbors once with every token. If a node is activated using a token, it will notify the original source vertex, so that the vertex can increment its internal counter for that specific token. Once no new nodes are activated, the program ends and all the vertices will report the average of token counters as their expected spread. Top k nodes are then selected as the initial seed. The number of tokens for each vertex determines the accuracy of the algorithm. While having thousands of tokens will increase the

Algorithm 7 Token Based Influence Maximization

```
if superstep == 0 then
    add your own id to list of influencedBy an increase all token counter values by one
    for each token do
        activate all neighbors using the token
    end for
else
    receive message from every one (message is the id of the vertex)
    for each message received do
        extract token out of the message
        if token is already in your influencedby list then
            vote to halt
        else
            activate all neighbors by using the token
            add to influecedBy token list
            notify source vertex
            vote to halt
        end if
    end for
end if
```

accuracy of the final spread, in real graphs this is not practical because that means each vertex has to carry a lot of extra information. Choosing the best number of tokens hence is a trade off between a better solution and a faster algorithm. Similar to previous method, depending on the structure of the graph, this method may produce a lot of network overhead, due to the amount of communication that is being done between nodes. Also it will require a bigger storage since each vertex has to store the spread for many different simulations. So choosing the number of the simulation, not only slows down the process, but it will also increases the risk of a crash in the system, due to size and network overhead. Similar to Single Cycle method, Token Based method has also failed to provide the result on very large graphs.

5.4 Edge Sampling

Edge Sampling is another novel method inspired by Chen et al. [CWY09] and Cheng et al. [CSH13]. In Edge Sampling the Independent Cascade process is tackled differently. This method is still using one or a few Monte Carlo Simulations, so in theory it is not as accurate as the Greedy methods, but it can achieve very good results and the number of Simulations can be easily extended as the procedure is very fast. In each iteration of the simulation, instead of having each vertex try to activate its neighbors based on edge probability, a subgraph G' is being sampled from the original graph G based on the edge probabilities. Each edge e will remain in the subgraph G' with probability of e_w, where e_w is the weight of the edge. G' is now a graph with many unconnected components, in this new graph we can define each vertex v's estimated spread that belong to connected component C as $v_{spread} = |C|$. The reason for that is since the edges are already sampled by their probabilities, that means any existing edge will propagate the message. Here we can report the top k vertices as the solution.

As discussed in Single Cycle Influence Maximization section, one draw back of blindly choosing the vertices with the highest estimated spread was that, vertices that are neighbors or are in the same connected components, will not contribute much to the final spread since having both in the initial seed is redundant. However, in this method since we know each vertex belongs to which connected components, instead of reporting the top k vertices, we can report one vertex from the top k connected components, that is the connected components with highest number of vertices in them. The algorithm can also decrease the noise of the randomness by using many sampled subgraphs and averaging the best vertices from all of the samples. The algorithm's definition is presented in Algorithm 8. As with other methods, some modifications have to be made to make this method more suitable in parallel frameworks. The connected components can be efficiently calculated in parallel using BSP model. It is already implemented as part of the GraphX library. To further optimize the algorithm, changes have been made to the connected component algorithm of GraphX. Since for each vertex, we only need the size of its corresponding connected component, we can modify the original algorithm to instead of reporting the connected component id, it will report the size of the connected component. The result of the new connected component algorithm would then be the list of vertices, each vertex carrying its id and corresponding connected components size. In this method, the connected component size is equal to the expected spread for

Algorithm 8 Edge Sampling

```
Require: $G(V, E)$: Graph $G$, with Vertex set V and Edge set $E$
Require: $R$: # of iterations
Require: $k$: Initial Set Size
  $v_{total} = 0$ for all $v \in G$
  for $i = 1$ to $R$ do
      sample $G'$ from $G$ based on edge probabilities
      $C$ = connected components if $G'$
      for every vertex $v \in G'$ do
          $v_{total} = v_{total} + |C_v|$ where $v \in C_v$
      end for
  end for
  for every vertex $v \in G$ do
      $v_{spread} = v_{total}/R$
  end for
  $S$ = top $k$ vertices based on $v_{spread}$
  Output $S$: Set of initial active vertices
```

each vertex.

The algorithm starts by initializing the list of all the vertices. Every vertex's expected spread is 0 at the beginning. In each iteration of the simulation, the sizes are updated based on the connected component size. The update can be done simply by doing a join. Vertex joins are very efficient in GraphX, and can be done very quickly. Increasing the number of simulations will increase the accuracy, but it also slows down the process.

5.5 Degree Discount Heuristic

In previous section, Chen et al. [CWY09] heuristic for choosing the best vertices for initial seed was discussed. Their method has several advantages that make it easy to implement, and it is very fast and scalable. This method does not require a simulation as it is a simple heuristic based on the degree of each vertex. According to Chen et al. [CWY09] it yields an extremely good result which is in par with greedy methods. Compared to other heuristic methods that will be discussed in the next section, this method is more difficult to implement in a BSP model. The number of iterations the algorithm runs depends on the seed size. Basically, after initializing

Algorithm 9 Degree Discount [CWY09]

```
Require: $G(V, E)$: Graph $G$, with Vertex set $V$ and Edge set $E$
Require: $k$: Initial Set Size
  $S = \emptyset$
  for each vertex $v$ do
      compute its degree $d_v$
      $dd_v = d_v$
      $t_v = 0$
  end for
  for $i = 1$ to $k$ do
      $u = \arg\max_v \{dd_v | v \in V\ S\}$
      $S = S \cup \{u\}$
      for each neighbor $v$ of $u$ and $v \in V\ S$ do
          $t_v = t_v + 1$
          $dd_v = d_v - 2t_v - (d_v - t_v)t_v p)$
      end for
  end for
  Output $S$: Set of initial active vertices
```

all the vertices, in each iteration the best vertex is chosen and added to the seed set. After that, the values for the neighbors of the selected vertex will be changed based on the formula presented in the algorithm 9. This means for a seed set of size k, the algorithm runs for k iterations. Although this method looks promising, the underlying strategy is not very feasible to implement in BSP. As a result, it will not scale to large graphs, at least using GraphX and Giraph frameworks.

