Large Scale Centrality Measures in Apache Flink and Apache Giraph

by

Janani Chakkaradhari

Submitted to the Department of Computer Science and Electrical Engineering
in partial fulfillment of the requirements for the degree of
Masters in Information Technology for Business Intelligence
at the
TECHNISCHE UNIVERSITÄT BERLIN

September 2014

©Technical University of Berlin

Author .................................................................
Department of Computer Science and Electrical Engineering
August 10, 2014

Certified by .........................................................
Sebastian Schelter, Research Associate / PhD student
Thesis Advisor

Certified by .........................................................
Prof. Dr. Markl, Volker
Thesis Supervisor
Large Scale Centrality Measures in Apache Flink and Apache Giraph

by

Janani Chakkaradhari

Submitted to the Department of Computer Science and Electrical Engineering on August 10, 2014, in partial fulfillment of the requirements for the degree of Masters in Information Technology for Business Intelligence

Abstract

Centrality measures are used to assess the structural properties of network entities. For example, it can be used to rank nodes for a recommender systems. The recent advance in measuring centralities for very large networks has set off a host of new computation methods that exploit the sparse nature of real world networks. Furthermore, an increasing interest in network analysis resulted in a plethora of graph centered parallel programming models which consequently demands a choice between different computing platforms for computation intensive problems. Among the computing platforms available, the open source distributed parallel frameworks Apache Flink (a general purpose distributed data processing platform) and Apache Giraph (a specially designed platform for graph processing ) iteratively are considerably noteworthy for their wide support of graph algorithms.

This thesis aims to elucidate a systematic comparison of both platforms by leveraging previous work on measuring scalable centralities in which the conventional closeness and betweenness measures are redesigned (as Effective Closeness and LineRank algorithms respectively) to reinforce embarrassingly huge, parallel frameworks. More specifically we address the issue of programmability with a comprehensive study on the modeling of complex algorithms in these platforms, and the performance of these systems with diverse experiments. The scalable version of these centrality measures are implemented in Apache Flink as well as in Apache Giraph and run on real graph datasets with millions of edges to evaluate their relative effectiveness.

We benchmark the results and present the novel findings from them. It can be observed that the vertex centric model becomes restrictive for modeling complex computations as in the LineRank algorithm whereas Flink is useful and appropriate in such cases by offering a declarative style programming interface. On the other hand, Apache Giraph performs better relative to Flink in most cases and Apache Flink performs equivalently better on large sparse graphs, as the numbers of computations are reduced by the early detection of converged nodes.
Acknowledgments

I would like to express my deep sense of gratitude to my advisor Sebastian Schelter for his excellence guidance, scientific advice and constant encouragement throughout this thesis work.

I would like to thank Prof. Dr. Volker Markl and Dr. Ralf-Detlef Kutsche for their kind co-ordination efforts. I would also like to thank Robert Metzger for his kind assistance in cluster set up and his useful advises on using Apache Flink and finally I also would like to thank Stephan Ewen and all the staff members of DIMA group for their kind advice and clarifications on understanding the platform.

I express my sincere thanks to all the staffs and professors from IT4BI programme for their help and support during my Masters degree.

I am grateful to thank all my classmates from IT4BI program for their support, unforgettable memories and kind assistance since the beginning of the IT4BI Masters degree.

Last but by no means least; I want to thank my wonderful family especially my mom, dad, my two sisters, my brotherin-laws and my nephew for their round-the-clock encouragement during the thesis.
Chapter 1

Introduction

Networks are everywhere! Networks, ranging from small protein interaction in our body cells to large collections of web pages in the Internet, are inherently complex in their structure. Several studies investigating interdisciplinary networks have been designed, in hopes of finding interesting phenomena [48, 47, 15, 11, 7] in the structure of the networks. In particular, identifying the most critical (central) node in a network is a fundamental concept in network analysis and mathematically these are measured by a set of indices called centrality measures (or rankings) [8, 7]. The most commonly used measures of this category are degree, closeness and betweenness [20]. In simple terms, degree is number of immediate neighbors of a node (e.g., direct friends in social network), closeness determines how well a node is connected within its local group and can reach most of them in very few steps and betweenness is a score that says to what degree a given node bridges two or more different groups, and so intuitively also the amount of information flow on this node.

1.1 Motivation

The techniques for finding centrality measures in a network are more than 60 years old. In 1958 [20], centrality measures were applied to understand the rationale behind the integration of diverse political systems in Indian social life. This experiment revealed highly cohesive network centers (closeness in local) that engender a coor-
dinated (betweenness) structure in their diverse society. In social media, targeted advertising (only to the most influential people in the group positioned close to others) is an invasive and less expensive tactic for spreading messages across the network, requiring minimal resources and effort\(^1\).

Betweenness centrality have been used to detect the components with high possibility of failures (most traversed edges) in a power-grid which in turn helps to take immediate actions to prevent a cascading power grid failures in real time energy systems [28]. Valdis Krebs’s study on the 9/11 attack unleashed the power of network centralities that revealed the covert leadership network of hijackers. Also, he identified the spearhead of the network with a high score of closeness and betweenness indices [32]. Additionally centrality measures are widely used in diverse domains like organizational networks [6], product network analysis [38], predicting polls [22] and protein interaction networks [27].

With an increasing use of emerging technologies via Internet, an enormous amount of graph data is being generated in day-to-day life. Gartner\(^2\) reported that the five dominant sources of massive amount of graph data generators in the world of Internet are Amazon, Apple, Facebook and Google. This shows a clear increasing trend towards the benefit of centralities in social media marketing, ranking numerous product recommendations and in web search. The rapidly increasing space of new opportunities in marketing and the demand for finding useful information in various fields speaks to the importance of finding centralities of large-scale networks.

The real world networks are extremely huge in size, so that the number of vertices and edges counts in the billions. Thus employing the standard version of centrality algorithms on these large networks doesn’t scale up due to their cumulative complexity in time and space requirements [30]. The tension between the ever growing size of data and the demand for retrieving meaningful insights from them, engendered the emergence of a wide variety of Big Data technologies. A notable breakthrough in Big Data is the introduction of MapReduce paradigm [12], a contemporary pro-

---

\(^1\)http://www.digitaltonto.com/2011/how-to-approach-social-influence/

\(^2\)https://www.gartner.com/doc/2081316
gramming platform for processing data based on a simple method called divide and conquer. Intuitively, it is analogous to dividing a given problem into multiple tasks and aggregating them to get the final solution more quickly.

Over the past 10 years, since the data processing community embraced MapReduce as a computing platform, several innovations and improvements have been evolving in this space. This results in a wide variety of ubiquitous frameworks that grounded on the de-facto concept of MapReduce paradigm for large-scale data processing such as Apache Hadoop\(^3\), HaLoop (an extension on Hadoop) [10], Twister [43] and more recently Apache Spark [51] and Apache Flink (formerly known to as Stratosphere) [4]. Apache Hadoop is mainly used for data processing that involves ETL tasks (like select and aggregate) and also has a loose support for parallel graph processing. In contrast to Hadoop, the other systems (HaLoop, Twister, Apache Spark and Apache Flink) have an extensive support for iterative graph processing in addition to the support for generic data analysis. On the other hand, a new set of frameworks are emerging into this space that are specially designed for processing graphs. The most popular graph computing platforms are Pregel [37], GraphLab [35] and Apache Giraph\(^4\).

