Estimating Reachability Set Sizes in Dynamic Graphs

Sudarshan Aji

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Anil Vullikanti, Chair
Madhav Marathe
Keith Bisset

May 5, 2014
Blacksburg, Virginia

Keywords: Dynamic Graphs, Reachability, Graph Mining
Copyright 2014, Sudarshan Aji

Estimating Reachability Set Sizes in Dynamic Graphs

Sudarshan Aji

(Abstract)

Graphs are a commonly used abstraction for diverse kinds of interactions, e.g., on Twitter and Facebook. Different kinds of topological properties of such graphs are computed for gaining insights into their structure. Computing properties of large real networks is computationally very challenging. Further, most real world networks are dynamic, i.e., they change over time. Therefore there is a need for efficient dynamic algorithms that offer good space-time trade-offs. In this thesis we study the problem of computing the reachability set size of a vertex, which is a fundamental problem, with applications in databases and social networks. We develop the first Giraph based algorithms for different dynamic versions of these problems, which scale to graphs with millions of edges.

Acknowledgments

This work has been made possible because of the support of many people. Firstly, I thank my advisor Dr.Anil Vullikanti for working with me and being a source of constant guidance and support in my research. I thank Dr.Madhav Marathe and Dr.Keith Bisset for supporting me by being on the committee. I have been exposed to a lot of quality research in diverse areas by being a part of NDSSL, and for that I am extremely thankful to all my colleagues and friends at NDSSL. I thank my brother Ashwin Aji, and my sister-in-law Tilottama Gaat for being pillars of support during my time at Virginia Tech. None of this would have been possible without them. Finally, I thank my parents for pushing me to do my best and supporting me in all my endeavors.

Contents

List of Figures **vii**

List of Tables **ix**

1 Introduction **1**

1.1 Preliminaries and problem statement 2

1.2 Problem Statement 4

1.3 Contributions 5

1.4 Organization 6

2 Motivation and Background **8**

2.1 Motivation 8

2.2 The DRS problem: Cohen's technique 11

3 Related Work **12**

3.1 Overview 12

3.2 Finding topological properties on large static graphs 13

3.3 Parallel programming to process large graphs 14
3.4 Distributed Graph Processing Frameworks 14
3.5 Computation of topological properties on large dynamic graphs 16

4 Sequential Algorithm for estimating sizes of reachability sets in dynamic graphs 18

4.1 Dynamic version of Cohen's technique: approach of Roditty and Zwick [1] . 19
4.2 Experimental evaluation 23
4.2.1 Purpose 23
4.2.2 Data sets 23
4.2.3 Execution 23
4.2.4 Results 25

5 Dynamic reachability using Giraph 28

5.1 Background 29
5.2 Parallel and dynamic algorithm for DRS-INC(ϵ, δ) in Giraph 31
5.3 Computing maximum and minimum reachability set size incrementally . . . 36
5.4 Parallel algorithm for DRS-DEC(ϵ, δ) in Giraph 36
5.5 Experimental evaluation 37
5.5.1 Purpose 37
5.5.2 Cluster Specifications 39
5.5.3 Hadoop Setup 39
5.5.4 Data sets 39

5.5.5 Experimental Setup . . . 40

5.5.6 Results . . . 40

6 Future Work and Conclusions **43**

6.1 Future Work . . . 43

6.2 Conclusion . . . 44

7 Bibliography **45**

List of Figures

1.1 The vertices in blue indicate the reachability set of u. The vertices in red indicate the reachability set of v. w is reachable from both u and v 3

1.2 An illustration of the change in the reachability sets of u and v when edges are added to the graph. Blue vertices are in the reachability set of u and the red vertices are in the reachability set of v. Vertices shaded with both colors belong to the reachability sets of both vertices 4

4.1 Timing: The figure represents the time taken to perform different subroutines in the algorithm. 26

4.2 Timing: Comparison of running times of the sequential algorithm in the static and dynamic settings. 27

4.3 Timing: The ratio of update time to the running time in the static setting. . 27

5.1 Comparison of time taken in the static and dynamic settings 41

5.2 Time of execution of one iteration of the approximation algorithm on data sets of different sizes. 41

5.3 Ratio of time taken to add an edge to total running time of the Giraph algorithm 42

5.4 Ratio of time taken to remove an edge to total running time of the Giraph algorithm 42
5.5 Comparing execution times of Giraph and the sequential algorithms 42

List of Tables

4.1 Description of Data collected 24

Chapter 1

Introduction

Networks form an important part of our lives today. We interact with other people in different ways every day. Interaction may be talking to someone face to face in the real world at a physical location such as an office or a school. An interaction between people may also be in the virtual world such as on Facebook or Twitter or any other social media platform that we use. The networks that we are a part of are typically very large, e.g the user network Facebook consists of millions of users. Mining data from such large networks, especially, computing different kinds of topological properties, is a popular and an important area of research today. Further, real world networks also grow in size with time. Therefore, it is interesting to study topological properties of dynamically growing networks.

There are many topological properties of graphs that are of interest to researchers, such as connectivity, clustering [2] [3], centrality [4] [5], subgraph isomorphism [6], etc. (see, e.g., [7]) These properties are useful in understanding the structure of the graphs (e.g., critical nodes and robustness properties), as well as in understanding processes defined on these graphs, e.g., the spread of diffusion processes, such as epidemics [8] and influence [9]. One of the most fundamental properties is related to "reachability": given a node $v \in V$ in a graph $G = (V, E)$, the reachability set $S(v)$ of v is defined as the set of all nodes $u \in V$ which are reachable from v. Computing the reachability sets is useful in a number of applications such

as semantic-web applications which make use of RDF and XML based documents [10] and program analysis [11]. In some applications, it suffices to compute the sizes of reachability sets. Yannakakis in [12] tackles the problem of transitive closure in database networks. Bramandia et al. in [13] apply graph problems to the domain of the semantic-web and graph structured databases

Studying the distributions of reachability set sizes [14] is useful in some applications such as identifying potentially influential users in a social network.

Computing $|S(v)|$ for all $v \in V$ is a challenging problem in very large graphs. For instance, matrix multiplication based approaches can be used to compute all the reachability sets in $O(n^{\omega})$ time, where $\omega \simeq 2.38$ denotes the exponent corresponding to matrix multiplication algorithms [15]. However, this is unfeasible both in terms of space and time in massive networks. This motivates the need for efficient approximation for the $|S(v)|$'s. Further, real networks are temporal and dynamic [16]. An observation of real networks such as an online social network reveals that edges and vertices are constantly being added and removed from the social network. This motivates dynamic algorithms for computing the reachability set sizes. In this thesis, we focus on estimating the sizes of reachability sets of all vertices in a large graph that is dynamically growing. We also come up with an implementation to estimate reachability set sizes of all vertices in a large graph that is dynamically shrinking.

1.1 Preliminaries and problem statement

We define some of the notation we use, and the problems we study formally. Given a directed graph $G(V, E)$, node w is said to be reachable from v if there exists a path from v to w (see, e.g., Figure 1.1); we use $v \rightsquigarrow w$ to denote the existence of such a path. We define $S(u) = \{v \in V | u \rightsquigarrow v\}$, the set of all nodes reachable from u, as the reachability set of u. In the case of Figure 1.1, the reachability set sizes of u and v are 4. The reachability set size of w is 1. Computing the sizes of all reachability sets takes super-linear time, and

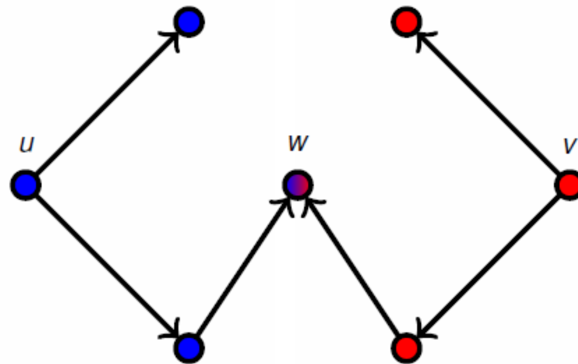

Figure 1.1: The vertices in blue indicate the reachability set of u. The vertices in red indicate the reachability set of v. w is reachable from both u and v

is unfeasible in large graphs. Therefore, we consider a notion of approximate computation. For given $\epsilon, \delta > 0$, we say that $\hat{s}(v)$ is an (ϵ, δ)-approximation to $|S(v)|$ if

$$\Pr[\hat{s}(v) \in [(1-\epsilon)|S(v)|, (1+\epsilon)|S(v)|]] > 1 - \delta$$

Given a directed graph $G = (V, E)$, and parameters ϵ, δ, the (ϵ, δ)-*Approximate Reachability Size* problem involves computing an (ϵ, δ)-approximation $\hat{s}(v)$ for each $v \in V$.

We focus on dynamic graphs, in which the graph evolves over time. For instance, the graph in Figure 1.2 is obtained by adding one edge to the graph in Figure 1.1. We formalize a dynamic graph as a sequence $\mathcal{G} = (G_i = G(V, E_i), i = 1, 2, \ldots)$ of graphs on the same node set V but changing set of edges. We assume the graphs change gradually. Specifically, we assume $||E_i| - |E_{i-1}|| = 1$, for $i = 1, 2, \ldots,$. There are also cases when graphs change quickly, i.e many edges may be added or removed from the graph at every step a change occurs to the graph.