5.6 Other Heuristics

Degree Discount heuristic discussed in previous section, is one of the many heuristics methods that have been used in influence maximization. As explained earlier, although heuristic methods are inferior to more complex methods such as the greedy algorithm in terms of quality of the solution, they have one advantage over them. These classes of solutions are highly scalable and can be easily applied to very large graphs. Below is a list of some of the heuristics methods that have been used for finding the initial set of active nodes in influence maximization problem.

Random: k random vertices are selected as the solution. This is used as a baseline

for other methods to see if they perform better than just choosing nodes at random.
Degree: Degree of a vertex is defined as the number of edges connected to a vertex. In this method, the vertices with the highest degree are chosen as the initial seed.
Single Discount: Improved version of Degree heuristic. After a vertex is chosen as the seed, it is removed from the graph, that means all of its neighbors' degree are reduced by one. This yields a better result than degree heuristic, since it lowers the chance of neighboring vertices to be selected as the initial seed.
Page Rank: Based on the popular Page Rank algorithm by Page et al. [PBM98], the algorithm was originally used in Google's search engine, to rank websites based on their importance.

6 Experiments

Two important aspects of solutions for influence maximization problem for large scale graphs are:

- **Quality** which is measured by the spread of the solution found(initial active set of vertices) under Independent Cascade model
- **Scalability** which is measured by the running time of the program.

In this section the details of the experiments, including the software and hardware platform, data sets and methods are explained in detail.

6.1 Experiment Setup

Experiments are run on two different computation platforms. For the smaller data sets, to test the accuracy and speed of the methods the experiments are run on a single machine. The bigger data set is run on an EMR cluster. The details are below.

- Single node setup :
  A Macbook Pro with 2,4 GHz Intel Core i5 processor and 8 GB of memory.
- Cluster setup:
  5 m3.xlarge Amazon EMR nodes. Each node has 4 CPU cores, 15 GB of memory and 80 GB of SSD storage.

Except for the Greedy CELF method, which is implemented in a single threaded Java application, the rest of methods and the simulations are implemented using Apache Spark's GraphX version 1.3.0 in Scala or Apache Hadoop's Giraph version 2.0 in Java.

6.2 Data set

Below is an overview of the data sets used in the experiments. All of the data sets are provided in an edge list format, where, in each line there are two vertex ids separated by a tab character. The two vertices are end points of an edge.

6.2.1 ArXiv collaboration network data set

A data set from the e-print arXiv[11] is chosen for testing the algorithms on smaller scale graphs. The data set contains academic collaboration for scientific research papers. In this data set, vertices represent researchers and edges define a collaboration between two researchers on writing a scientific paper. The data set has been used by several other researchers in the field of influence maximization and can provide a good comparison for the quality of the methods. The *High Energy Physics-Theory* graph, which contains papers from 1991 to 2003, with 15,233 nodes and 58,981 edges is used as a base comparison of the methods proposed in section 5 and state of the art greedy method. The data set in this format was made publicly available by Chen et al. [CWY09] and can be found here: http://research.microsoft.com/en-us/people/weic/graphdata.zip.

6.2.2 LiveJournal Social Network

LiveJournal is a free online community with almost 10 million members, with many of them being active. Similar to other types of social networks, users can form friendship with other members. It consists of 4,847,571 vertices and 68,993,773 edges. This is considerably larger than a lot of data sets usually used in influence maximization research. The data set is publicly available at https://snap.stanford.edu/data/soc-LiveJournal1.html.

6.3 Methods & Implementation details

List of influence maximization algorithms used in the experiments

- **Single Cycle**: Based on *Single Cycle Influence Maximization* algorithm. This method is implemented using Apache Giraph for ease of implementation.
- **Multi-attempt Single Cycle**: Based on *Token Based Influence Maximization* algorithm. Similar to single cycle, this method is also implemented in Apache Giraph
- **PageRank**: Based on popular Page rank algorithm developed by Google. Implemented using Spark's Graphx.

[11]http://arxiv.org/

- **Degree**: Simple method, that chooses nodes with highest degree. Implemented using Spark's Graphx.
- **Degree Discount**: Based on Degree Discount heuristic, discussed in previous section. This method is also implemented using Spark's Graphx.
- **Edge Sampling**: Based on Edge Sampling algorithm, this method is implemented using Spark's Graphx.
- **Random**: Choose initial seed randomly. This is used as base comparison.

Some of the methods proved to be easier to implement using certain framework. There are subtle differences between, Graphx's pregel API and Giraph's pregel API. Graphx's pregel requires the user to supply three methods :

- Vertex program: Responsible for modifying the value of vertices. It receives the messages for each vertex and perform internal calculations based on the messages and vertex properties. The return type, is the same as the vertex data type and cannot be changed.
- Send message: Inputs an *EdgeTriplet* data type, which contains the source, and destination vertices along with their attributes, and edge data type. It returns an *Iterator* which specifies the message and the destination the message should be sent to.
- Merge messages: Merge messages intended for a destination vertex. It receives two messages of the same type and return the combined message that should have the same data type as the input messages.

Although, these three methods work along side each other, they are separated and do not have access to the internal values of each other. This makes it very difficult and not efficient to implement the two methods: Single Cycle and Token Based. These two algorithms, are designed to act upon the message they receive immediately and based on the value of each message, they have to decide either to forward a message to neighboring vertices or not. This is unfortunately not possible in Graphx.