In essence, the current trend is overwhelmed with plenty of large-scale graph processing frameworks. However, though this resolves the problem of scalability in standard centrality measures, it still cannot be executed on parallel execution models, because it is only suitable for sequential executions. For most likely explanation of this fact is that the computing methods were not designed with parallel architectures in mind. This essentially necessitates a fundamental redesign of the standard centrality algorithms [30].

There are quite a few works on parallelizing the classic centrality algorithms that exploits the distributed parallel computing architecture [13, 30]. Among these researches, the work by Ukang et al is notable for its insightful algorithms proposed in their paper [30]. This thesis work is inspired by the paper [30] for measuring closeness and betweenness in very large networks. Thanks to the novel algorithms, Effective

---

\(^3\)hadoop.apache.org/

\(^4\)http://giraph.apache.org/
Closeness and LineRank efficiently measure closeness and betweenness, respectively. The primary advantages of using these algorithms to measure scalable centralities are as follows:

1. The author efficiently utilizes the sparse propensity of billion scale real world networks for computing betweenness measure. He proposed a novel approach to compute the importance score of edges, in which he efficiently splits the computation of incoming and outgoing edges of a node. If the original graph is sparse then the matrices that represents the incoming and outing edges will also be sparse in a way that it contains exactly m non-zero elements (m is the total number of edges) 1 for each row. Intuitively this reduces the space complexity of the standard algorithm from quadratic to $O(m)$.

2. The utilization of a memory efficient technique for counting the number of neighbors while computing the closeness measure. The time complexity of conventional closeness is cubic while for Effective Closeness it is reduce to $O(dm)$ where d is the diameter of the graph and m is the total number of edges in the graph.

1.2 Goal

The large volume of graph data posits two main issues in front, when we want to compute quickly and effectively, 1) the algorithms (the depth of complexities) and 2) the computing platform on which we execute the selected algorithm [16]. Because our goal is to measure centralities and their applications in very large networks, we try to answer the following questions, which focus on the experimental evaluation of large-scale distributed parallel frameworks:

1. What are the challenges in implementing and evaluating a computation intensive centrality algorithms such as Effective Closeness and LinRank on recently available large-scale data processing platforms?
2. Which of these systems do better in terms of scalability for complex graph algorithms?

3. How well does the programming model of the data processing system fit the algorithms?

We decided to choose two versatile computing platforms such as Apache Flink and Apache Giraph. In this thesis we implement and evaluate scalable centrality measures such as Effective Closeness and LineRank algorithms on these platforms for large-scale graphs.

1.3 Outline

The outline lays out the approach that we followed for the thesis that subsequently helped us to find the answers of these questions mentioned in the previous section.

- A comparative study on the existing large-scale distributed computing platforms and providing reasonable evidence for choosing Apache Flink and Apache Giraph as the platforms for evaluations (Chapter 2)

- A comprehensive study of the mathematical graph theory and the scalable centrality algorithms (Chapter 3)

- Implementation of scalable Effective Closeness and LineRank algorithm for measuring closeness and betweenness centralities in both the platforms (Apache Giraph and Apache Flink) (Chapter 4)

- Experimenting and evaluating the scalability and the performance of these two computing platforms by executing the implemented code on diverse data sets ranging from few thousands to millions of edges (Chapter 5)

- A brief overview of the related works on centrality algorithms (Chapter 6)

- We conclude by discussing the results and findings.
Chapter 2

Distributed Graph Processing

Any form of fundamental interaction can be modeled as graph for mainly investigating the behavioral properties that exists among the interacting elements. The demonstrative nature of graphs motivates their usage in various fields. Network of things are everywhere and with the advantage of newly emerging technologies the connectivity grows tremendously.

In 2003, Newman classified the real world networks based on their complex structure into four groups such as social networks (e.g., Facebook, Twitter), information networks (e.g., Google, citation network), biological networks (e.g., interactions between proteins , network of genetic regulatory system) and technological networks (e.g., Internet, transportation routes, electric power grid) known to grow increasingly enormous ever since the beginning [39] which surpasses the processing capacity of a single computer.

The lack of resources and processing capabilities lead to new directions in the field of computing and eventually evolved with distributed parallel processing for managing large amount of data. MapReduce [12] programming model is the first of its kind introduced by Google in 2004 for embarrassingly parallel processing of data on cheap clusters. Apache Hadoop\(^1\) is an open source implementation of MapReduce framework.

Over a decade, MapReduce paradigm grows in to a top notch approach and

\(^1\)hadoop.apache.org/
adopted by many leading industries to run data intensive workloads. There are two main reasons for the fast acquaintance of MapReduce as a computing platform, 1) Ease of expressing data transformations and aggregations with two simple functions known as Map and Reduce 2) It allows the user to focus on writing logic while the runtime system handles data partitioning, scheduling, fault-tolerance and coordination of nodes in the cluster [12].

Moreover new parallel computing frameworks are emerging over time by taking advantage of MapReduce abstraction but they differ by adding many new features. In this section we will discuss about two categories systems emerging systems in the context of support features to iterations [49].

1. Generic Data Parallel Systems: Systems that supports commonly used data transformation operators as well as iterative graph processing,

2. Graph Parallel Systems: Specially designed systems for graph processing.

2.1 Generic Data Parallel Systems

A considerable amount of literature has been published on MapReduce that are widely used for mainstream scientific computing. In this section we will describe briefly about the existing parallel data flow systems with their support for recursive data processing facilities.

2.1.1 Apache Hadoop

Functional programming is the base for MapReduce paradigm. The generic map and reduce methods are second order functions that consumes an input data set (or a portion of it) and a user defined function (UDF) that is applied to the corresponding input data. This novel idea inspires in automatic parallelization on a large distributed cluster brings about the combined effort of computational power and memory [1].

The iterations in Hadoop are achieved by simply invoking a single MapReduce job recursively from an external driver program. This is so nave in the way it executes
the iterative computation on the results of individual MapReduce jobs sequentially [10]. Although MapReduce is highly successful in horizontal scale to billions of data, previous studies have reported the limitations of Hadoop and the most crucial issues that draws the attention in graph computational view are listed below

- First, the flexibility of highly customizable UDF restricts the opportunity of optimization in the code by the underlying system, [1]

- Secondly, lack of inherent support of join operator that is frequently used in many iterative algorithms like PageRank. This imposes on user to hand code the logic for handling heterogeneous inputs as MapReduce is designed for single input

- Last but not least, the lack of inherent support of iterations in Hadoop compels to write an external program to repeatedly invoke the MapReduce jobs which in turn incurs increased IO latency and serialization issues and there is no explicit support for specifying termination conditions [10].

Regardless of above mentioned limitations, MapReduce is an extremely efficient framework such that no other system (e.g., databases) can shatter its performance for which the system is specialized at (such as data warehousing tasks).

2.1.2 HaLoop

In recent years, researchers have investigated a variety of approaches [50, 10, 44] to incorporate iterative functionality into Hadoop data flow model. Among these here will discuss briefly about HaLoop, noticeable system in the hierarchy. HaLoop is a variant of Hadoop by offering multi-step MapReduce job and targets to the reuse the loop invariant data across the iterations by incorporating efficient caching and scheduling mechanisms. One major drawback of Hadoop’s naive implementation is it

\( ^2 \)http://users.cis.fiu.edu/~lzhen001/activities/KDD_USB_key_2010/forms/tutorials.htm

\( ^2 \)referred from tutorial on Large-scale Data Mining: MapReduce and Beyond at KDD 2010 Workshop

\( ^3 \)http://users.cis.fiu.edu/~lzhen001/activities/KDD_USB_key_2010/forms/tutorials.htm
reflects each pair of map and reduce call as a separate job instead HaLoop deliberates a set of all map and reduce pairs as a single job.