When edges are added to the graph, the reachability sets of vertices change. We will have to maintain the reachability sets of vertices as and when edges are added to the graph. This is a non-trivial task. Figure 1.2 represents the changes that take place when 1 edge is added

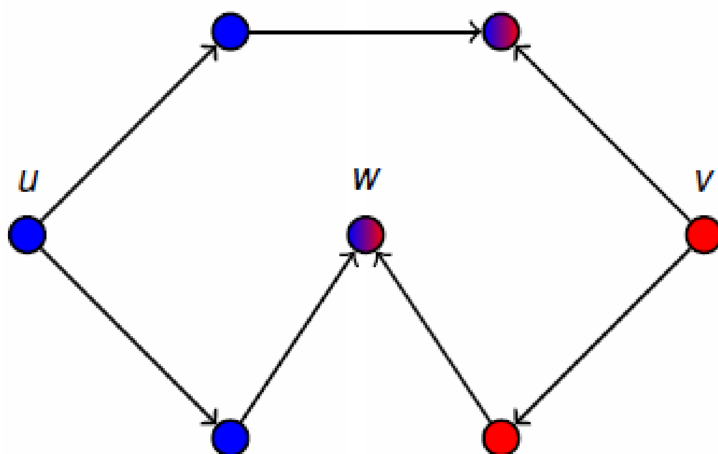

Figure 1.2: An illustration of the change in the reachability sets of u and v when edges are added to the graph. Blue vertices are in the reachability set of u and the red vertices are in the reachability set of v. Vertices shaded with both colors belong to the reachability sets of both vertices

to the graph in 1.1. The addition of the edge causes the reachability set of u to change, and increases the size of its reachability set. $|S(u)|$ is now 5. The changing nature of reachability sets in dynamic graphs are illustrated in Figure 1.2

1.2 Problem Statement

Finding Reachability Sets in a dynamic graph Given a sequence of graphs $\mathcal{G} = (G_i = G(V, E_i), i = 1, 2, \ldots,)$ with $||E_i| - |E_{i-1}|| = 1$, and parameters $\epsilon, \delta > 0$, the goal is to compute an (ϵ, δ)-approximation $\hat{s}(v)$ for every node $v \in V$, for each i.

We consider two kinds of dynamic changes in a graph G_i.

- **Incremental:** An edge is added to G_i. We use DRS-Inc(ϵ, δ) to refer to the incremental setting henceforth.

- **Decremental:** An edge is removed from G_i. We use DRS-Dec(ϵ, δ) to refer to the decremental setting henceforth.

We are also interested in computing the following properties.

- $max_{v \in V} \hat{s}(v)$, $\min_{v \in V} \hat{s}(v)$
- The distribution of the estimated sizes of the reachability sets.

1.3 Contributions

We study different versions of the DRS problems in dynamic graphs. We focus on the Giraph framework for distributed algorithms because Giraph allows us to scale the problem to graphs having millions of edges. We have scaled the problem to graphs with upto 80 million edges in our experiments. Giraph can be used to scale the problem to graphs with a larger number of edges given sufficient resources.

The main contributions of this thesis are:

- A Python based sequential implementation of [1] to solve the DRS-Inc(ϵ, δ) problem.
- A Giraph based approach for DRS-Inc(ϵ, δ) and DRS-Dec(ϵ, δ). For DRS-Inc(ϵ, δ), our Giraph based adaptation has the same complexity (amortized $O(1)$ time per update) as the best sequential dynamic algorithm of Roditty et al. [1].
- Our implementations scale to graphs with 80 million edges. Updating the graphs on Giraph takes very little time. The observed ratio of time taken to update the reachability data structures to the running time of the algorithm on static graphs is very low. This confirms that the Giraph algorithm for dynamic graphs is efficient.

Techniques Used. We adapt the approach of Cohen [17] and the extension by Roditty et al. in [1] for the DRS problem.

Challenges

- We estimate reachability set sizes in dynamic graphs. The main challenge lies in updating the data-structures required to find out the reachability set size of a vertex when edges are added or removed from the graph.
- We use Giraph, a graph processing framework to find reachability set sizes. It takes effort to setup the framework on a multi-node cluster, and to thoroughly understand how the framework can be efficiently used to find reachability set sizes.
- We cannot use explicit data-structures to maintain reachability information in Giraph. Reachability information has to be maintained in an implicit data-structure in Giraph.

Theoretically the running time of the algorithm of Roditty et al. is $O(m \log n + q)$. Roditty et al. make use of Italiano's data structure [18] to maintain reachability sets dynamically. Italiano's data structure requires $O(n^2)$ space to answer reachability queries. Roditty et al.'s approach involves maintaining reachability information for $\log n$ sinks. Therefore, the space requirement for using Roditty et al.'s approach is only $O(n \log n)$. We further reduce the space requirement by adapting the algorithm to Giraph.

Empirically we find that the serial implementation of the algorithm scales to graphs with upto 2 million edges. We have used Networkx for the serial implementations of the algorithm. Networkx is known to have a large memory footprint. We have to explore other libraries with a lower memory footprint to implement the serial version of Roditty et al.'s algorithm. We find that the Giraph implementation scales well to graphs with 80 million edges, and that the running time of the Giraph implementation is much lesser than the serial implementation.

1.4 Organization

The thesis is organized as follows. The background and motivation for the thesis are presented in Chapter 2. Relevant literature that was studied is presented in Chapter 3. The

contribution involving the serial implementation of the algorithm from [1]is presented in Chapter 4. We explore the giraph framework, and present the Giraph implementation of the algorithm in Chapter 5. We conclude our findings and present future work in Chapter 6.

Chapter 2

Motivation and Background

2.1 Motivation

Real world networks can be represented as graphs; these include networks such as biological networks and social media networks such as Facebook or Twitter. Properties of the networks have been the subject of research in a large number of areas, including computer science [19], sociology [20], economics [21] and biology [22].

Some of the commonly studied properties include connected components, reachability, clustering coefficient [2] [3], different kinds of centrality measures [4] [5] and subgraph isomorphism [6]. See [7] for introduction and definitions of these graph properties. Some of the key applications of these graph properties include: understanding the overall structure of the network, e.g., the "bow-tie" structure of the web-graph [23], in comparing and distinguishing between different networks [24], and studying influence of members of social networks [25].

Reachability is one of the most basic properties of a graph. Consider a directed graph $G = (V, E)$. We say a node $w \in V$ is reachable from a node $v \in V$ if there exists a directed path from v to w in G. The set $S(v) = \{w : w \text{ is reachable from } v\}$ is referred to as the reachability set of node v. See Figure 1.1 for an illustration— the set of nodes shaded blue

form the reachability set of node u, while the red nodes form the reachability set of v; note that some nodes, such as w belong to $S(u) \cap S(v)$. In the case of undirected graphs, the size of the reachability set of a vertex is the size of the connected component of the graph to which it belongs. Finding reachability set sizes of vertices in an undirected graph is easier than finding reachability set sizes in directed graphs.

Computing connected components, reachability sets and reachability queries for a pair $u, v \in V$ of nodes are some of the most common subroutines in graph problems. Computing the reachability sets is useful in a number of applications such as semantic-web applications which make use of RDF and XML based documents [10] and program analysis [11]. In some applications, it suffices to compute the sizes of reachability sets. Yannakakis in [12] tackles the problem of transitive closure in database networks. Bramandia et al. in [13] apply graph problems to the domain of the semantic-web and graph structured databases Another application of measures related to reachability is in the context of diffusion processes on networks. Social networks are hubs of information exchange between people. They act as forums for people to voice their opinions on various issues, and act as a platform for people to get together and organize themselves. A model of information propagation on social networks is the cascade model, where information is propagated in the network with time. There are a number of models of cascades, e.g., [26]; these include the "independent cascades" and SIS models. Some of the key problems related to understanding cascades include: determining influential spreaders at a particular time, and designing algorithms for selecting seeds for the cascades so that the resulting cascade size is maximized. A common approach for such problems is to find nodes which cause the largest cascades– this corresponds to the node with the largest reachability set. Other properties related to cascades, e.g., the maximum cascade size and distributions over them, are useful in understanding occurrence of critical events.

The focus of this dissertation is on computing reachability set sizes of all nodes, and other associated statistics in real world networks. There are a number of challenges with such networks:

1. Massive scale: Real world networks such as social networks have millions of vertices and edges. e.g. The Twitter network we studied for this dissertation had close to 200 million edges. We could only use a subset of this graph for our study due to system constraints.

2. Heterogeneous structure: most networks have irregular and multi-scale structure. Broder et al. have discussed the structure of the World Wide Web in [23]. The degree distribution of vertices in social networks follows the power law. Amaral et al. have studied the structure of different types of real world networks in [27].

3. Dynamic: Real world networks are dynamic in nature. Consider a social network such as the Facebook network. Every user is a vertex, and a friendship between users is an edge. In Facebook, new users are added, existing users are deleted, and users add other users as friends and so on. Therefore, in the Facebook network, vertices and edges are constantly being added and deleted with time. Similarly, even in other real world networks, vertices and edges are being added and deleted with time. Therefore, real world networks are dynamic in nature.

Polynomial time algorithms for computing reachability properties are well known, but they do not easily scale to such massive and dynamic real world networks. We focus on approximate solutions. There has been a lot of work on approximate computation of reachability properties. One of the key results is by Cohen[17], who developed a novel technique based on properties of the exponential distribution. Cohen also developed a parallel approximation algorithm, but her work is limited to static graphs. Roditty et al. have proposed a version of Cohen's reachability estimation algorithm [17] for graphs that grow in size [1]. We build on the techniques of Cohen and Roditty et al. for reachability size estimation. Serial algorithms do not allow us to scale the problem to large graphs. Therefore we explore parallel algorithms to scale the problem to large dynamic graphs.

There has been research in processing large static graphs over the past few years. Apache Hadoop has been used to implement graph algorithms on a distributed environment [28].