Giraph's pregel API is designed differently. It consists of the following methods, some of which are optional and do not need to be supplied by the user:

- compute : The most important method. It has to be supplied by the user. The input to the method, is the vertex and all messages intended for that vertex. User can modify the value of vertex, send messages to other vertices, and etc.

- initialize: Called once at the beginning of the program. Generally used for defining any aggregators or other global operations that will be used during the run time of the program.
- preSuperStep: Called before every super step. It is not mandatory, but it is generally used to update global values before the super step begins.
- postSuperStep: Similar to previous method, with the difference that it is run after super step has ended.
- sendMessage: Method for sending messages to other vertices. It requires the id of the destination vertex and the intended message. This method is generally not overridden, and it is simply intended to send a message to another vertex. It can be called many times from the compute method. Giraph takes care of synchronization process. Generally all the messages are sent, once all the compute methods for all the vertices are executed.

Giraph provides more freedom to run algorithms in pregel framework, as it leaves the responsibility of sending messages to the user rather than enforcing it to be send in every iteration. This makes, Giraph ideal for implementing the first two methods, because the decision to either send or not send a message depends on the content of the message itself.

7 Results and Discussions

In this section, the results of running the algorithm discussed in section 5 are shown. The spread and running time of different methods on different real graphs are reported. In the last part of the section, several random graphs having common characteristics found in real graphs are used to further analyze and explain the advantages and disadvantages of each of these methods.

7.1 ArXiv collaboration network data set

In this section the result of running different methods on collaboration network data set are explained. To calculate the spread for each method, multiple simulations needed to be run. In these set of experiments number of simulations is set to 100. While increasing the number will make the result more accurate, it will also increase the running time. Experiments have been run with different number of simulations and the results show that the difference between number of simulations is not very significant. Figure 7 shows the result of running search methods on

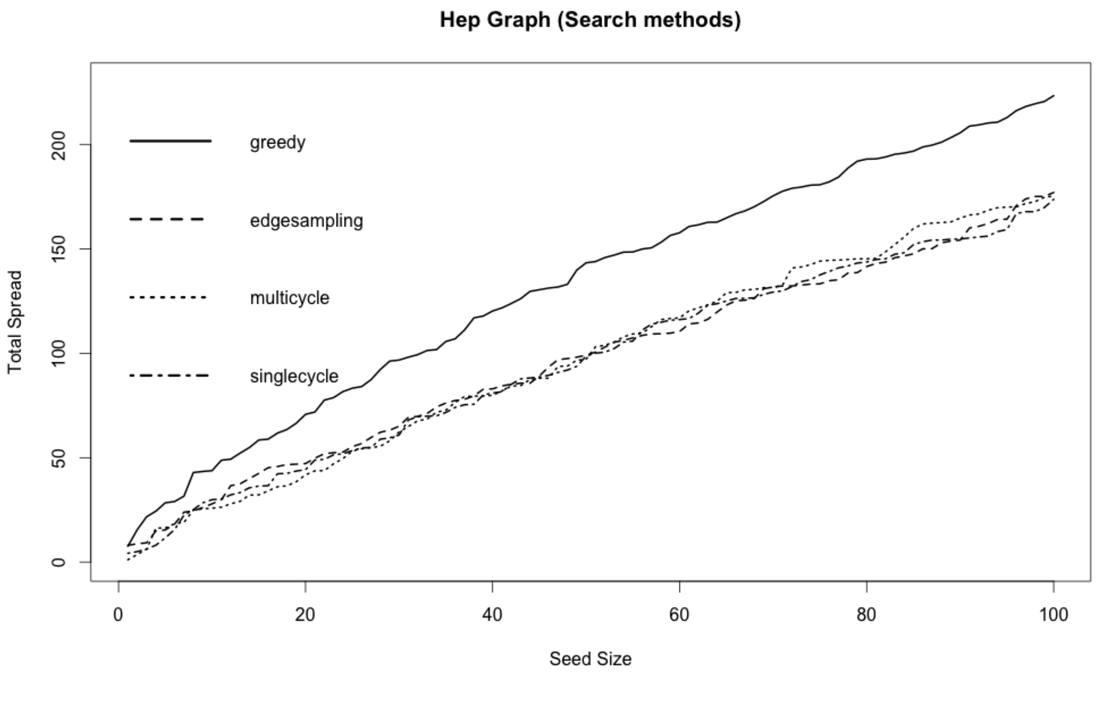

Figure 7: Search methods

collaboration graph. Clearly the greedy method is much better than all of the other methods. As discussed in previous chapters, the underlying approach of the other three methods are similar and hence their spread is more or the less the same. Both Edge sampling's number of iterations and multicycle's number of tokens are set to 100, and their results are a little better than single cycle. However, as explained earlier both the noise from calculating the spread and the algorithms themselves make it difficult to clearly distinguish their differences. Figure 8 shows the spread