In addition, the loop aware scheduler redirects the intermediate results of iterations to the same physical machines. This way HaLoop is proven to incur only 4% of data shuffling during iterative MapReduce phase. The running time is shown to improve by 1.85 \times \text{ than Hadoop} [10]. Although caching at different stages eliminates the overhead in loading and shuffling data during iteration, HaLoop presumes the user to specify the usage level of caching (for example map input cache or reduce input cache) depending on the design of graph algorithm.

2.1.3 Twister

Twister is yet another framework built mainly to support iterations using MapReduce paradigm [17]. It avoids repeated loading of static data in iteration. It reads the input data from local disks and process in distributed shared memory setup and imposes the user to split the large data files before initiating the job. It follows stream based architecture for data flow where the output of mappers is directly pointed to reducers by providing a long running parallel MapReduce tasks. Twister is exposed to perform 80x faster than Hadoop [43]. In-memory distributed cache in Twister makes the system to not scale for very large data and thereof suffers to provide concrete fault-tolerance in case of failures [24].

In addition to these systems, there is a recent study on constructing iterations as first class citizens in Hadoop MapReduce framework [44] along with runtime optimizations.

2.1.4 Apache Spark

Apache Spark contributes similar level of abstraction as MapReduce but emphasizes in-memory data processing on cluster [51]. The high-level goal here is to provide a distributed shared memory abstraction by exploiting the concept of Resilient-Distributed Datasets (RDD). RDDs are immutable collection of records and it generates a log
called data lineage that represents a set of transformation on the dataset and uses this lineage for reconstructing of RDDs in case of failures. Conceptually RDDs are identical to views in a database in which Apache Spark preserves logging of operations (lineage) and maintains fault tolerance in a larger level where as the later allows fine grained updates to track consistency. Moreover, it delivers a richer set of in-built operators for data manipulations such as join; group-by or reduce-by in addition to standard map and reduce [51].

It has a simple interface and the complex computations are specified in few lines of code using Scala. Iterations are invoked by using for loop structure in Scala. In essence, a new RDD is created for loop variant data set during each iteration. Since data are in memory, Spark is proven to perform 30× faster than Hadoop. Recently, a graph library (GraphX) is added on top of it that addresses graph specific issues such as graph partitioning and computations using Resilient Distributed Graphs (RDG), RDD for graphs.

The excessive usage of memory restricts its usage in a cluster with limited memory and resources but a recent developments show to support out of core operations in case of memory deficiency Update for Out-of core. Spark allows its user to explicitly indicate what data to cache in memory in the data flow pipeline. Joshua et al [44] argues that it turns out to be hard for the user to determine these performance related parameters in multi-user cluster as the configurations changes frequently.

2.1.5 Apache Flink

Apache Flink has more advantages over Hadoop and other existing parallel execution models Dyrad and Hyracks for large scale data analysis. This framework consists of two parts: a PACT compiler and a Nephele Execution Engine [4]. The PACT compiler is a data processor driven by second order functional abstractions like MapReduce for data parallelism. It essentially targets declarative programming model independent of execution strategy and becomes the natural place for automatic optimizations (it offers different kinds of optimizations which try to minimize the amount of data shuffling, formulating optimized data processing pipeline). The Nephele Engine works
on data processing pipeline represented via data flow graphs and offers scheduling, resource allocation, fault-tolerance, task communication and execution monitoring services in parallel systems [4].

Apache Flink provides a specially designed operator for handling algorithms that are recursive and alleviates the lack of looping competency in parallel data flow systems. Unlike Giraph, Flink integrates iteration in the existing data flow model where the former framework solely dedicated for managing iterative algorithms that operates on structural graph networks. Flink user can express the iterative part of data processing as simple as loop statements along with exit conditions similar to any other programming or declarative languages. Thus it evolves with a unified solution by incorporating several in built second order functions in addition to map and reduce. The other data transformation operators currently supported by Flink includes FlatMap, Join, Reduce on grouped dataset, GroupReduce, Cross, CoGroup, Union, Bulk iterations, Delta iterations and many others which can be referred at API Guide.

It is not feasible to perform a complete data analysis task that consists of iteration in the pipeline by only using Giraph. It requires Hadoop to do typical analysis tasks and Giraph for iterations. This necessitates a fine tuned orchestration amongst these frameworks and thus turns out to be fragile for scaling out systems [18].

**Delta-Iteration Operator**

Most of the graph and Machine Learning algorithms repeatedly perform same computation to evolve into a fine-tuned result set. Let us consider a popularly known example of PageRank algorithm [29]. It works on a recursive matrix-vector multiplication where the vector is turns out to be the page ranks after considerable number of multiplications and the matrix represents transition probabilities of pages. This is formally referred to as power method that approximately determines the dominant eigenvectors or page ranks [34]. Moreover, in power method the matrix is often said to be associated with a constant data path and remains the fixed source input of iteration. On the other hand, the vector is associated with dynamic data path where the values are updated during every iteration. Based on this mathematical
understanding, the algorithms are designed in such a way that the result of current iteration (updated vector) is fully fed back as an input of the next iteration (in the dynamic input path). Advanced research in PageRank computation [29] reveals an interesting pattern in convergence rate of pages in power iteration. The pages with highest rank values are supposed to have less converge rate, on the other hand the pages with high convergence rate are presumed to have low ranks. This novel concept of prior detection of early converging pages which subsequently alleviates the rank re-computation of non changing values in the next iterations. This is experimentally proven to improve the computation speed in remaining iterations of power method. Later, this concept turns out to be one of the factors in optimizing graph mining algorithms. Flink exploits the sparse computational dependency of graph algorithms by integrating an incremental iteration. It also extends to identify the already converged nodes in the iteration and try to not re-compute its value in the next iterations [18, 29]. The concept of delta iteration is realized incrementally in Flink.

- Delta iterations keep on maintaining two data sets. One holds the output of every iteration and the other detain the input of next iteration in contrast to conventional method where both are assumed to be same.

- These two data sets are termed as work set and solution set right after the iteration initialization. The work set represents result of the current iteration and also it is the input of the next iteration hence can be termed as partial solution.

- The solution set consists of set of nodes that are converged. Initially the work set and solution set are initialized with the same values. But only the work set is undergoing iteration’s compute method updates and then will be compared with solution sets for convergence.

- If it converged then the node is added in the solution set and it will be removed from work set hence the converged nodes are not anymore involved in the next iteration’s computation [18]. The solution set is updated by producing a delta for it, which is merged into the solution set at the end of each iteration step.
Apache Flink currently has no support for fault tolerance. If there is a failure in any of the task managers then the user has to restart the entire job. The experiments conducted on Apache Flink are considered to have full availability of all the workers. Apache Flink proven to perform equivalent with Giraph and better than spark for incremental iterations [18].

2.2 Graph Parallel Systems

It has been realized from the studies [36] that the inherent structure of graph increases the data access in computations compared to data transformation tasks. This part of the discussion will provide a brief description of large scale graph processing systems dedicated for graph mining algorithms.