But Hadoop is not good at handling dynamic data. Therefore we use Apache Giraph, an implementation of Pregel. Google introduced a distributed graph processing framework, Pregel [29]. The Apache developer community have an implementation of Pregel called Giraph [30] to process large graphs in parallel on distributed network. Giraph is used to handle large graphs easily and more effectively. It also allows the graph to be manipulated dynamically. Therefore, we have extended the approximation algorithm of Roditty et al. [1] and have evaluated the performance of our implementation.

2.2 The DRS problem: Cohen's technique

Given a graph $G(V, E)$, and parameters ϵ, δ, Cohen's algorithm involves the following steps:

- Repeat the following steps for k iterations
- Cohen states that for $k = O(\epsilon^{-2} \log(\frac{1}{\delta}))$, $\frac{||S(v)| - \hat{s}(v)|}{|S(v)|} \leq \epsilon$
- Assign a rank $\ell(v)$ to every vertex $v \in V$ based on some permutation applied to the set of vertices. The ranking is a bijection $V \rightarrow \{1, 2, 3, ..., |V|\}$
- $S(v) = \{u \in V | v \rightsquigarrow u\}$
- For every $v \in V, min_{u \in S(v)} \ell(u) = le(v)$ is found.

$\hat{s}(v)$ is estimated by using the relation $\hat{s}(v) = \frac{k}{\sum_{i=1}^{k} \ell(le(v))}$.

Theorem 1. *(Cohen [17]) Given a graph $G = (V, E)$, and parameters ϵ, δ, the above algorithm gives an (ϵ, δ)-approximation $\hat{s}(v)$ for each node $v \in V$ in time $O(m \log(n))$*

The Least-Element subroutine used in the paper to determine $\forall v \in V, le(v)$ takes $O(m)$ time. Cohen also suggests to repeat the sub-routine for k iterations where $k = O(\epsilon^{-2} \log(n))$ to get a good estimate. Therefore, the total running time to find $\hat{s}(v) \forall v \in V$ is $O(m \log(n))$.

Chapter 3

Related Work

3.1 Overview

Mining large quantities of data is a prominent area of research today. The ease in which data from social media is made available to researchers has contributed to an explosion in data. Researchers make use of data made available by social networks to analyze the diffusion of various contagions. Analyzing Twitter-data to detect the diffusion of public sentiment is a prominent area of research. Borge-Holthoefer et al. in [31] discuss the issue of influential spreaders in an information network. They have studied the 2011 protests in Spain for their research. The spread of information happens in the form of "cascades".
A cascade is a dynamic graph that grows when information spreads through the network. The cascade model has also been used to study the spread of information by Leskovec et al. Their paper [26] discusses patterns in spread of information over the blog network. They discuss the structure of a cascade that represents the flow of information between blogs. They make their observations by using various topological properties of the cascades of blogs.

A cascade has different properties that are of interest to a data analyst. Generic topological

properties of graphs such as the radius of a node, and the diameter of the graph can be applied to cascades as well. Properties such as the reachability set of a node, the size of the reachability set, and the distributions of the sizes of the reachability sets are all important properties of cascades. The reachability set is defined as the set of nodes reachable from a source node within a time interval. The size of the reachability set has been used in [31] to determine the most influential spreaders in an information network. The distribution of the size of cascades has been used by Lerman et al. in [32] to study the popularity of topics in Digg and Twitter.

3.2 Finding topological properties on large static graphs

The most fundamental property of the cascade that is useful is the size of the cascade itself. There have been theoretical approaches to estimate topological properties of graphs. Cohen provides a framework to estimate the size of the reachability set of a node in a directed graph [17]. Cohen's approach assigns ranks to the vertices of the graph from an exponential distribution, and makes use of the properties of the distribution to approximate the sizes of reachability sets. The framework follows a novel approach where the edges of the graph are reversed, and then the reachability set of each node is populated by traversing through sub-graphs of the reversed graph. Cohen's algorithm allows us to estimate the reachability set sizes of all vertices by traversing the reachability trees of only a few vertices in the graph. One iteration of Cohen's framework takes $O(m)$ time. Cohen's algorithm is run for multiple iterations to obtain an accurate approximation.

3.3 Parallel programming to process large graphs

There are various parallel programming constructs, languages, and frameworks available to optimize existing algorithms to compute topological properties of graphs.

A parallel approach to estimating the size of the reachability set has also been theorized in [17]. This paper has provided the foundation for many other works to solve similar problems. It might be easy to flood the network, so that every node in the network has information about every other node in the network. This might help in obtaining a distribution of the sizes of reachability sets. Baumann et al. establish tight bounds in [33] for flooding a network. The authors also introduce a tool to analyze flooding time in dynamic graphs. Their approach reduces the flooding protocol problem to computing diameters of random weighted graphs. Gregor et al. in their work [34] have extended the Boost Graphics Library to process large graphs in parallel. Their approach provides developers with a generic algorithm which requires "only the introduction of external (distributed) data structures for parallel execution".

Barnat et al. have implemented a CUDA based solution in [35] to compute strongly connected components of a graph. The limitations of this approach are that, it cannot be applied to dynamic graphs, and the input graph must be represented as a matrix. We will try to extend their approach to dynamic graphs, and to non-matrix representations of graphs.

3.4 Distributed Graph Processing Frameworks

The availability of frameworks such as Hadoop has enabled researchers to optimize existing algorithms by implementing efficient parallel approaches to solve problems on graphs. Kang et al. propose HADI, a Hadoop based algorithm to estimate the radius and diameter of massive graphs in [28]. They have showed that their estimator is 7.6 times faster than the

naive approach. They have implemented a parallel version of the Flajolet-Martin algorithm [36] to find the radii of nodes in a graph. The Flajolet-Martin approach has been used to optimize the space requirements of the algorithm in [28].

In our work, we also explore other parallel graph processing frameworks such as Pregel [29], and Giraph [30]. Pregel has been designed by Malewicz et al. to efficiently process massively parallel graphs. Pregel follows a "vertex-centric" approach. The graphs are processed in iterations. In each iteration, every vertex receives messages from its neighbors, performs some computation, and sends messages to its neighbors. Pregel has been designed to be scalable and fault tolerant. The Apache community has developed a framework based on Pregel, called Giraph [30], to process massively large graphs in parallel. Giraph is implemented in Java and runs on the Apache Hadoop map-reduce framework. However, Giraph uses a programming model called bulk synchronous parallel [**?**] to process large graphs. Giraph has methods to define computations that a vertex has to perform. All vertices that are active perform the defined computation in parallel. Vertices can also receive messages from other vertices and send messages to other vertices. Giraph allows the user to add or remove edges and vertices from the graph. The flexibility offered by Giraph, and the simple programming model help in processing large graphs allows us to implement algorithms for dynamic graphs.

In our research, we make use of cascades obtained from Twitter networks. We are currently focusing on estimating the sizes of the reachability sets of nodes in a large cascade. We will evaluate the different approaches we have studied, and implement an efficient algorithm to process large cascades in parallel .

3.5 Computation of topological properties on large dynamic graphs

Real world graphs evolve with time. Graphs such as a Facebook network or a Twitter network grow slowly. But a graph such as a cascade of information on Twitter grows at a much more noticeable rate. Therefore there is a need to estimate topological properties of graphs on large dynamically growing networks. Roditty et al. in [1] have proposed an array of algorithms to compute topological properties in dynamically evolving graphs. They propose a decremental algorithm for maintaining strongly connected components, algorithms to answer reachability queries between vertices in a graph and an algorithm to estimate the sizes of reachability sets of vertices of the graph. They have extended the approximation algorithm of Cohen [17] and have provided an improved algorithm to compute dynamic estimation of the size of reachability sets. They propose a $O(\log(m))$ algorithm [1], which is a significant improvement over the linear algorithm proposed in [17]. The incremental algorithm of Roditty et al. makes use of $\log(n)$ sinks. They reduce the problem of maintaining reachability sets to a problem of answering reachability queries. Vertices are assigned ranks, and an edge is added from a vertex to a sink based on the rank assigned to the vertex. Even after edges being added to the graph, the rank of the least element of a vertex can be answered by finding the least indexed sink reachable from that vertex. The running time of this algorithm is $O(m \log(n) + q)$ where q is the number of reachability queries made. The algorithm makes use of Italiano's incremental data structure [37] for answering reachability queries. Italiano's data structure is essentially a transitive closure matrix and is used to query the presence of a path between any two vertices in a graph. We modify this data structure to answer reachability queries only from $\log(n)$ sinks. Therefore Roditty et al.'s incremental algorithm requires $O(n \log(n))$ space.

Roditty et al. in [1] also describe algorithms for answering reachability queries. An approach that they use involves a decremental maintenance of strongly connected components of a

graph. While maintaining strongly connected components Roditty et al. in [1] have made use of techniques to maintain data structures for reachability information based on the works of other researchers. They use techniques proposed by Henzinger et al. [38] to maintain shortest path trees in answering reachability queries between vertices. Italiano in [37] discusses an algorithm for decremental maintenance of the transitive closure of a DAG. The work of Italiano has been extended to general directed graphs by Frigioni et al. in [39]. Italiano in [18] proposed an algorithm to incrementally maintain the transitive closure of a graph. We use the methods proposed in [18] to estimate sizes of reachability sets in graphs that are dynamically growing.