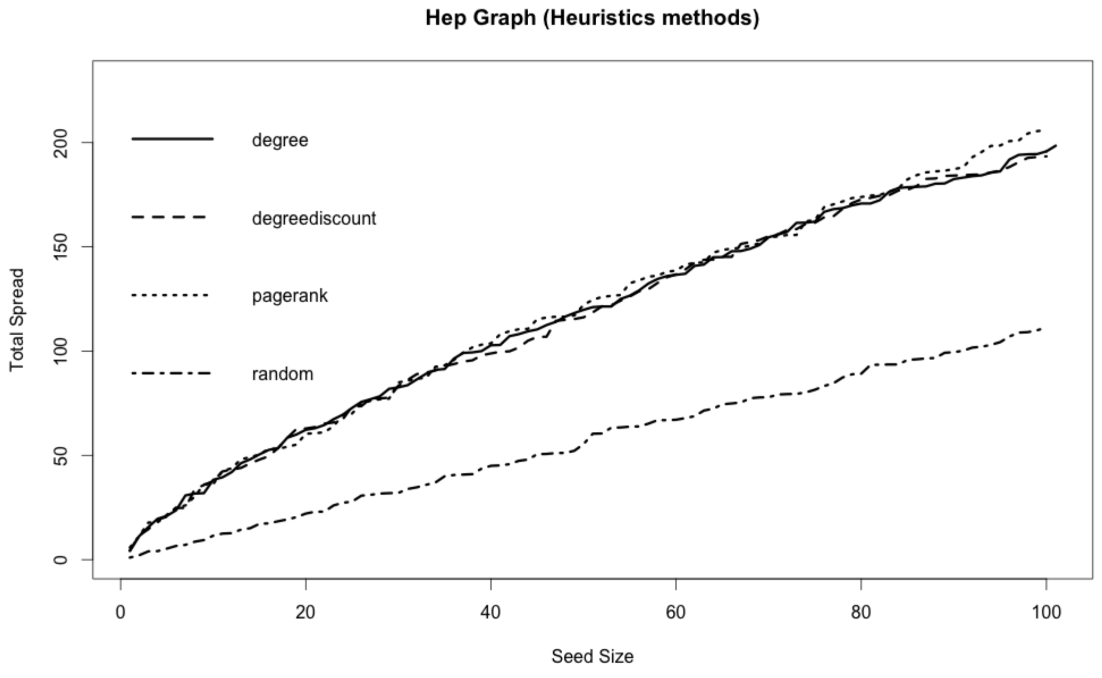

Figure 8: Heuristic methods

of the heuristics methods on collaboration network graph. All heuristics methods except random produce similar results. Page rank is slightly better than degree and degree discount method, mostly because when calculating the importance of a vertex, it doesn't just consider its immediate neighbors. Comparing heuristics with search methods shows that greedy methods is the best among all of the methods and heuristics methods are performing better than edge sampling, single and multi cycle methods.

Table 1 shows the running time all of the methods except for the greedy method. As explained in previous chapters, running time of all of the methods specified in the table is independent of seed size, which is their biggest advantage. Figure 9 shows the running time of the greedy method based on seed size. To make the comparison easier, running time of other methods are also shown in the figure. To

method	time (s)
random	0.84
degree	1.01
degreediscount	1.156
pagerank	2.338
edgesampling	7.73
singlecycle	45.2
multicycle	50.34

Table 1: Running time

find a seed set of size 100 on the collaboration network data set after applying the CELF optimization, it takes almost 2 hours, and this graph is much smaller than a typical big data graph. Being an NP-hard problem, the algorithm will not scale linearly, this means for data sets that are 100 times bigger than collaboration network, it might take thousands of time more to run the algorithm