2.2.1 Pregel

In 2010, Google published the Pregel [37], a new computing model for graph processing at very large scale. The fundamental concept of this model is derived from Bulk-Synchronous Parallel (BSP) [46] commonly known for parallel computation and the working structure is illustrated in the Figure 2-1. The key idea of BSP model grounds on three main operations,

1. Local computations at individual processor level (aka workers)
2. Exchanging messages between the processors at cluster level
3. A stage of synchronization that waits for all the parallel processors to complete and decides either to signals the start of next round of execution or terminate based on certain conditions.

One complete parallel execution of all these three phases is termed as a superstep or iteration. In Pregel, a recursive execution of these supersteps represents the iterative computation of graph algorithms. Vertices in a graph are considered to be
the first-class citizens and hence each vertex is associated a user defined function (step function/update function) that mainly contains the logic of graph algorithms.

The UDF updates the current state of a vertex (for example it computes the new value of a vertex in each superstep). This vertex centric design imposes a presumed format on input data to including the details of a vertex, its outing and incoming edges depending on the graph algorithm. During each superstep, a vertex receives information from its neighbors along the incoming edges that are sent to them in the previous iteration (in case of PageRank the received values are usually aggregated).

Then the vertex executes the step function to re-compute its value based on the received neighbor values in the current iteration and at the end it sends the updated value to its neighbor nodes which are the input for the next successive iteration. In addition to that, any vertex can request to stop once it finishes its computation based on algorithm’s requirements goes to in-active state.

At the same time the vertex can be activated by receiving messages. As similar MapReduce framework, the Pregel model also has a controlling master node and a set of worker nodes that are actual processor. In principle, a worker is allocated with a partition of input data that contains a group of vertices. For the purpose of fault-
tolerance the master instructs the workers to save the state of their partition and this frequency of check pointing is user controllable.

### 2.2.2 Apache Giraph

Apache Giraph is a loose implementation of Pregel system in Java. It is known for the quote "**think like a vertex**" \(^4\). Giraph uses Hadoop as its run-time system. The independent local computation in parallel is analogous to Hadoop’s map only job as illustrated in the Figure 2-2. Each worker in the figure is assigned with set of vertices and for the each vertex it runs the Hadoop’s map task by maintaining the data in memory. Recent up gradations moves from an in-memory scalable system to out-core support system **Apache Giraph**. Any graph algorithm in Giraph should implement a "Vertex" class to inherit the features of vertex’s functionalities such as sending messages, removing and adding of vertices and edges. The logic of the step function is specified inside the "**compute**" method.

The curse of slow job is a known problem in this model in which the slowest processor in the cluster determines the speed of the execution \([35]\). Facebook’s contribution to the open source Giraph project inculcated a number of optimizations to

\(^4\)http://www.slideshare.net/Hadoop_Summit/processing-edges-on-apache-giraph

\(^a\)http://www.slideshare.net/Hadoop_Summit/processing-edges-on-apache-giraph
the original system. In a report, it is evident that Giraph is performing 10× faster than conventional Hadoop.

### 2.2.3 Distributed GraphLab

In principle, GraphLab [35] is an asynchronous graph processing system unlike BSP it changes the state of the graph while updating by providing a novel framework to maintain the consistency across the graph network. Unlike Giraph, the data is associated with both vertices and edges. When a vertex scheduled to execute an update function, it reads required information (regarding the neighbors), updates its vertex value and decides to reschedule for the vertex based on the user specified termination condition. Moreover, the order of vertex to be updated is decided by the scheduler. GraphLab tend to perform 20 to 60× faster than Hadoop. Though GraphLab is proven to have faster and efficient runtime performance, it suffers from non-deterministic execution due to asynchronous execution. Furthermore, it is considerably hard to use the system for complex graph implementations [49].

### 2.3 Comparison of Large Scale Parallel Processing Systems

Table 2.1 summarizes the ground covered in this chapter, essentially targets to compare a required set of features for running computation intensive graph algorithms. We claim the following reasons for choosing the systems Apache Flink and Apache Giraph to implement and evaluate large scale centrality algorithms.

As shown in Table 2.1, there is an obvious similarity between the two systems Apache Hadoop and HaLoop. For the reason that LineRank algorithm (described in section) is fundamentally centered on many join operations, it is likely to be an overhead to implement the algorithm with HaLoop as well as in Hadoop.

---

5 https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920

6 http://www.slideshare.net/sscdotopen/introducing-apache-giraph-for-large-scale-graph-processing
Even though Twister is a considerable platform for the computation intensive algorithms, it has been identified \(^7\) that MR task and the environment setup is time consuming and highly complex.

Furthermore, both Apache Spark and GraphLab are based on shared memory abstraction in their own way. Nevertheless Spark can be used to develop graph algorithms, there is no evidence in the system to identify early convergence rate of vertices which is a common pattern during power iteration in large sparse graphs. Hence it incurs excessive data movement during iterations. On the other hand, GraphLab is the only asynchronous platform in the list and the model itself is different form MapReduce paradigm. In addition to that as mentioned earlier, it imposes the user to specify priority of the vertices that is later used for optimization at runtime.

<table>
<thead>
<tr>
<th>Framework</th>
<th>Apache Hadoop</th>
<th>Hadoop</th>
<th>Twister</th>
<th>Apache Spark</th>
<th>Apache Flink</th>
<th>Apache Giraph</th>
<th>Distributed GraphLab</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Model</td>
<td>MR</td>
<td>MR</td>
<td>MR</td>
<td>MR</td>
<td>FACT-MR</td>
<td>Map Only</td>
<td>GraphLab</td>
</tr>
<tr>
<td>Synchronous/Acynchronous</td>
<td>Out of core</td>
<td>Out of core</td>
<td>Distributed in-memory</td>
<td>Distributed shared memory (RDD)</td>
<td>Out of core</td>
<td>Out of core</td>
<td>Sequential Shared memory</td>
</tr>
<tr>
<td>Ease of Use</td>
<td>Simple but tricky for joins</td>
<td>Simple but tricky for joins</td>
<td>Simple for iterations</td>
<td>Simple</td>
<td>Simple</td>
<td>Simple depend on algorithm</td>
<td>Hard for complex iterations</td>
</tr>
<tr>
<td>Generic/Special Purpose</td>
<td>Generic to ETL</td>
<td>Iterative algorithms</td>
<td>Iterative algorithms</td>
<td>Generic to ETL, Support for ML, and iterative algorithms</td>
<td>Generic to ETL, Support for ML and first order construct to support iterations</td>
<td>Designed for iterative tasks but also fits into linear algebraic expressions</td>
<td>Specially designed to support ML and graph algorithms</td>
</tr>
<tr>
<td>Runtime optimization</td>
<td>NA</td>
<td>Static data caching</td>
<td>Static data caching</td>
<td>Static data caching in iteration</td>
<td>Emulates algebraic optimizations in UDFs, caches static data path</td>
<td>Estimates memory based on graph data</td>
<td>Optimises order of vertices execution</td>
</tr>
<tr>
<td>Fault Tolerance</td>
<td>Excellent Fault Tolerance</td>
<td>Same as Hadoop</td>
<td>NA</td>
<td>The data lineage is rebuilt to create new RDD for the failed task</td>
<td>NA, but supports optimistic recovery for fixed point iterative algorithm</td>
<td>Checkpoints iteration's results at fixed intervals and it is configurable</td>
<td>Distributed checkpoint both synchronous asynchronous fashion at fixed intervals</td>
</tr>
<tr>
<td>Storage</td>
<td>HDFS</td>
<td>HDFS</td>
<td>Local Disks</td>
<td>HDFS</td>
<td>HDFS</td>
<td>HDFS</td>
<td>HDFS</td>
</tr>
<tr>
<td>Native Lang.</td>
<td>Java</td>
<td>Java</td>
<td>Java</td>
<td>Scala</td>
<td>Java</td>
<td>Java</td>
<td>C++</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of Large Scale Distributed Parallel Processing Systems in the context of support for iterative algorithms

The most significant reasons for selecting Apache Flink and Apache Giraph:

1. The execution component of both the systems is Java Virtual Machine (JVM) and therefore the comparison will be easy as they have same runtime environment.