We have also studied other novel techniques to answer reachablity queries on dynamic graphs. Some proposed methods include efficient labeling of vertices of a graph to answer reachability queries efficiently. Yildirim et al. in [40] have discussed the trade-offs needed to be considered while answering reachability queries. They have introduced an indexing scheme for large graphs that can be used to answer reachability queries efficiently. Bramandia et al. [13] propose a 2-hop labeling scheme for large graphs, and introduce an algorithm for the incremental maintenance of labeling on large dynamically evolving graphs. In [13], for any given vertex v in the graph, the authors maintain lists $L_{in}(v)$ and $L_{out}(v)$ to assist in answering reachability queries. The presence of a path from a vertex u to a vertex v in the graph is identified by querying if $u \in L_{in}(w)$ and $v \in L_{out}(w)$. Bramandia et al. also describe algorithms to maintain the reachability data structures for dynamic graphs.

Chapter 4

Sequential Algorithm for estimating sizes of reachability sets in dynamic graphs

We consider the DRS-INC(ϵ, δ) problem and start with a sequential algorithm. Adding a single edge may result in the need to update the reachabilty sets of many vertices in the graph. The challenge lies in updating reachability sets of all vertices in the graph when edges are added to it. A naive version of Cohen's algorithm to solving DRS-INC(ϵ, δ) requires redoing DFS when edges are added. This might be expensive in the worst case, though it is possible the average complexity might be bounded because ranks are chosen randomly. However, this is hard to analyze. Therefore, we use the approach of Roditty et al. [1].

4.1 Dynamic version of Cohen's technique: approach of Roditty and Zwick [1]

Roditty et al. in [1] design a simple trick to reduce Cohen's technique to reachability queries from a small set of additional nodes; this enables efficient updates in the incremental setting. The idea in [1] is to reduce the number of reachability sets being maintained in trying to estimate the sizes of reachability sets of the vertices in a graph. They also reduce the problem of updating reachability set sizes to merely answering reachability queries. The algorithm introduces meta-sinks to the graph. A reachability set is maintained at each meta-sink. The reachability sets thus maintained are used to identify the smallest label that can reach a vertex. The reachability sets of the meta-sinks are maintained by using Italiano's incremental algorithm [18] for maintaining the transitive closure of dynamically growing graphs. Italiano in [18] uses a transitive closure matrix to update reachability information of vertices. The space consumed by a transitive closure matrix is $O(n^2)$. An $O(n^2)$ does not scale well with size, therefore by using Roditty et al.'s approach we reduce the space consumption to $O(n\log(n))$, thus improving the scaling of the algorithm.

For a given graph $G = (V, E)$, and parameters $\epsilon, \delta > 0$, the main idea of Roditty et al. in [1] involves the following steps:

- Assign labels $\ell(v)$ to each node $v \in V$ randomly, as described in [17] (this process has to be repeated $O(\delta\frac{1}{\epsilon})$ times, using the estimator, as discussed earlier; we discuss one iteration here).
- Introduce a set of nodes $U = \{u_i : 1 \le i \le \log_{1+\epsilon}(n)\}$. For $u_i \in U$, we set $\ell(u_i) = ((1+\epsilon)^i$.
- For each node $v \in V$ with $\ell(v) \in [(1+\epsilon)^{i-1}, (1+\epsilon)^i]$, introduce edge (v, u_i).
- For any $v \in V$, let $\text{QUERY}(v, u_i)$ return TRUE if u_i is reachable from v. The minimum rank $le(v)$ among the nodes reachable from v is set to $le(v) = \min\{\ell(u_i) :$

$\text{QUERY}(v, u_i) == TRUE, u_i \in U\}$.

The initialization procedure is described in more detail as Algorithm $RZ - COHEN - Initialize$ in Algorithm 1. The update, query, and least element assignment subroutines are described as algorithms in Algorithm 2, Algorithm 3, and Algorithm 4 respectively.

Algorithm 1 $RZ - COHEN - Initialize$

1: **Input:** Graph $G(V, E)$, $|V| = n$, $|E| = m$, $\epsilon > 0$
2: **Data structures:** Reachability Set $REACH(s_i)$
3: **Variables:** Lowest label that can be assigned to a vertex - Least Element LE
4: **Algorithms:** $BFS(G, u)$: Returns the nodes in the BFS traversal of G' with u as the source
5: Reverse the edges of the graph G. Let the reversed graph be $G'(V', E')$
6: Introduce $log_{1+\epsilon}n$ meta-sinks. Let the set of meta-sinks be $S = s_i|1 \leq i \leq n$
7: Using a random permutation of the vertices, assign a label $L(v) : V' \rightarrow \{1, 2, 3, ..., |V|\}, \forall v \in V'$
8: **for** $v \in V'$ **do**
9:    **if** $L(v) \in [(1 + \epsilon)^{i-1}, (1 + \epsilon)^i]$ **then**
10:       Add an edge (s_i, v) to G'
11:       Add v to $REACH(s_i)$
12:    **end if**
13: **end for**
14: **for** $s \in S$ **do**
15:    Add $BFS(G', s)$ to $REACH(s)$
16: **end for**

Implementation of the algorithm

The algorithm can be divided into three parts -

Algorithm 2 $RZ - COHEN - Update$

1: **Input:** Graph $G(V, E)$, $|V| = n$, $|E| = m$, Edge $(src, dest)$, $src, dest \in V$
2: **Data structures:** Reachability Set $REACH(s_i)$
3: **Algorithms:** $BFS(G, u)$: Returns the nodes in the BFS traversal of G' with u as the source
4: **for** $s \in S$ **do**
5: **if** $src \in REACH(s)$ AND $dest \notin REACH(s)$ **then**
6: Add $BFS(G', dest)$ to $REACH(s)$
7: **end if**
8: **end for**

Algorithm 3 $RZ - COHEN - Query$

1: **Input:** $REACH(s), \forall s \in S$, $u \in V$
2: **if** $u \in s_i$ **then**
3: RETURN i
4: **else**
5: RETURN -1
6: **end if**

Algorithm 4 $RZ - COHEN - AssignLE$

1: **for** $v \in V$ **do**
2: $minIndex \leftarrow min_{i,1 \leq i \leq n}(QUERY(s_i, v)$
3: $LE(v) \leftarrow [(1 + \epsilon)^{minIndex-1}, (1 + \epsilon)^{minIndex}]$
4: **end for**

- Initialization of data structures
- Update of data structures
- Querying

Initialization of data-structures:

We initially read the graph in memory and assign a random value to each vertex. The graph is reversed. The vertices are then assigned a label by using a random permutation of the values assigned to them. In our algorithm we sort the vertices by value assigned to obtain a ranking.

We use a dictionary to maintain the $log_{1+\epsilon}(n)$ reachability trees. For some vertex $v \in V$, the vertex is added to one of the reachability sets based on the label assigned to it. Along with v all vertices that v can reach in the reverse graph are also added to the reachability set. The vertices that v can reach are found by running a BFS on the reverse graph with v as the source node of the BFS.

Update of data-structures:

Italiano's incremental data structure to maintain transitive closure has been used to update the reachability sets. Every time an edge (v, w) is added to the graph, we add the reverse edge (w, v) to the reverse graph. Reachability sets containing w but not v are updated by adding the descendants of v in the reverse graph to them. The descendants are obtained by running a naive graph traversal algorithm on the reverse graph with v as the source. This may seem like an expensive operation, but not all reachability sets need to be updated, and the operation is dependent on various factors such as the presence or absence of the end points of the edge in a particular reachability tree, the number of descendants of v in the reverse graph, and the number of reachability trees that have to be updated.

Query:

A query $query(v, u_i)$ returns true if $v \in REACH(u_i)$, and false otherwise. Once the reachability sets are populated, for a vertex $v \in V$ we find the smallest u_i which returns true for $query(v, u_i)$. The value i can be used to estimate the smallest label that can be assigned to

v

4.2 Experimental evaluation

4.2.1 Purpose

Experiments were designed to answer the following questions.

- How efficient is the algorithm?
- How well does the algorithm scale with increase in input size?
- How does the update operation affect the running time of the algorithm?

4.2.2 Data sets

We have run experiments on directed graphs of different sizes, which are summarized in Table 4.1. We use networkx to generate relatively small directed graphs with upto 500000 nodes. These are randomly generated, and don't represent any real world data. We also consider larger graphs with over 900K nodes from CINET, which represent synthetic populations, and a Twitter follower network from Venezuela.

4.2.3 Execution

- The graph was read from a file and stored in memory using Networkx.
- Random vertices of the graph were chosen, and an edge was introduced between them.
- The reachability size of each vertex was recorded was estimated, and recorded. At the same time a histogram was created to record the distribution of the sizes of the reachability sets of all vertices in the graph.

Table 4.1: Description of Data collected

No. of Nodes	No.of Edges	Source
10000	9999	Networkx
50000	49999	Networkx
100000	99999	Networkx
500000	499999	Networkx
approx. 905000	approx. 1M	NDSSL
approx 1.1M	2M	NDSSL
approx. 1.4M	approx. 1.7M	NDSSL
approx 1.8M	5M	NDSSL
approx 2.2M	8M	NDSSL
3.6M	36M	NDSSL
4.8M	48M	NDSSL
4.85M	70M	LiveJournal
31.6M	80M	NDSSL

- The reachability sizes were estimated using an ϵ value of 0.9.
- The times taken for initialization, update and query were also recorded to study the efficiency and scalability of the algorithm.

4.2.4 Results

Timing Information:
We recorded the time taken for reading the graph, initializing the data structures, and populating them. We also recorded the time taken for updating the reachability trees when edges were added to the graph. Queries are made in constant time. We have recorded the time taken to compute the distribution of the reachability trees of the vertices in a graph. The timing information is seen in Figure 4.1. We notice that the algorithm scales well for graphs with upto 500K vertices. For graphs with more than 500K vertices, we notice a steep incline in the times for populating the data structures and running queries. This algorithm does not scale well for graphs of larger sizes.