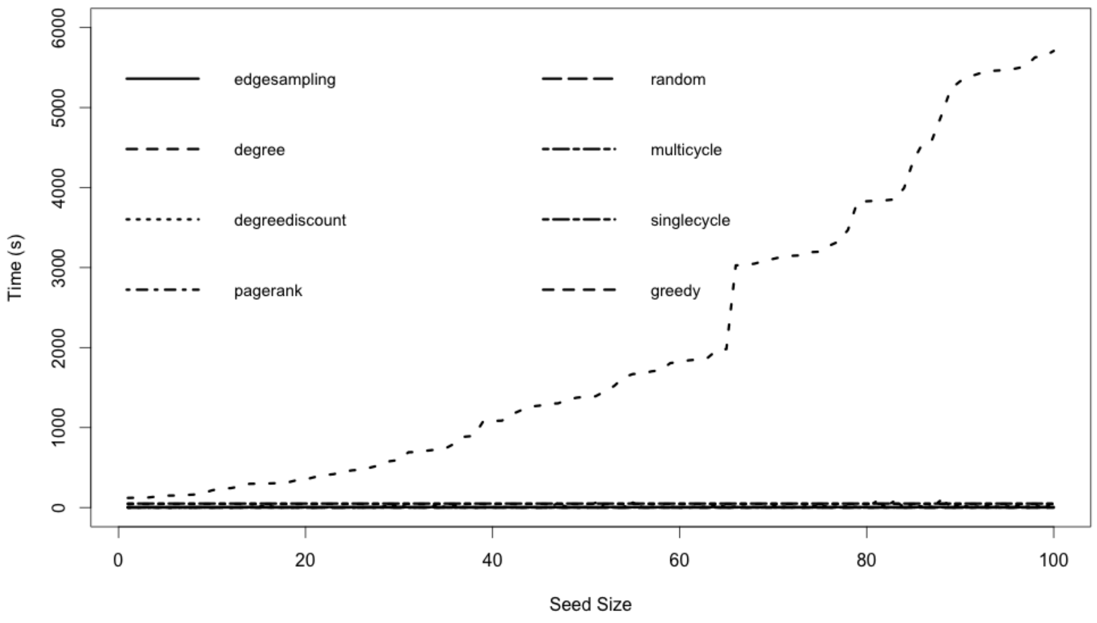

Figure 9: Running time of different methods on collaboration network data set

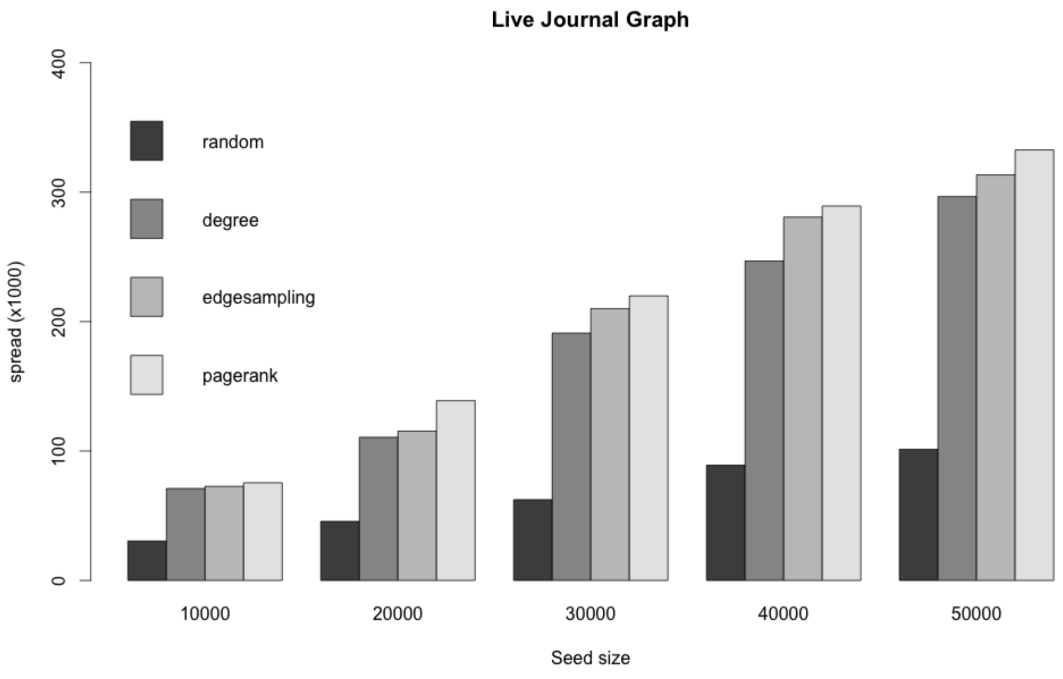

Figure 10: Spread of different methods on Live Journal Data set

7.2 Live Journal

Same as collaboration network data set, the number of simulations is set to 100. Here the seed sizes are much larger, and the results are reported for seed sizes of 10000, 20000, 30000, 40000 and 50000. Due to its size, running the greedy method is impossible, and both the single cycle and multi cycle as specified earlier run into problems, because of the massive amount of data they have to store in memory. For this data set, only edge sampling, degree, page rank and random seed selection methods are executed and the results are reported. Figure 10 shows the spread achieved by running different methods on the live journal data set. Horizontal axis shows the seed size, and the vertical axis shows the total spread. From the figure it is clearly obvious that page rank and edge sampling are superior to the other methods. Random method is just used as a baseline to compare with the other methods.

Table 2 shows the running time of different algorithms. random and degree methods as expected are very fast and page rank and edge sampling both run in acceptable time given the input size. As explained earlier, greedy method cannot scale to graphs of this size.

method	time (s)
random	21
degree	49
pagerank	206
edgesampling	705

Table 2: Running time

7.3 Artificial Graphs

In this section the spread of degree, pagerank and edge sampling methods on different types of artificially generated graphs are investigated. As a baseline the spread of random set of seeds are also drawn. Each of the graphs carry some specific properties that distinguishes them from each other. As observed from the real graph examples in the previous sections, edge sampling method is not always performing better than the heuristics methods such as pagerank and degree. Here, it will be shown for which types of graphs edge sampling method is performing better.

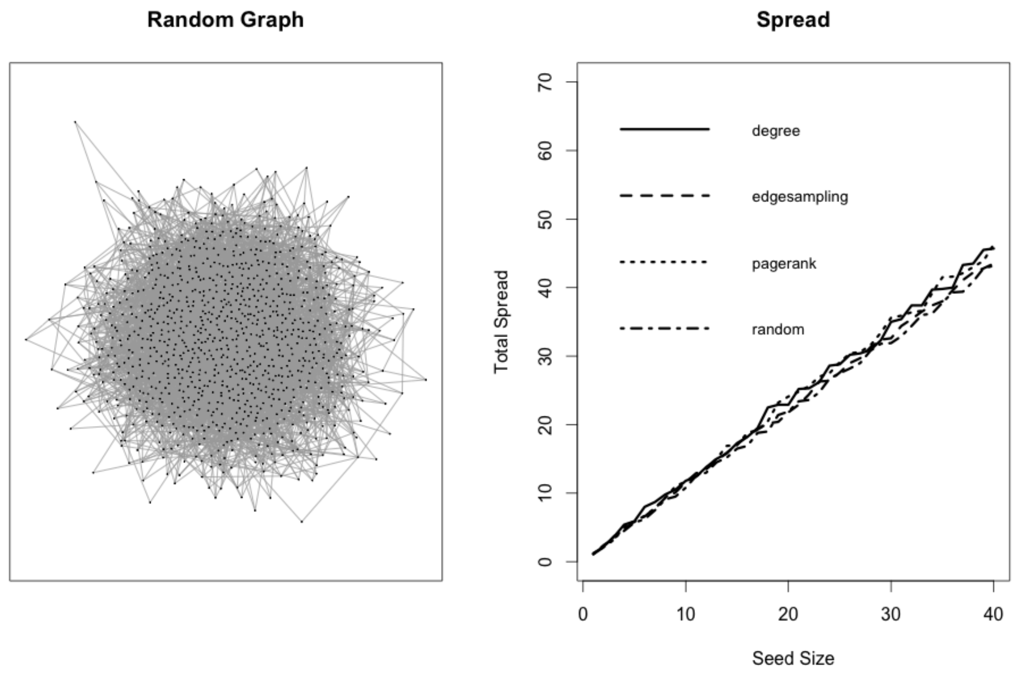

Figure 11: Random graph of 1000 vertices and 4000 edges

Figure 11 shows the spread on a graph with randomly generated edges. The source and destination of edges are sampled randomly from all the vertices available. From the figure, it can be observed that pagerank and degree method are performing better than edge sampling, although the difference in the final result is not significant.