\(^7\)http://dsp.acm.org/view_lecture.cfm?lecture_id=6523

\(^7\)referred from the lecture slides
2. If the compute method in Apache Giraph, encodes the algorithm in way to explicitly invoke vote to halt option, then intuitively it is roughly equivalent (disregarding other factors like partitioning and hashing techniques) to delta-iteration in Apache Flink. So in this way we argue that it is reasonable to compare both the systems in the perspective of computation execution time.

3. In contrast if there is no explicit specification of vote to halt option in Apache Giraph, then it is again roughly equivalent to bulk iteration in Apache Flink.

4. Loosely speaking, by choosing Apache Giraph and Apache Flink, we consider to compare platforms that belongs to two different application domains (generic data and specific graph)

On the basis of the aforementioned points, we determined our platforms (Apache Flink and Apache Giraph) to implement the closeness and betweenness algorithms. In the next following chapter, we turn our direction to discuss about the centrality measures and their mathematical explanations.
Chapter 3

Scalable Centrality Measures

The major objective of this chapter is to layout a detailed study on graph theoretical terms and the concepts used throughout this thesis followed by an attempt to illustrate the intuition behind the centrality measures such as closeness and betweenness. Finally, the parallel version of these measures, Effective Closeness and LineRank algorithms [30] are discussed understand the computations in detail.

3.1 Graph Theoretical Background

A connected set of objects are collectively termed as Graphs. Formally such a relationship among objects are denoted as \( G = (V, E) \) where the set of pair inside braces represents the set of vertices \( V \) (or nodes or points) and edges \( E \) (or line or connection) [14]. Graphs are inherently simple to express many real world problems. For instance road map is a commonly used graph representation of locations and the paths connecting them to reach the desired location. Here the locations are vertices and connecting paths are edges.

In this thesis, we refer the term network synonymously with graph. Graph theory has got its root from trying to solve a problem (seven bridges of Knigsberg\(^1\)) and so from then onwards numerous complex problems are solved by many graph algorithms postulated by researchers from various domains. In computational science, graphs

\(^1\)http://en.wikipedia.org/wiki/Seven_Bridges_of_K%C3%B6nigsberg
are usually symbolized as a matrix. Informally matrix can be defined as a two di-
mensional representation of the interaction between objects/entities and turned out
to the natural way of expressing graph data [3]. The **Adjacency matrix** $A$ of the
example graph 3-1 is shown below,

\[
A = \begin{bmatrix}
1 & 2 & 3 \\
1 & 0 & 1 & 1 \\
2 & 1 & 0 & 1 \\
3 & 1 & 0 & 0
\end{bmatrix}
\]

### 3.1.1 Transition Probability Matrix

The notion of transition probability is derived from novel concept of Markov Chain.
Takis Konstantopoulos describes Markov Chain in simple terms as a random pro-
cess in which the probability distribution of future only depend on present state [41].
Mathematically such random process is characterized by a state space and an associ-
ated transition matrix. The transitions are the changes of the states of the process\(^2\)
With respect to our example, the vertices represent the state space and the corre-
sponding transition probability matrix is given below 3-1. If we take a closer look at
both example graph and matrix $P$, it clear that a process at vertex 1 (in the column)
can change its state either to vertex 2 or vertex 3 and hence it is assigned with equal
probabilities of transitions. The matrix $P$ is a stochastic matrix where the sum of the

\(^2\)http://en.wikipedia.org/wiki/Markov_chain
columns equals to 1.

\[
T = \begin{bmatrix}
1 & 2 & 3 \\
1 & 0 & 0.5 & 1 \\
2 & 0.5 & 0 & 0 \\
3 & 0.5 & 0 & 0.5 & 0
\end{bmatrix}
\]

### 3.1.2 Stationary Distributions

Markov Chain (stochastic or random process) is stationary if for any time the joint distribution of states in the current process is the same as the joint distribution of the states in the previous process [45]. This corollary is recognized to have significant effect in many of the graph algorithms (e.g., PageRank). Consider the following example as an attempt to emphasize the stationary distribution of Markov Chain and moreover Markov processes are considered to be mathematically equivalent to random walks on directed graphs [25].

If for any process (let us say an actor walking on the example graph 3-1) consider that the process is stationary and if we know the stationary probability of vertex 3 is 1/5. Then after 200 random walks, we should expect that during 40 of those walks the actor might have visited vertex 3 [45].

This stationary distribution of random walks on a given network is commonly referred as eigenvector centrality (or PageRank) and the corresponding eigenvalue is 1. The most commonly used method to find the stationary probability distribution of large graphs is power iteration method.

### 3.1.3 Power Iteration

Power iteration (or power method) produce a value and a non-zero vector called dominant eigenvalue and eigenvector respectively for a given matrix A [34].

\[
X_{i+1} = AX_i/norm(AX_i)
\]  
(3.1)
The formulation in 3.1 is executed for number of times until the vector $X_{i+1}$ converges. Power iteration usually converges on the largest eigenvector. Throughout this thesis we refer to the term “convergence” if the $L^1$ – Norm $|L1Norm|$ of the vector in the iteration (i.e $X_{i+1}$ ) is less than a certain threshold.

The computation logic of LineRank algorithm (betweenness centrality) is based on the power iteration method. So it is useful to recall the well-known PageRank algorithm with our example graph, a small network of 3 pages 3-1 and the corresponding adjacency matrix A. Consider a surfer randomly starts to visiting the nodes of the graph. In order to know where the surfer ends up after 1 step is simply estimated by multiplying the transition probability matrix $P$ with an initial random vector $(X_{i+1})$. In power method this multiplication is performed iteratively to find the probability of the surfer to position steady state vector which is the dominant eigenvector of the matrix $A$.

### 3.1.4 Column Normalization

In power iteration method, if matrix is non column stochastic then the vector resulted from matrix-vector multiplication in the iteration has to be normalized. Eigenvectors are usually normalized in such a way that the sum of the squares of the each element in the vector is equal to 1. This is also known to as rescaling of eigenvector to unit length. In linear algebraic way, a vector can be unit normalized by first calculating its length and then dividing each of its elements by its length [26].

So far this chapter has focused on providing a strong theoretical background of graphs and the associated mathematical concepts that are frequently used in graph algorithms for computation. The remaining part of this chapter discusses about the two centrality measures closeness and betweenness.