An edge insert operation has many consequences. Updating reachability trees may have different times depending on the endpoints of the edge being added. Factors that influence updating reachability trees are -

- Presence or absence of the end-points of the edge in a reachability set.
- Number of descendants of the destination vertex of the edge that have to be added to the reachability set.

In order to study the effects of the edge insert operation on updating the reachability trees that we are maintaining. We added random edges to the graphs and recorded the time taken to update the reachability trees being maintained. The number of edges added differed for graphs of different sizes. We also recorded the ratio of the update time to the running time of the algorithm when the graph was static. The observations made can be seen in Figure

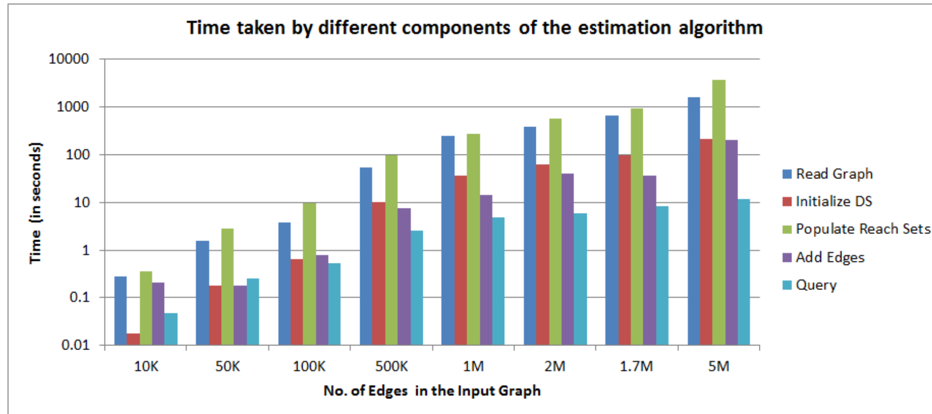

Figure 4.1: Timing: The figure represents the time taken to perform different subroutines in the algorithm.

4.3. We have also compared the running times of the sequential algorithm in the static and dynamic contexts. We can observe from Figure 4.2 that adding edges to the graph doesn't really affect the overall execution time of the algorithm.

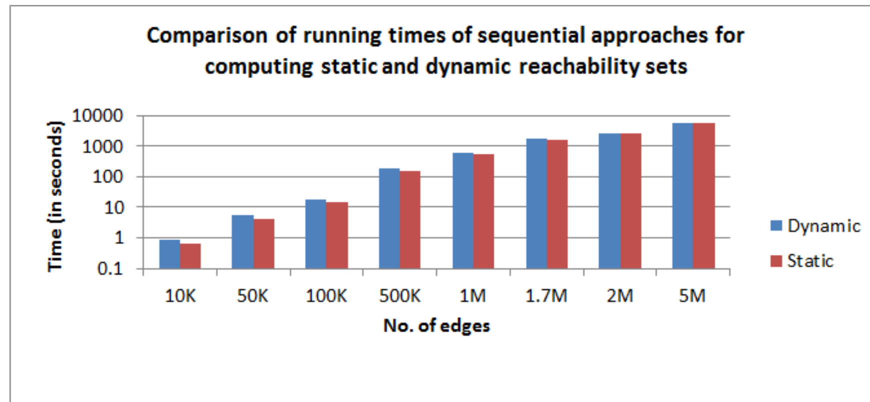

Figure 4.2: Timing: Comparison of running times of the sequential algorithm in the static and dynamic settings.

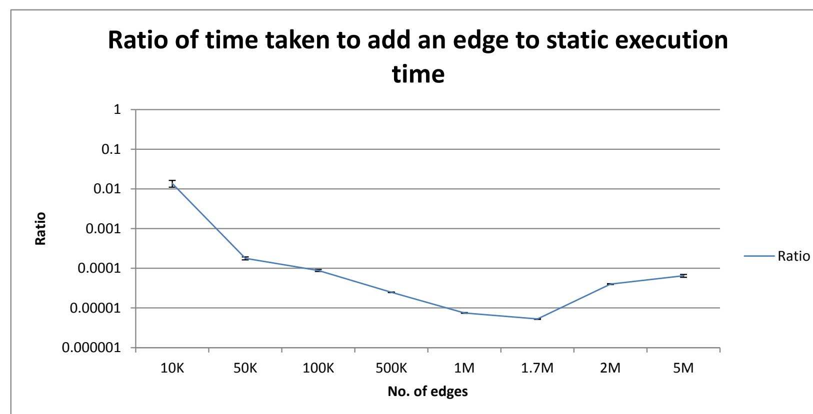

Figure 4.3: Timing: The ratio of update time to the running time in the static setting.

Chapter 5

Dynamic reachability using Giraph

A parallel algorithm was designed by Cohen for the reachability set size problem, while Chapter 4 gives a sequential dynamic algorithm for the DRS-Inc(ϵ, δ) problem. We now study how to combine these and obtain a parallel solution to the DRS-Inc problem. Specifically, we will explore methods to parallelize the the approach of Roditty et al. [1] discussed in Chapter 4. Recall that their approach reduces the DRS-Inc(ϵ, δ) problem to reachability by making use of $\log(n)$ external sinks. In order to parallelize this, the reachability information from the sinks has to be updated dynamically when the graph changes.

We explore Apache Giraph, a distributed graph processing framework for this problem, because of its very general vertex programming framework for parallel algorithms. We first discuss the programming model in Giraph briefly, and then describe our algorithm for DRS-Inc(ϵ, δ). We also describe a Giraph algorithm for finding reachability set sizes in decremental graphs DRS-Dec(ϵ, δ).

5.1 Background

Giraph is a Pregel-like graph processing framework developed by the Apache community. Giraph runs on the Hadoop File System (HDFS), and makes use of a message passing implementation to implement graph algorithms over a distributed framework. Giraph works in iterations called supersteps. In a superstep every vertex can receive messages, run computations and send messages. The model adopted by Giraph can be used to implement a wide range of graph processing algorithms. The programming model of Giraph is very simple. Giraph is implemented in Java. The main function a vertex is supposed to perform in every superstep is defined in the "compute" method. Variables can assigned, computations can be performed, messages can be sent or received in the compute method. A skeleton Giraph algorithm is illustrated in Algorithm 5. Giraph provides other methods such as *preSuperstep()* and the *postSuperstep()* to define the functionality of a vertex before and after each superstep. A detailed description of all methods that can be used in Giraph can be found in [30].

Along with the message passing properties, Giraph also allows to modify the structure of the graphs by adding or removing edges and vertices of the graph. Therefore, Giraph is highly suited for implementing a solution to estimate the sizes of reachability sets on large dynamic graphs.

Challenges of using Giraph

The sequential approach to solving the DRS problem makes use of reachability sets for the external sinks in the algorithm described in [1]. The Giraph programming model has the following salient features

- Giraph has a vertex centric programming model.
- Data can be stored in vertices of the graph.
- Vertices can send and receive messages.

Algorithm 5 Giraph Sample

```
public Class Foo extends BasicComputation(I,V,E,M) {
        public void preSuperstep() {
                //optional
        }
        public void compute(Vertex<I,V,E> vertex, Iterable<M> messages) {
                .
                .
                //Initialization of variables being used
                for (message : messages) {
                        // do something
                }
                sendMessageToAllVertices(vertex, message to be passed);

                vertex.voteToHalt();
        }
        public void postSuperstep() {
                //This method is optional
        }
}
```

- Algorithms are executed by vertices in iterations.
- The use of external data structures may have an adverse effect on the performance of the algorithm.

The Giraph programming model makes it difficult for us to use external data structures such as sets to maintain reachability information of the vertices. We can thus store the reachability information of each vertex in the vertex itself. The id of the lowest sink that can reach a vertex is stored in the vertex itself. The lowest sink id is propagated along the edges of the graph thus correctly answering the query of which sink's reachability set a vertex belongs to.

For e.g. the traditional implementation of Dijkstra's algorithm to find a single source shortest paths makes use of a priority queue to update the distance of the vertices from the source. It is unfeasible to use a shared memory heap to implement the priority queue on the Giraph framework. Therefore, the Giraph implementation maintains the distance of each vertex from source in the vertex itself. And this information is propagated along the edges of the graph, and is used to update the distance of every vertex from the source in subsequent

iterations of the Giraph algorithm. At the end of the algorithm, each vertex in the graph will contain in it the shortest-path distance from the source.

Giraph algorithms run in multiple iterations across different machines. Therefore, traditional methods of evaluation cannot be used to evaluate Giraph algorithms. Klauck et al. [41] discuss evaluation strategies for dynamic graph algorithms. We study their approaches to determine whether their approaches can be applied to evaluate Giraph algorithms. They evaluate the performance of the algorithm in terms of ”rounds” and provide bounds to a set of well known graph algorithms. Rounds can be compared to the iterations of the giraph algorithm. Klauck et al. in [41] prove that the lower bound for computing the minimum spanning tree is $\tilde{\Omega}(\frac{n}{k})$ and the upper bound is $\tilde{O}(\frac{n}{k})$. They also prove that the lower and upper bounds for the connectivity verification problem is $\tilde{\Omega}(\frac{n}{k})$ and $\tilde{O}(min(\frac{n}{k}, \frac{m}{k^2} + D\Delta k))$ where D is the diameter of the graph and Δ is the maximum degree. The authors say that $\tilde{O}$ and $\tilde{\Omega}$ are used to hide the polylog(n) terms involved in each round of the algorithm. However, there hasn't been much research carried out in dynamic graph algorithms on Giraph like frameworks.

5.2 Parallel and dynamic algorithm for DRS-Inc(ϵ, δ) in Giraph

We have leveraged the message passing capability of Giraph to implement a version of Roditty et al.'s algorithm as a solution to the DRS-Inc(ϵ, δ) problem.