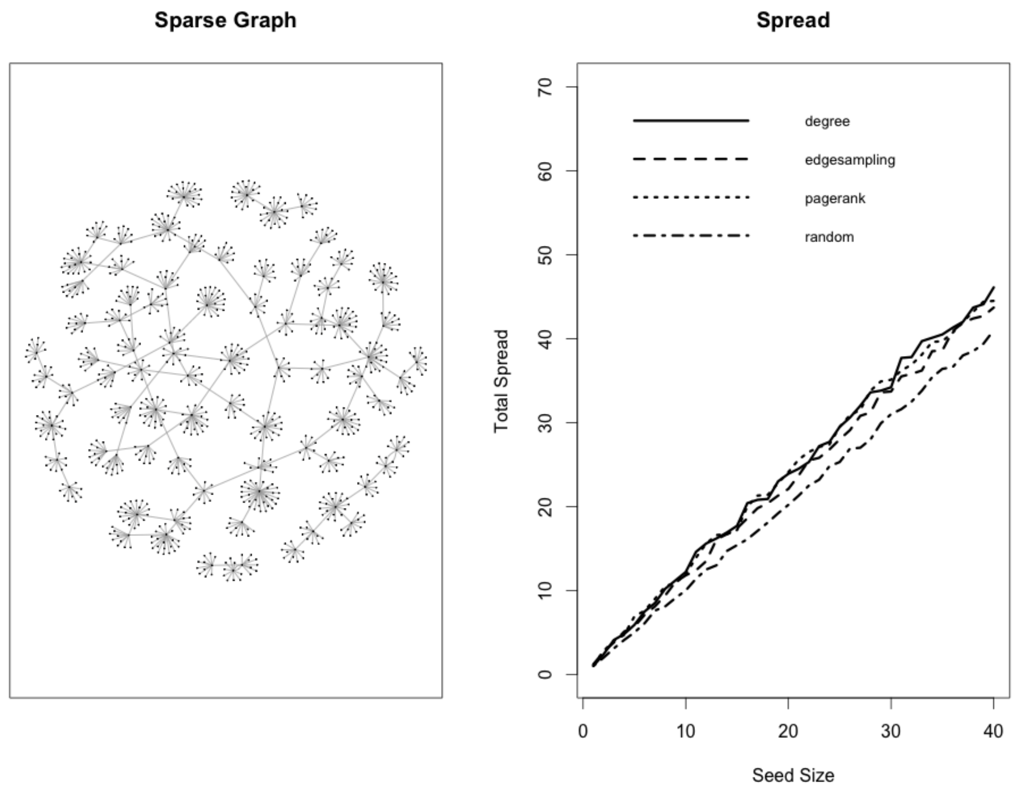

Figure 12: Sparse graph with 1000 vertices and 1000 edges

Figure 12 shows the spread on a sparse graph. The graph is generated similar to random but with a lot fewer edges. Similar to the random graphs, the spread of the different methods do not differ significantly.

Figure 13 shows the spread on a skewed graph. Here the difference in the results are more significant. As expected page rank and degree methods are performing better than edge sampling. This can be explained by the fact that seeds are picked from the more dense regions of the graph and page rank and degree are choosing the nodes with highest amount of neighbors. Due to the nature of edge sampling methods, it is possible that some nodes from the less denser area of the graph are chosen as seed which have caused the spread for edge sampling method to be lower.

The vertices and edges in the graph of figure 14 are clustered into two set. The smaller cluster is a fully connected component, which means all of the vertices are connected to each other. Because of this all of the vertices from the smaller cluster

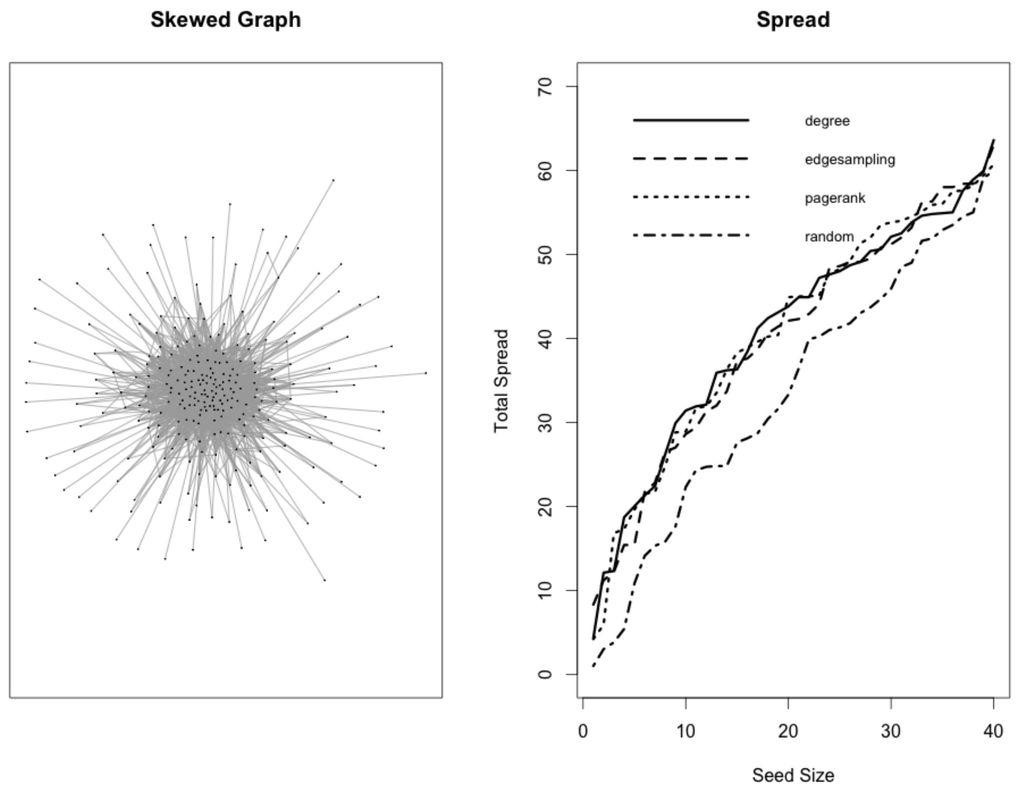

Figure 13: Skewed graph with 1000 vertices and 4000 edges

have very high degree. This explains the reason why degree method is not performing as well as edge sampling and page rank methods, both of which consider more global information when choosing the seeds.