### 3.2 Centrality Measures

Centrality measures are regarded as an index of each individual vertex in a network of connected things. The index can reveal some unknown insights relative to the
position of a node in a network. It helps to find how close a node to all other nodes in a graph, how fast information can be spread within and across a network and the most influential vertex in a network.

3.2.1 Closeness

Closeness can be used to estimate the time taken to spread information from a vertex of interest to all other vertices in a network. It determines the reach of a vertex within a network. Closeness naturally gives high scores to nodes that are near the center of local ties. In turn it can be used to identify most powerful influencer within a local community.

Several forms of closeness measure are defined in the past since 1950 [21] and the most commonly used form of closeness measure based on geodesic paths defined by Beauchamp [5] is referred for this thesis work. For a given graph $G$, the closeness of a vertex $v$ is defined as reciprocal of the average of shortest distances to all other vertices and mathematically it is denoted as,

$$C_c(v) = \left( \frac{\sum_{u \in V} d(v, u) / n - 1}{n-1} \right)^{-1}$$  \hspace{1cm} (3.2)

where $C_c(v)$ is the inverse of the average shortest distance between $v$ and all other vertices in the graph $G$. $n-1$ represents the minimum sum of distances for a vertex that is adjacent to all other vertices.

Closeness centrality is grounded on all-pairs shortest paths algorithm. It is the inverse function of farness centrality. Average of shortest distances to every other reachable node from a source node is measured as Farness. Consider a network shown in figure 3-2 as a concrete example for describing closeness measure.

The shortest path distance of a node relative to other reachable nodes in the example network is computed for all possible pairs of nodes and the corresponding layout of matrix of $6 \times 6$ colored in brown is displayed in table 3-3. The last three columns in the table represent the sum, farness and closeness measures for each node in example graph. It reveals that node 4 and 5 have highest closeness centrality in
the toy graph. But time complexity of this computation is $O(n^3)$ and hence it is extremely expensive if we consider it for very large graphs.

![Example graph of 6 nodes](image)

Figure 3-2: Example graph of 6 nodes

![Shortest Path Computation](image)

Figure 3-3: Shortest Path Computation

### 3.2.2 Scalable Closeness

As an effort to surpass aforementioned computationally expensive method, U Kang et al. in [30] proposed an alternative way to compute approximate closeness measure for very large networks. Instead of computing all-pairs shortest paths, the algorithm computes the number of neighbors of a given node within $r$ steps where $r : 1 \rightarrow d$ and $d$ is the diameter of the network.

This can be easily understood with an illustration shown in table 3-4 for the same example graph discussed in section 3.2.1. The part of table 3-4 displayed in dark green represents for each node the number of nodes in shortest paths of length 1, 2 and 3. Here the diameter is 3 hence the matrix with $6 \times 3$ dimension is formulated. The column *sum* is computed by multiplying each cell with its corresponding length and aggregating the resulting value for each row. The last two columns follow the same procedure as described in the previous section. We can notice that the results

<table>
<thead>
<tr>
<th>Vertices</th>
<th>Sum</th>
<th>Farness (Sum/N)</th>
<th>Closeness (Farness)^-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 1 2 2 1 3</td>
<td>9</td>
<td>1.8</td>
</tr>
<tr>
<td>2</td>
<td>1 - 1 2 1 3</td>
<td>8</td>
<td>1.6</td>
</tr>
<tr>
<td>3</td>
<td>2 1 - 1 2 2</td>
<td>8</td>
<td>1.6</td>
</tr>
<tr>
<td>4</td>
<td>2 2 1 - 1 1</td>
<td>7</td>
<td>1.4</td>
</tr>
<tr>
<td>5</td>
<td>1 1 2 1 - 2</td>
<td>7</td>
<td>1.4</td>
</tr>
<tr>
<td>6</td>
<td>3 3 2 1 2 -</td>
<td>11</td>
<td>2.2</td>
</tr>
</tbody>
</table>

---

2Diameter of a graph is the longest shortest path between two vertices in the graph
of this method are similar to former shortest path computation showed in table 3-3.

| Vertices | Shortest Path Lengths (SL) | Sum $\sum_{(SL \times \text{NumNodes})}$ | Farness $(\text{Sum/N-1})$ | Closeness $(\text{Farness}^{-1})$
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7 2 1</td>
<td>$1(1x2+2x2+3x1)$ = 9</td>
<td>1.8</td>
<td>0.556</td>
</tr>
<tr>
<td>2</td>
<td>3 1 1</td>
<td>$1(1x3+2x1+3x1)$ = 8</td>
<td>1.6</td>
<td>0.625</td>
</tr>
<tr>
<td>3</td>
<td>2 3 0</td>
<td>$1(1x2+2x3+3x0)$ = 8</td>
<td>1.6</td>
<td>0.625</td>
</tr>
<tr>
<td>4</td>
<td>3 2 0</td>
<td>$1(1x3+2x2+3x0)$ = 7</td>
<td>1.4</td>
<td>0.714</td>
</tr>
<tr>
<td>5</td>
<td>3 2 0</td>
<td>$1(1x3+2x2+3x0)$ = 7</td>
<td>1.4</td>
<td>0.714</td>
</tr>
<tr>
<td>6</td>
<td>1 2 2</td>
<td>$1(1x1+2x2+3x2)$ = 11</td>
<td>2.2</td>
<td>0.454</td>
</tr>
</tbody>
</table>

Figure 3-4: Scalable Shortest Path Computation

This novel approach for computing closeness is termed as "Effective Closeness" algorithm in [30]. It attempts to computes the number of neighbors in shortest distances progressively. Therefore this method is well suited for iterative computation where the iteration is analogous to progressive shortest distances as shown in 3-4. We compute the partial sum of shortest distances in iteration for all nodes. Also the algorithm is embarrassingly parallel as the computation for each vertex in this way is independent of other node’s computation within a single iteration.

The main idea is to calculate number of nodes in shortest paths from source node in iteration. For each node, compute the number of neighbor nodes within $r$ step and get the number of neighbor nodes within $r-1$ step from the previous iteration. Subtract the later from the former to estimate the number of nodes that have shortest distance $r$ to the source node. Consider node 4 from the sample graph.

- Number of neighbor nodes within step 1 (at iteration 1) = 3
- Number of neighbor nodes within step 2 (at iteration 2) = 5
- Number of nodes that have shortest path distance 2 from 4 = 5 – 3 = 2

### 3.2.3 Flajolet-Martin Sketch

For the purpose of computing the count of neighbor in each step (iteration) for all the nodes in the network, the algorithm efficiently employs Flajolet Martin algorithm (FM Sketch). It is an approximation algorithm for finding the number of unique items in a large collection of data with a very less memory requirements of $O(\log n)$
where \( n \) is the number of unique items [19]. A brief summary of the FM Sketch is as follow,

1. The algorithm maintains a bit array of substantial length \( P \), such that \( 2^P > n \).

   In most cases \( P=32 \) is sufficient and it is also allowable to 64.

2. A hash function that hashes our vertex ids to bit or binary representation

3. The sketch maintains 1 bit for each vertex id and sets the corresponding bit depending on the number of zeros in the least significant bit (LSB).

4. Hence the count of unique vertices are approximately equivalent to the number of consecutive zeros starting from LSB towards most significant bit (MSB) until the first bit with 1 is detected.