In Giraph, the vertices are capable of storing information. We discussed the need to maintain the reachability data structure implicitly. In our approach, the relevant reachability information is stored in the vertex itself. Let the set of sinks described in [1] be denoted by $S = \{s_1, s_2, ..., s_k\}$. We store the sink indices corresponding to multiple iterations of the al-

gorithm as a list in the vertex. For each node v, $\forall j, 1 \leq j \leq q$, we store $min_{1 \leq i \leq k}\{s_i^j | s_i^j \rightsquigarrow v\}$ where q is the number of iterations you want to run the algorithm.

This way, we are inherently maintaining the reachability sets required by the algorithm. As in Chapter 4, we describe different components of the algorithm. Algorithm 6 describes the initialization component. Algorithm 7 describes the algorithm to propagate vertex information within the graph. Algorithm 8 describes the algorithm to add an edge to the graph.

Algorithm 6 DRS-Giraph Initialize

1: **Input:** Graph $G(V, E)$, $|V| = n$, $|E| = m$, $\epsilon > 0$, No. of iterations $= k$
2: **Variables:** Lowest label that can be assigned to a vertex - Least Element LE, boolean flag *changed*, origValue, $IDSET$ - The set of sink ids stored in the vertex
3: $\forall v \in V$ do in parallel:
4: **if** $superstep.id == 0$ **then**
5:     **for** $i \leftarrow 1$ *to* k **do**
6:         Generate a rank for the vertex
7:         Generate an index $index, 1 \leq index \leq log_{1+\epsilon}n$ based on the rank
8:         Add index to $IDSET$
9:     **end for**
10:     Store the $IDSET$ in origValue. Set the value of the vertex to be $IDSET$
11:     Send the value to all out neighbors
12: **end if**

1 In superstep 0, generate a series of ranks for every vertex in the graph. Based on the ranks generated, assign a series of index values $i, 1 \leq i \leq log_{1+\epsilon}n$ to the vertex. For a vertex v, the series of index values is stored in $IDSET(v)$. $IDSET(v)$ corresponds to the sink ids. in $RZ - COHEN$.

2 In subsequent supersteps, check the values that a vertex receives from its incoming neighbors. If the current value of the vertex is greater than the minimum value among

Algorithm 7 DRS-Giraph Propagate

1: **for** $MESSAGE \in INCOMING_MESSAGES$ **do**
2:     **for** $i \leftarrow 1$ *to* k **do**
3:         **if** $IDSET[i] > MESSAGE[i]$ **then**
4:             $IDSET[i] = MESSAGE[i]$
5:             $changed = TRUE$
6:         **end if**
7:     **end for**
8: **end for**
9: **if** $changed = TRUE$ **then**
10:     Send $IDSET$ to all out neighbors
11: **end if**

Algorithm 8 Add Edge

1: Add (u, v) to the graph.
2: Pass $IDSET(u)$ to v
3: Call Propagate

all the messages that it receives, then update the value of the vertex with the minimum value, and propagate the updated value along all its outgoing edges. i.e. For a vertex v, that receives messages from a set of neighbors $IN(v)$, for every $val(v_i) \in IDSET(v)$, and every $val(u_i) \in IDSET(u), \forall u \in IN(v)$,if $val(v_i) > min_{u \in IN(v)} val(u_i)$ then $val(v_i) \leftarrow min_{u \in IN(v)} val(u_i)$. $IDSET(v)$ is thus populated with the updated values of v_i. The new value of $IDSET(v)$ is propagated along all its outgoing edges.

3 When an edge (u, v) has to be added to the graph, call the Giraph method to add the edge (u, v), pass the current value stored at u along (u, v) and continue the supersteps.

4 Once all the supersteps are complete, every vertex $v \in V$ will contain a list of lowest sink ids that can reach v in every iteration of the algorithm.

5 The voteToHalt() method is called to stop a vertex from sending messages along its edges when its value has not been updated.

Lemma 1. *Algorithm 8 correctly solves the* DRS-INC(ϵ, δ) *problem with amortized query time of* $O(1)$*, amortized update time* $O(\log(\frac{1}{\delta}))$ *and* $O(n \log(\frac{1}{\delta}))$ *space.*

Proof. Cohen in [17] states that when the number of iterations of the algorithm is $O(\epsilon^{-2} \log(n))$, with a probability $1 - O(\frac{1}{poly(n)}) = 1 - \frac{1}{\delta}$, for every $v \in V$, $\frac{||S(v)| - \hat{s}(v)|}{|S(v)|} \leq \epsilon$. In our algorithm, we store the sink ids corresponding to all k iterations of the algorithm in the vertex itself. Our space requirement is there $O(nk)$. When we choose $k = O(\epsilon^{-2} \log(\frac{1}{\delta}))$, the space requirement is proved to be $O(n \log(n))$. Queries can be answered merely by examining the value stored in a vertex. Therefore a query can be answered in $O(1)$ time.

We prove the correctness of the Giraph algorithm by drawing parallels to the proof of correctness of the algorithm in Chapter 4. In the sequential algorithm, for every sink $s_i \in S$ we maintain a reachability set of s_i. We estimate the reachability set size of a vertex v by finding $min_{1 \leq i \leq k}(s_i) | v \in REACH(s_i)$.

In the Giraph approach we initialize every vertex with some sink id s_i that can reach it. Each vertex then passes this id along all its edges. In each superstep, every vertex reads

the incoming sink ids. If the incoming sink id is lower than the current sink id stored in the vertex, then the value of the vertex is updated to the lower sink id. Therefore, by construction, the lowest sink id is stored in each vertex in every superstep. When an edge is added, we merely propagate the sink id along the new edge. Therefore, the algorithm is correct even in the incremental setting.

We now prove that the amortized update time of the algorithm is $O(1)$.

- Let the value stored in a vertex v be denoted by $val(v)$
- When an edge (u, v) is added to G, the $val(v)$ is updated only of $val(u) < val(v)$
- Consider a path of vertices $V' = \{v_0, v_1, v_2, ..., v_k\}, V' \subset V$
- Let $val(u) < val(v_0)$
- Before the edge (u, v_0) is added, assume that $\forall v_j \in V'$. The series of edge insertions $(v_i, v_j), 0 \leq i \leq j \leq k, val(v_i) > val(v_j)$ is not charged as it does not involve updating $val(v_j)$.
- Now, when an edge (u, v_0) is added, the values of all $v_i \in V'$ have to be updated. Therefore, one edge addition is charged k. The total cost of adding an edge is amortized over at least k operations.
- Therefore, the amortized time taken for one update operation is $O(1)$
- Since we need to update sink ids corresponding to $\log(\frac{1}{\delta})$ iterations, the overall running time of the update operation is $\log(\frac{1}{\delta})$

□

5.3 Computing maximum and minimum reachability set size incrementally

In some applications, we just need statistics about the reachability set sizes, e.g., the maximum, $\max_{v\in V} |S(v)|$, the minimum $\min_{v\in V} |S(v)|$ or the distribution. We discuss how Algorithm 8 can be modified to compute the maximum and minimum easily in Algorithm 9.

Algorithm 9 Compute Max. and Min. Reachability Set Sizes

1: Add meta-sinks M to the graph.
2: Connect all vertices to M
3: Pass the sink id currently stored in the vertex to M
4: The lowest sink-id stored in M is used to estimate $max_{v\in V}(\hat{s}(v))$
5: The highest sink-id stored in M is used to estimate $min_{v\in V}(\hat{s}(v))$

Lemma 2. *Algorithm 9 correctly returns a $(1 \pm \epsilon)$ approximation to the maximum and minimum reachability set size using $O(n \log n)$ space and $O(1)$ time*

Similar to the running time analysis described above, by choosing the number of iterations $k = O(\epsilon^{-2} \log(\frac{1}{\delta})$, the space requirement is determined to be $O(n \log(n))$

5.4 Parallel algorithm for DRS-Dec(ϵ, δ) in Giraph

We have solved the problem for approximating dynamic reachability set sizes for decremental graphs using Giraph. Similar to the incremental setting, we store the set of sink ids corresponding to multiple iterations within the vertex itself. The challenge lies in updating the values stored in vertices when an edge is removed from the graph.

Approach

- We initialize the value of each vertex with the sink indices corresponding to multiple iterations of the algorithm as a list in the vertex. For each node v, $\forall j, 1 \leq j \leq q$, we store $min_{1\leq i\leq k}\{s_i^j | s_i^j \rightsquigarrow v\}$ where q is the number of iterations you want to run the algorithm.

- For every vertex, store the initial list of sink ids.

- Propagate the sink ids along the edges as described in Algorithm 7.

- When an edge is removed, reset the values of all vertices to their initial list of sinks and restart the propagation procedure.

The initialization procedure is described in Algorithm 10. The algorithm to propagate sink ids is described in Algorithm 7. The algorithm for updating the reachability sets when an edge is removed is described in Algorithm 11.

We will look at the empirical performance of the Giraph solution to DRS-Dec(ϵ, δ) as we were not able to obtain theoretical bounds better than a naive solution to DRS-Dec(ϵ, δ).

5.5 Experimental evaluation

5.5.1 Purpose

We study the following questions:

- What effect does adding an edge have on the running time of the algorithm?

- How well does the algorithm scale to input data sets of larger sizes?

- How does the Giraph implementation compare to the sequential implementation of the algorithm?

Our experimental setup is as described in the mid-semester milestone.