7.4 Discussion

From the experiments performed in this section, it is obvious that while none of the methods are performing as good as the greedy method, some of them are good candidates for influence maximization on large graphs. Pagerank, Degree and Edge Sampling methods are very fast, and the spread achieved by these methods are acceptable. Different properties of graphs, affect the performance of each of the methods, hence the best way to chose the seed is to run all three methods on the graph and compare the spread for different seed sizes. None of the methods depend on the seed size, and running them once will return the list of vertices in order of how good they are in increasing the spread in the graph. Users then can easily chose the desired number of seeds. Pagerank and edge sampling, generally consider more

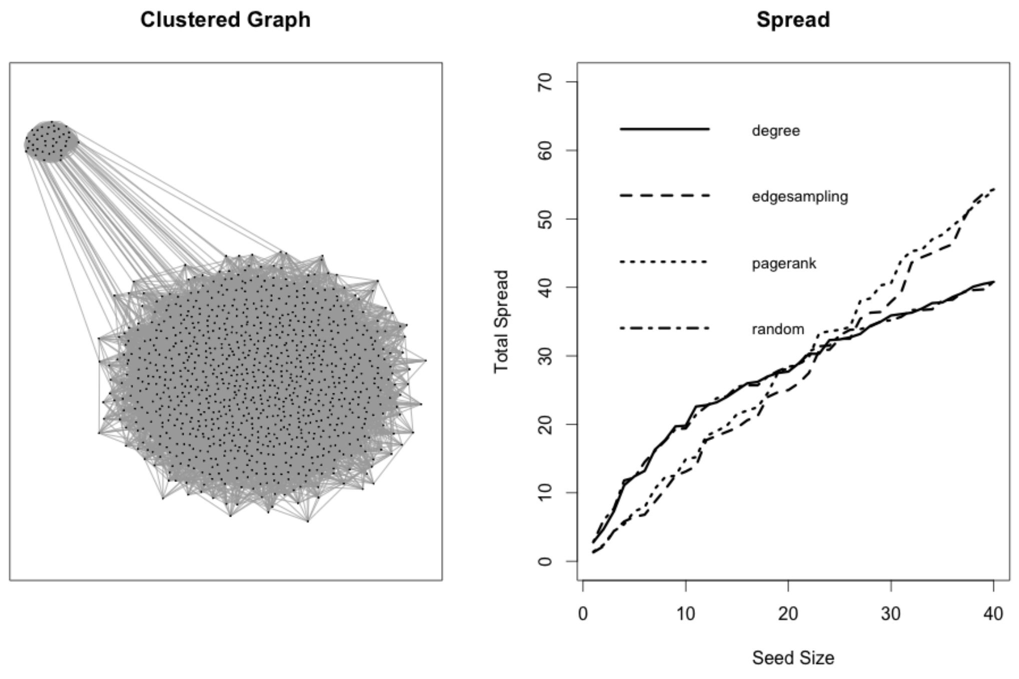

Figure 14: Clustered graph with 1000 vertices and 10000 edges

global information than degree method. Degree method, might perform very good on some types of graphs, even better than edge sampling, but it does not guarantee the solution is good for all types of graphs.

8 Conclusion

Influence maximization gained a lot of attention with introduction of social networks. It is the problem of finding a set of vertices that could maximize the spread of information throughout the network. This thesis was set out to study the influence maximization, specially its application in marketing, and its current solutions in detail. The study has also sought to provide a breakdown of the advantages and disadvantages of the solutions to Influence maximization, specifically in terms of scalability. Three main questions were studied in this thesis:

- Does the traditional greedy solution to Influence maximization apply to large scale graphs?
- What are the characteristics of a solution for large scale graphs?
- And how do these solutions compare with the greedy method and with each other in terms of quality?

Both the literature and the empirical findings in this thesis have shown the greedy solution presented by Kempe et al. [KKT03] has scalability issues. Even on small graphs it requires a substantial amount of time and resources. The greedy solution has two main flaws that attribute to the scalability problem. First of all, it is iterative, which means it requires tens of thousands of iterations in order to achieve a good solution. Secondly, it cannot be parallelized due to the way the algorithm is designed. At the core of their computation engines, all big data platforms rely heavily on parallelism. Platforms such as Hadoop and Spark, take advantage of big clusters of computers, by breaking down the program into several smaller tasks. These tasks are then processed by multiple computation units simultaneously. As a result, greedy method cannot be implemented on big data platforms. Other solutions that were discussed; Pagerank, Degree and Edge sampling are all parallelizable. Pagerank and Degree are simple heuristics that only require local information in the graph, hence the program can be run for all the vertices in parallel. The Edge sampling method that was introduced in the thesis, is a combination of the greedy and the heuristics. Although it requires some iterations to achieve a better solution, it is parallelizable and can be applied to graphs of any size.