### 3.2.4 Betweenness

Betweenness centrality single out nodes that are crucial in brokerage. Unlike closeness, it plays a vital role while rapidly spreading information across various communities. It acts as a main channel of communication in bonding two different entities and measures the amount to which a node lies on the paths between others. Betweenness centrality is the number of times a node occurs in the shortest paths of any two pair of vertices. A convention way of measuring betweenness centrality is can be found at [2].

\[
BC(v) = \sum_{s \neq t \neq v \in V} \frac{\sigma_{st}(v)}{\sigma_{st}}
\]  

(3.3)

where \( \sigma_{st} \) is the number of shortest path between \( s \) and \( t \) and \( \sigma_{st}(v) \) is the number of those shortest paths that passes through \( v \). A naive way of measuring betweenness is calculated by directly counting the number of times a node occurs in all-pair shortest paths of a network. It is illustrated in the figure 3-5 for the same example graph shown in the figure 3-2. It is obvious that node 4 has highest betweenness value that occurs many times in the shortest paths of any two pairs as compared to other nodes.
There are also other variants of shortest path betweenness computation that improvise on it [9]. Two notable variants of betweenness centrality are flow-betweenness and random walk-betweenness which considers the contributions from non-geodesic paths are given below [40]:

[Freeman]: The flow betweenness of a vertex $i$ is defined as the amount of flow through vertex $i$ when the maximum flow is transmitted from vertex $s$ to vertex $t$ averaged over all pairs of $s$ to $t$.

[Newman]: Random-walk betweenness of a vertex $i$ is equal to the number of times that a random walk starting at vertex $s$ and ending at vertex $t$ passes through vertex $i$ along the way, averaged over all possible $s$ and $t$.

The time complexity of algorithm based on single source shortest paths to compute betweenness is $O(nm + n^2 \log(n))$ [22] which is not scalable.

### 3.2.5 Scalable Betweenness

**Line Graph $L(G)$**

The standard definition of line graph can be found at the reference paper [30]. The following diagram illustrates the notion of how line graph $L(G)$ is constructed from a graph $G$.

As shown in the figure 3-6, the blue and green lines in the first graph become the nodes in the second diagram as R1 and R2 respectively. A vertex which has these blue and green lines as incoming outgoing edge then it becomes the edge in line graph $L(G)$.

In 2011 UKang et al, proposed a scalable version of Newman’s betweenness cen-
Figure 3-6: Betweenness computation using Shortest Paths algorithm

trality that is based on random walk on the line graph known as LineRank.

The main target of LineRank is to find the importance score of the incident edges of each node in the graph. Briefly it can be summarized as follows:

1. The algorithm is based on the concept of constructing a line graph $L(G)$ from a given graph $G$. In this way, the edges are translated to nodes. Then, the power iteration is employed on $L(G)$ to determine the stationary probabilities (dominant eigenvectors) of the edges.

2. In order to find the actual betweenness measure, in the final step the algorithm aggregates the stationary probabilities of the incident edges of vertices in $L(G)$.

The stationary probabilities of the vertices in $L(G)$ can be obtained by employing power iteration on the transition probability matrix derived from $L(G)$ (i.e a random walk on line graph). as similar to PageRank. Since the construction of line graph is computationally expensive [30], the author introduced a decomposed form of $L(G)$.

$L(G)$ is decomposed into two sparse matrices known as Source Incidence $S(G)$ and Target Incidence $T(G)$ matrices. $L(G)$ can be reconstructed by as shown in equation 3.4

$$L(G) = T(G)S(G)^T$$

This actually results in the adjacency matrix of line graph $L(G)$. So in order to do the power iteration method for finding the steady state probabilities of the vertices in the line graph, we need to generate the column stochastic transition probability
matrix of this line graph (which is can be done by simply transposing the adjacency matrix and column normalizing the resultant matrix as mentioned earlier). Since we wanted use the sparse matrices the author achieves the above mentioned in the following way.

As we don’t have the entire L(G) matrix, we can’t do column normalization and generating transition probability matrix before head of the iteration. So instead of making the matrix column stochastic (dividing each element in the column by its column sum) and multiplying it with the vector in the iteration, the author first computes the sum of the columns before the iteration and divides the vector in the iteration with the corresponding values of the column sums. This is illustrated in the appendix A.6.

3.3 Analogy between PageRank and LineRank

In PageRank, we find the importance score of the vertex while in LineRank the score computed is the significant score of edges. There are two important analogy between the workings of LineRank and of PageRank algorithm to be noted as follows:

1. Both algorithms targets to compute principal eigenvector of the matrix describing the transition probability

2. They are implemented based on PowerMehod for computing the eigenvector

So far this chapter has focused on the detailed study of the centrality algorithms in various perspectives along with an introductory graph and matrices terminologies. Having knowledge it is now the time to look into the details of the practical implementations of these algorithms in the selected distributed data processing platforms.
Chapter 4

Implementation

Having defined what is meant by Effective Closeness and LineRank algorithm, this chapter extends to give an in-depth examination of how these measures are essentially implemented in a completely distinct computing platforms such as Apache Flink and Giraph. This topic can be best treated under four headings:

1. Effective Closeness in Flink
2. Effective Closeness in Giraph
3. LineRank in Flink
4. LineRank in Giraph

4.1 Effective Closeness in Flink

Effective Closeness based on Flajolet-Martin sketch (introduced in the section 3.2.3) computes the number of nodes in distances starting from 1 to the diameter of the graph. In the reference paper [30], the algorithm follows an approximation approach for counting the number of neighbor nodes. This technique is popular for counting unique number of items in a group of elements known as Flajolet-Martin sketch. This method uses a memory efficient bit string to keep track of the count on unique items.

Effective Closeness algorithm starts by initializing a bit string to each node and updates it based on the available number of neighbor nodes in every super step.
This update is efficiently achieved by performing a BITWISE-OR operation on the bit strings of neighbor nodes. In every iteration, each node updates its information about its neighbor nodes (the count of the available neighbor nodes) by adding the current node’s bit string with a list of its neighbor nodes’ bit strings.

As illustrated in the figure 4-1. Initially each node in the example graph exchanges its assigned bit string with its neighbor nodes (as shown in the left part of the figure). In the first iteration, each node receives its bit string from its neighbor nodes (boxed in red) and executes a BITWISE-OR operation to update its count for the nodes that are reachable in one step.

![Figure 4-1: The Effective Closeness algorithms as an Incremental Iteration](image)

It is clear to state that, a bit string of a node is only updated if at least one of the neighbor nodes has changed its value in the previous iteration. Therefore a relative update on a bit string of a node directly impacts only the neighbor nodes in the next iteration. This kind of update (computation) of a node that depends only on few other nodes is known as sparse computational dependency [18] and it can be efficiently implemented using delta iteration construct in Apache Flink. The iterative data flow of Effective Closeness algorithm implemented in Apache Flink is shown in the Figure 4-2.

**Goal: To compute sum of the shortest path distances of each node**

Effective Closeness algorithm implemented using Apache Flink requires two kinds of input sources stored in distributed file system (HDFS).

1. **Vertices** has a format of `verId`. 
2. **Edges** are represented as a pair of source and destination vertices \((srcId, desId)\).