Algorithm 10 DRS-Dec-Giraph Initialize

1: **Input:** Graph $G(V, E)$, $|V| = n$, $|E| = m$, $\epsilon > 0$, No. of iterations $= k$

2: **Variables:** Lowest label that can be assigned to a vertex - Least Element LE, boolean flag *changed*, origValue, $IDSET$ - The set of sink ids stored in the vertex, $INITSET(v)$ - The initial set of sink ids generated for a vertex $v \in V$

3: $\forall v \in V$ do in parallel:

4: **if** $superstep.id == 0$ **then**

5:     **for** $i \leftarrow 1$ *to* k **do**

6:         Generate a rank for the vertex

7:         Generate an index $index, 1 \leq index \leq log_{1+\epsilon}n$ based on the rank

8:         Add index to $IDSET$

9:         Assign $IDSET$ to $INITSET(v)$

10:     **end for**

11:     Store the $IDSET$ in origValue. Set the value of the vertex to be $IDSET$

12:     Send the value to all out neighbors

13: **end if**

Algorithm 11 Remove Edge

1: Remove (u, v) from the graph.

2: $\forall v \in V$ reset the set of sinks in v to $INITSET(v)$

3: Call Propagate

5.5.2 Cluster Specifications

The cluster set-up is as described in the project proposal. We have used the ”shadowfax” cluster of VBI. Each compute node in the cluster has 12 Intel Xeon processors. Each compute node has 48G memory. For the experiments run until this stage of the project, we have used upto 32 compute nodes to set up Hadoop. We have used the Torque queuing system to submit giraph jobs.

5.5.3 Hadoop Setup

Shadowfax is a shared cluster. Therefore, it is not possible to have a dedicated set of compute nodes for the Hadoop setup. We are using myHadoop [42] to provision Hadoop on the fly on the Torque queuing system. Each time we provision Hadoop, we have a dedicated Hadoop cluster for 48 hours. Currently, we provision upto 32 nodes every time we want to use Hadoop. Depending on the size of the input data required for future experiments, we can provision a Hadoop cluster on more nodes.

Giraph was setup following instructions in [30]. The maven packaging tool is being used to compile the source files being used for the experiment.

5.5.4 Data sets

We have used the same data set as in the Sequential approximation algorithm. The data set is described in Table 4.1

5.5.5 Experimental Setup

- The input graph was copied to HDFS. The class used to read the input to Giraph was modified to reverse the input graph.
- Giraph jobs were written to solve the DRS-Inc and DRS-Dec problem.
- The sizes of the reachability sets were then estimated, and a distribution of the sizes of the reachability sets was also computed.
- The experiment was run on graphs of varying sizes, as seen in Table 4.1. The times taken for running the experiment on different sizes of graphs have been collected.

5.5.6 Results

We have compared the times taken by the static and dynamic algorithms. We notice that the performance of the incremental algorithm is very similar to that of the static algorithm. And the incremental algorithm scales well with increase in size of the number of input edges. The observations made are seen in Figure 5.1

We know that Giraph exhibits high latency. From the Figure 5.1 we can notice that for graphs containing upto 1.7M edges, the running time is almost the same. This is because Giraph takes some time to setup the job. With an increase in size of the input data set, the time taken for execution does not increase as quickly as traditional parallel programming paradigms . For smaller data sets, the overall running time of the algorithm is dominated by the setup time. We also know that algorithms implemented in Giraph scale well with increase in size of input data. We observe the effect of the setup time in Figure 5.2.

We compare the time taken to add an edge and update the values of vertices in the graph with the overall running time of the distributed Giraph algorithm. We observe that the time taken to add an edge is much lesser when compared to the overall running time of the algorithm on Giraph. This observation can be seen in 5.3. Similarly we measure the ratio of

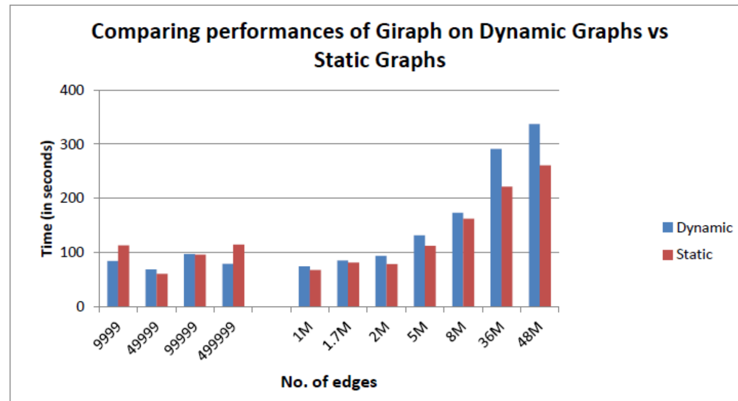

Figure 5.1: Comparison of time taken in the static and dynamic settings

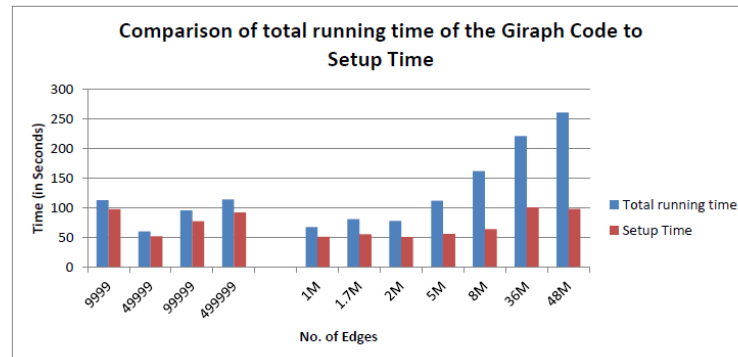

Figure 5.2: Time of execution of one iteration of the approximation algorithm on data sets of different sizes.

the time taken to remove an edge and update the graph to the overall running time of the Giraph algorithm. The observation for the decremental setting can be seen in Figure 5.4.

Giraph is scalable for large graphs. Giraph does not take much time for processing large graphs. Whereas the sequential algorithm is inefficient for large graphs. This observation is recorded in Figure 5.5.

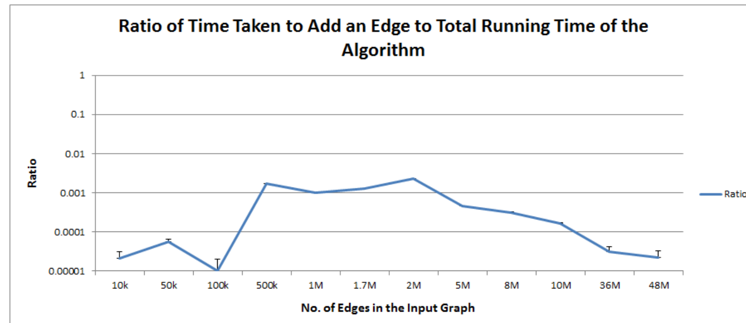

Figure 5.3: Ratio of time taken to add an edge to total running time of the Giraph algorithm

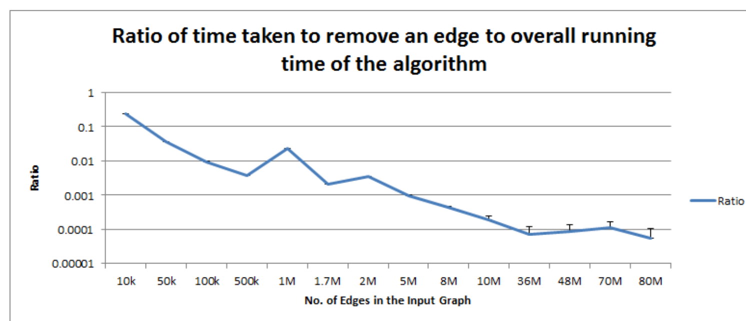

Figure 5.4: Ratio of time taken to remove an edge to total running time of the Giraph algorithm

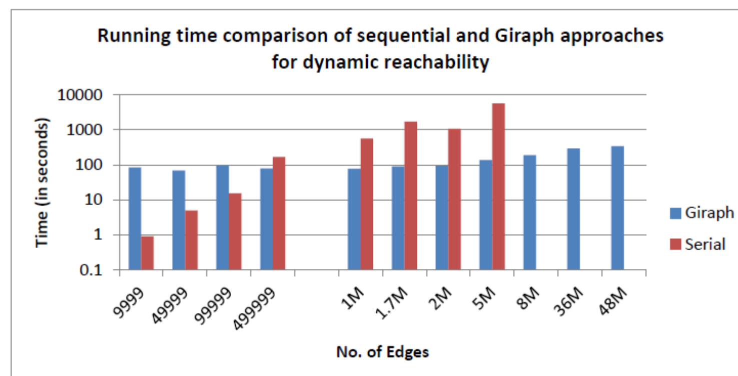

Figure 5.5: Comparing execution times of Giraph and the sequential algorithms

Chapter 6

Future Work and Conclusions

6.1 Future Work

We have designed solutions to estimate the sizes of reachability sets in large dynamically growing graphs. We are exploring decremental algorithms to estimate the sizes of reachability sets in graphs when edges or nodes are deleted from them. Roditty et al. in [1] have said that for DAGs, maintaining a decremental data structure for estimating sizes of reachability sets takes $O(m)$ worst case running time. We are looking at the following approach to extend estimation of sizes of reachability sets to decremental graphs -

- For the sake of convenience, let us assume that removing edges will not disconnect the graph.
- When edges are removed it is difficult to update reachability sets without knowing the structure of the graph.
- If the graph structure is not maintained in the reachability set, it might be necessary to traverse the graph in order to update the reachability set.
- Italiano in [37] proposes an algorithm that takes $O(mn)$ time for a sequence of edges

for DAGs.