In section 6, several experiments on different data sets were performed. In terms of quality, the greedy method performs best on small graphs. Pagerank, Degree and Edge sampling also performed quite well on the small data set, although none could match the greedy method. But on bigger data sets the greedy method could not provide a solution within a reasonable amount of time. All other three methods ran very fast and provided comparable solutions. However these solutions were not consistent across different graphs. On some graphs, one method performed better than the others, which was not the case in other graphs. Experiments on artificial data sets revealed that the shape of the graph affects the solution provided by each method. Pagerank and Edge sampling are able to provide a more consistent solution across different types of graphs. Whereas, Degree method does not guarantee a good solution all the time and it is heavily affected by the shape of the graph. All three methods fall under the heuristics category. This means that, although the experiments in this thesis showed that Pagerank and Edge sampling are performing very well, they do not guarantee to provide the best solution for every data set. Given that these methods run reasonably fast on large graphs, my recommendation is to test all the methods on the desired graph, and choose the one that provides the best solution.

Influence maximization is a vast topic and in this thesis only the basic problem was covered. Several related topics were left out, and many other challenges still remain. Finding a deterministic function (one that does not rely on the Monte Carlo simulations) for the greedy method is one area that is still being studied. Topic based influence maximization, i.e. users influencing each other differently with regards to different topics is another area that was not covered in the thesis. Influence degradation, a research area that involves dynamic influence probabilities that will change over time, was also left out. The Edge sampling method presented here, is utilizing some of the advanced features in Spark, such as in memory computation and vertex partitioning. However, there is still room for improvement. Better memory usage, faster connected component size computation, and more intelligent ranking of seed nodes are areas that can be improved further.

References

CSH13 Cheng, S., Shen, H. and Huang, J., Staticgreedy: solving the scalability-accuracy dilemma in influence maximization. *Proceedings of the 22nd ACM international conference on Conference on information and knowledge management*, San Francisco, California, USA, 2013, pages 509–518.

CWY09 Chen, W., Wang, Y. and Yang, S., Efficient influence maximization in social networks. *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, New York, NY, USA, 2009, ACM, pages 199–208.

DjG04 Dean, J. and Ghemawat, S., Mapreduce: Simplified data processing on large clusters. *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, Berkeley, CA, USA, 2004, USENIX Association, pages 10–10.

DpR01 Domingos, P. and Richardson, M., Mining the network value of customers. *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, New York, NY, USA, 2001, ACM, pages 57–66.

DpR02 Domingos, P. and Richardson, M., Mining knowledge-sharing sites for viral marketing. *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '02, New York, NY, USA, 2002, ACM, pages 61–70.

GBL10 Goyal, A., Bonchi, F. and Lakshmanan, L. V., Learning influence probabilities in social networks. *WSDM '10: Proceedings of the third ACM international conference on Web search and data mining*, New York, NY, USA, 2010, ACM, pages 241–250.

GBL11 Goyal, A., Bonchi, F. and Lakshmanan, L. V. S., A data-based approach to social influence maximization. *Proc. VLDB Endow.*, 5,1(2011), pages 73–84.

GGL03 Ghemawat, S., Gobioff, H. and Leung, S.-T., The google file system. *SIGOPS Oper. Syst. Rev.*, 37,5(2003), pages 29–43.

GLW11 Goyal, A., Lu, W. and Lakshmanan, L. V., Celf++: Optimizing the greedy algorithm for influence maximization in social networks. *Proceedings of the 20th International Conference Companion on World Wide Web*, WWW '11, New York, NY, USA, 2011, ACM, pages 47–48.

KKT03 Kempe, D., Kleinberg, J. and Tardos, E., Maximizing the spread of influence through a social network. *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '03, New York, NY, USA, 2003, ACM, pages 137–146.

KmS06 Kimura, M. and Saito, K., Tractable models for information diffusion in social networks. *Knowledge Discovery in Databases: PKDD 2006*, FÃŒrnkranz, J., Scheffer, T. and Spiliopoulou, M., editors, volume 4213 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pages 259–271.

LKG07 Leskovec, J., Krause, A., Guestrin, C., Faloutsos, C., VanBriesen, J. and Glance, N., Cost-effective outbreak detection in networks. *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, New York, NY, USA, 2007, ACM, pages 420–429.

MAB10 Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N. and Czajkowski, G., Pregel: A system for large-scale graph processing. *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, New York, NY, USA, 2010, ACM, pages 135–146.

NEM78 Nemhauser, G., Wolsey, L. and Fisher, M., An analysis of approximations for maximizing submodular set functions. *Mathematical Programming*, 14,1(1978), pages 265–294.

PBM98 Page, L., Brin, S., Motwani, R. and Winograd, T., The pagerank citation ranking: Bringing order to the web. *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, 1998, pages 161–172.

SNK08 Saito, K., Nakano, R. and Kimura, M., Prediction of information diffusion probabilities for independent cascade model. *Proceedings of the*

12th International Conference on Knowledge-Based Intelligent Information and Engineering Systems, Part III, KES '08, Berlin, Heidelberg, 2008, Springer-Verlag, pages 67–75.

Val90 Valiant, L. G., A bridging model for parallel computation. *Communications of the ACM*, volume 33, New York, NY, USA, aug 1990, ACM, pages 103–111.

VVM13 Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B. and Baldeschwieler, E., Apache hadoop yarn: Yet another resource negotiator. *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, New York, NY, USA, 2013, ACM, pages 5:1–5:16.

XRG13 Xin, R. S., Gonzalez, J. E., Franklin, M. J. and Stoica, I., Graphx: A resilient distributed graph system on spark. *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, New York, NY, USA, 2013, ACM, pages 2:1–2:6.

ZCD12 Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S. and Stoica, I., Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, Berkeley, CA, USA, 2012, USENIX Association, pages 2–2.

ZCF10 Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S. and Stoica, I., Spark: Cluster computing with working sets. *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, Berkeley, CA, USA, 2010, USENIX Association, pages 10–10.

ZZG13 Zhou, C., Zhang, P., Guo, J., Zhu, X. and Guo, L., Ublf: An upper bound based approach to discover influential nodes in social networks. *Data Mining (ICDM), 2013 IEEE 13th International Conference on*, Dallas, TX, Dec 2013, pages 907–916.