- We are looking at maintaining reachability information for regular graphs as well. Therefore we cannot utilize the algorithm presented in [37]. We are trying to make use of a labeling scheme where we pick some random nodes in the graph, and make them meta-sources. A graph is induced using these meta-sources. Therefore, when an edge is deleted, the reachability sets can be updated efficiently by looking at the meta-sources and the component to which the source and destination vertices of edge belong to.
- On similar lines, Bramandia et al. in [13] have propose algorithms for incremental maintenance of 2-HOP labels to answer reachability queries of graphs. We aim to adapt their approaches to design a solution for decremental maintenance of data-structures to estimate the sizes of reachability sets of vertices of a graph.

6.2 Conclusion

We studied different methods to estimate sizes of reachability sets in dynamic graphs. A lot of the literature that we studied focused on answering reachability queries for graphs, and maintaining data structures to efficiently answer reachability queries for dynamic graphs. $RZ - COHEN$, The algorithm proposed by Roditty et al. in [1] combined the techniques of answering reachability queries in dynamic graphs with techniques to estimate the sizes of reachability sets in graphs. By running our experiments we determined that updating reachability sets can be done efficiently. Therefore $RZ - COHEN$ is a good solution to estimating the sizes of reachability sets of dynamic graphs.

But, we found that using a sequential algorithm does not scale well for large graphs. Therefore we adapted the approximation algorithm to Giraph. We found that the implementations in Giraph scale well for large graphs. We thus conclude that using Giraph to estimate sizes of reachability sets on dynamic graphs is an efficient, effective and scalable solution.

Chapter 7

Bibliography

[1] L. Roditty and U. Zwick, "Improved dynamic reachability algorithms for directed graphs," in *Foundations of Computer Science, 2002. Proceedings. The 43rd Annual IEEE Symposium on.* IEEE, 2002, pp. 679–688.

[2] Z. Bar-Yossef, R. Kumar, and D. Sivakumar, "Reductions in streaming algorithms, with an application to counting triangles in graphs," in *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2002, pp. 623–632.

[3] S. Arifuzzaman, M. Khan, and M. Marathe, "Patric: A parallel algorithm for counting triangles and computing clustering coefficients in massive networks," Network Dynamics and Simulation Science Lab., Virginia Tech, Tech. Rep. 12-042, July 2012.

[4] D. Bader, S. Kintali, K. Madduri *et al.*, "Approximating betweenness centrality," in *Proc. of Workshop on Algorithms and Models for the Web-Graph (WAW2007)*, ser. LNCS, vol. 4863, 2007, pp. 134–137.

[5] D. Bader and K. Madduri, "Parallel algorithms for evaluating centrality indices in real-world networks," in *Proc. of Int. Conf. on Parallel Processing (ICPP)*, 2006.

[6] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[7] U. Brandes and T. Erlebach, "Network analysis: Methodological foundations," 2005, lecture Notes in Computer Science Tutorial, Springer-Verlag.

[8] C. Barrett, K. Bissett, J. Chen, X. Feng, V. S. A. Kumar, and M. Marathe, "Epifast: A fast algorithm for large scale realistic epidemic simulations on distributed memory systems," in *Proc. of Int. Conf. on Supercomputing (ICS)*, 2009.

[9] M. Newman, "The structure and function of complex networks," *SIAM Review*, vol. 45, pp. 167–256, 2003.

[10] H. Yıldırım, V. Chaoji, and M. J. Zaki, "Grail: a scalable index for reachability queries in very large graphs," *The VLDB JournalThe International Journal on Very Large Data Bases*, vol. 21, no. 4, pp. 509–534, 2012.

[11] T. Reps, "Program analysis via graph reachability," *Information and software technology*, vol. 40, no. 11, pp. 701–726, 1998.

[12] M. Yannakakis, "Graph-theoretic methods in database theory," in *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems.* ACM, 1990, pp. 230–242.

[13] R. Bramandia, B. Choi, and W. K. Ng, "On incremental maintenance of 2-hop labeling of graphs," in *Proceedings of the 17th international conference on World Wide Web.* ACM, 2008, pp. 845–854.

[14] M. Faloutsos, P. Faloutsos, and C. Faloutsos, "On power-law relationships of the internet topology," in *ACM SIGCOMM*, vol. 29, no. 4. ACM, 1999, pp. 251–262.

[15] D. Coppersmith and S. Winograd, "Matrix multiplication via arithmetic progressions," *Journal of symbolic computation*, vol. 9, no. 3, pp. 251–280, 1990.

[16] C. Tantipathananandh, T. Berger-Wolf, and D. Kempe, "A framework for community identification in dynamic social networks," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '07. New York, NY, USA: ACM, 2007, pp. 717–726. [Online]. Available: http://doi.acm.org/10.1145/1281192.1281269

[17] E. Cohen, "Size-estimation framework with applications to transitive closure and reachability," *Journal of Computer and System Sciences*, vol. 55, no. 3, pp. 441–453, 1997.

[18] G. F. Italiano, "Amortized efficiency of a path retrieval data structure," *Theoretical Computer Science*, vol. 48, pp. 273–281, 1986.

[19] S. Chandan, S. Saha, C. Barrett, S. Eubank, A. Marathe, M. Marathe, S. Swarup, and A. K. Vullikanti, "Modeling the interactions between emergency communications and behavior in the aftermath of a disaster," in *The International Conference on Social Computing, Behavioral-Cultural Modeling, and Prediction (SBP)*, Washington DC, USA, April 2-5 2013.

[20] C. Barrett, K. Channakeshava, F. Huang, J. Kim, A. Marathe, M. Marathe, G. Pei, S. Saha, B. Suppiah, and A. Vullikanti, "Human initiated cascading failures in societal infrastructures," *PLoS ONE*, vol. 7, no. 10, 2012.

[21] C. Barrett, K. Bisset, J. Leidig *et al.*, "Economic and social impact of influenza mitigation strategies by demographic class," *Epidemics*, vol. 3, no. 1, pp. 19–31, 2011.

[22] K. Atkins, C. Barrett, R. Beckman, K. Bisset, J. Chen, S. Eubank, A. Feng, Z. Feng, S. Harris, B. Lewis *et al.*, "An Interaction Based Composable Architecture for Building Scalable Models of Large Social Biological, Information and Technical Systems," *CTWatch Quarterly*, vol. 4, no. 1, 2008.

[23] A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener, "Graph structure in the web," *Computer networks*, vol. 33, no. 1, pp. 309–320, 2000.

[24] N. Pržulj, "Biological network comparison using graphlet degree distribution," *Bioinformatics*, vol. 23, no. 2, p. e177, 2007.

[25] J. Leskovec, A. Singh, and J. Kleinberg, "Patterns of influence in a recommendation network," *Advances in Knowledge Discovery and Data Mining*, pp. 380–389, 2006.

[26] J. Leskovec, M. McGlohon, C. Faloutsos, N. Glance, and M. Hurst, "Cascading behavior in large blog graphs," *arXiv preprint arXiv:0704.2803*, 2007.

[27] L. A. N. Amaral, A. Scala, M. Barthelemy, and H. E. Stanley, "Classes of small-world networks," *Proceedings of the National Academy of Sciences*, vol. 97, no. 21, pp. 11 149–11 152, 2000.

[28] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec, "Hadi: Mining radii of large graphs," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 5, no. 2, p. 8, 2011.

[29] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data.* ACM, 2010, pp. 135–146.

[30] A. G. D. Community. (2013, Aug.) Apache giraph. [Online]. Available: http://giraph.apache.org/

[31] J. Borge-Holthoefer, A. Rivero, and Y. Moreno, "Locating privileged spreaders on an online social network," *Physical Review E*, vol. 85, no. 6, p. 066123, 2012.

[32] K. Lerman and R. Ghosh, "Information contagion: An empirical study of the spread of news on digg and twitter social networks." *ICWSM*, vol. 10, pp. 90–97, 2010.

[33] H. Baumann, P. Crescenzi, and P. Fraigniaud, "Parsimonious flooding in dynamic graphs," *Distributed Computing*, vol. 24, no. 1, pp. 31–44, 2011.

[34] D. Gregor and A. Lumsdaine, "Lifting sequential graph algorithms for distributed-memory parallel computation," *SIGPLAN Not.*, vol. 40, no. 10, pp. 423–437, Oct. 2005. [Online]. Available: http://doi.acm.org/10.1145/1103845.1094844

[35] J. Barnat, P. Bauch, L. Brim, and M. Ceska, "Computing strongly connected components in parallel on cuda," in *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, 2011, pp. 544–555.

[36] P. Flajolet and G. Nigel Martin, "Probabilistic counting algorithms for data base applications," *Journal of computer and system sciences*, vol. 31, no. 2, pp. 182–209, 1985.

[37] G. F. Italiano, "Finding paths and deleting edges in directed acyclic graphs," *Information Processing Letters*, vol. 28, no. 1, pp. 5–11, 1988.

[38] M. Henzinger and V. King, "Fully dynamic biconnectivity and transitive closure," in *Foundations of Computer Science, 1995. Proceedings., 36th Annual Symposium on*, Oct 1995, pp. 664–672.

[39] D. Frigioni, T. Miller, U. Nanni, and C. Zaroliagis, "An experimental study of dynamic algorithms for transitive closure," *Journal of Experimental Algorithmics (JEA)*, vol. 6, p. 9, 2001.

[40] H. Yildirim, V. Chaoji, and M. J. Zaki, "Grail: Scalable reachability index for large graphs," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 276–284, 2010.

[41] H. Klauck, D. Nanongkai, G. Pandurangan, and P. Robinson, "The distributed complexity of large-scale graph processing," *arXiv preprint arXiv:1311.6209*, 2013.

[42] S. Krishnan. (2012) myhadoop. [Online]. Available: http://www.sdsc.edu/us/consulting/myHadoop-SDSC.pdf