The implementation starts by reading the vertices from HDFS and each vertex \((verId)\) is associated with a unique bit string and a sum \((FMCounter, sum)\) where the sum is initialized to 0. A simple Map operator in Flink can be used to accomplish this task which in turn emits \(verId\) as a key and \((FMCounter, sum)\) as value. Each record from the Map operator (Assign Bit Array to Vertex) is initially considered to be the solution set \(S\) as well as the work set \(WS\) of delta iteration as shown in the figure 4-2 and the record is uniquely identified by the key \(verId\). The dashed line represents the iteration operator, encompassing the data flow that computes \(WS'\) and \(\Delta\) from \(WS\) and \(S\) respectively.

**Delta computation:**

The current work set \(WS\) is joined with the data source Edges to derive next iteration’s work set \(WS'\). Another point we should note here is that, the current work set’s bit array has to be distributed only to the nodes from which the current node has incoming edges alternatively the bit arrays are only replicated as many times as equal to the in-degree count of current node.

Edges of the form \((srcId, desId)\) are joined on \(desId\) with the key attribute of current working set \(WS\) \(verId\) and finally the join operator emits the record of the form \((srcId, (FMCounter, sum))\) for each joining key using. This is said to be the partial bit arrays of \(srcId\). Implicitly by this way, each node in the current working
set (WS) is sending its bit array to its immediate neighbor nodes and also the bit array is replicated only to its incoming edges.

Following the join operation, grouping by srcId on the join result will get a list of all the partial bit arrays of desId which are then BITWISE-OR’ed to obtain the updated count of neighbor nodes. This is accomplished by using a GroupReduce operator in Flink.

Finally, we combine the newly updated working set WS with the solution set S through a join operation called Update Function as shown in Figure 4-2. Here we perform main two tasks:

1. First, compute the sum (partial sum of the shortest path which will be updated in each iteration): For this we need the count of the neighbor nodes in the current as well as in previous iteration. The count of neighbor nodes in the current iteration can be obtained from our updated bit array. The count of neighbor nodes in the previous iteration can be retrieved from solution set S. (The solution set is persistent across all the iterations)

2. Second, update the partial solution Δ with the current iteration’s values. At the same time, a termination condition is verified to check whether the current vertex is converged, i.e if the current vertex’s count is same as the in the previous iteration then the vertex is said to be converged. In that case, the corresponding record is filtered out from the working set WS.

At the end of delta iteration, we compute the average of the sum of shortest distances for each node. The sum parameter resulted from the delta iteration represents the sum of the shortest distance. Average computation is achieved by using a simple Map operator ("AverageComputation") as shown in the Figure 4-2 and the total number of vertices are given as an input parameter while submitting the job. Finally the computed closeness results are written back to HDFS.
4.2 Effective Closeness in Giraph

The counterpart of Effective Closeness in Giraph is referred from an existing academic project work from a group of student at TUBerlin and the corresponding code can be found here. In short, Giraph based implementation follows a vertex centric model, where the step function constitutes the computation executed by each vertex in the graph. In our case it is the logic of counting the neighbors using Flajolet-Martin sketch incrementally through iteration.

Moreover, a second data structure is maintained to keep track of the number of nodes in every shortest path distances. Thereof, at end of iteration the count of neighbor nodes in the shortest path distances from the data structure is added to generate the summation” part of the algorithm. On the other hand, Giraph works in batch processing mode where the output of current iteration is entirely forwarded as an input of next iteration in the sequence whereas in Flink, the delta iteration works by filtering the converged nodes from the iteration, consequently reduces the data set to be processed in the following iterations.

4.3 LineRank in Flink

LineRank algorithm is implemented by utilizing the bulk iteration operator in Apache Flink. The description is based weighted graphs in fact for unweighted graphs by default the weights are considered to be 1. Figure 4-3 demonstrate a complete picture of LineRank algorithm implemented in Flink. This can be explained in detail by dividing into following sections.

1. Building Incidence matrices

2. Computing normalization

3. Power Iteration

4. Aggregation of incident edges
4.3.1 Building Incidence Matrix

To build two sparse matrices known as **Source Incidence Matrix** $S(G)$ and **Target Incidence Matrix** $T(G)$ from a source file that contains edges in the form $srcId \rightarrow desId$.

For each input line from the edge file, an edge id is generated sequentially and
it is paired with srcId to generate S(G) of the form \((\text{edgeId, srcId, weight})\). Similar procedure is followed to generate T(G) of form \((\text{edgeId, desId, weight})\). Each record of this form belongs to a sparse representation of source and target incidence matrices. These two incidence matrices are precomputed before the power iteration. So it constitutes the constant path of bulk iteration and thus it is depicted in dotted lines denoted as by two circles S and T as shown in the figure 4-3.

### 4.3.2 Computing Normalized Vector

Column-normalization is performed by dividing each column element by the sum of its column values. The sum of the columns are precomputed (before the power iteration and the concept is illustrated in Appendix A.6) and its inverse is multiplied with the vector in the iteration that turns out to be the dominant vector of the power iteration method. The column aggregation is computed efficiently by utilizing two sparse matrices in the following way:

Compute \(d_1 \leftarrow S(G)^T \mathbf{1}\)

A matrix-vector multiplication is performed between \(S(G)^T\) and a unit vector of size equal to the total number of edges to compute aggregation of rows in \(S(G)^T\).

In Apache Flink it is implemented by first invoking a groupBy operation on srcId in the records of S(G) \((\text{edgeId, srcId, weight})\) (which is equivalent to transposing S(G)). Then in the next step, an aggregation operation on the grouped row values is performed to get the resulting records of the form \((\text{srcId, weighted-sum})\). The REDUCE operator just above ‘S’ in the Figure 4-3 illustrates this functionality.

Compute \(d_2 \leftarrow T(G)d_1\)

As a second step in normalization, T(G) is multiplied with \(d_1\) computed in the previous step, in order to generate the sum of column elements as mentioned in the beginning of this section.

In order to do sparse matrix-vector multiplication, the elements in matrix T(G) and the matching element in the vector \(d_1\) are joined on the same index id. For example, in our case we have the sparse matrix with the records of the form \((\text{edgeId,} \)
targetIncId, weight) and the vector with the records of the form (sourceIncId, weighted-sum). A join on targetIncId and sourceIncId is performed.

\[
\begin{pmatrix}
0 & 1 & 0 & 0 & 0 \\
2 & 3 & 7 & 3 & 3 \\
0 & 3 & 0 & 0 & 0
\end{pmatrix}
\]

Target Incidence Matrix has only one non-zero value for each row irrespective of the number of nodes and edges. It implies that for each row in T(G) exactly one corresponding element in the vector is multiplied and the others becomes zero which eliminates the need of aggregation followed by multiplication in the matrix-vector multiplication. This operation is displayed as a JOIN that falls under T in the figure 4-3.

Compute \( d \leftarrow 1/d2 \)

As a final step in normalization, an element-wise inverse operation is performed on d2 resulted in the previous step using a Map operator in Flink. This is a factor that normalizes the vector in the iteration that estimates the principal eigenvector. In this way, LineRank algorithm efficiently moves the static computation part of normalization factors out of the iteration and reduces the workload inside iteration.

4.3.3 Power Iteration

Bulk iteration operator is utilized to find the principle eigenvector of the matrix induced from L(G) which is proclaimed as edge scores. As mentioned in [30], instead of materializing entire L(G), the algorithm performs matrix vector multiplication in a decomposed form. It first multiplies the vector(edge score) in the iteration with \( S(G)^T \) and then the result is multiplied with T(G). Since both of these matrices are sparse the time (\( O(km) \), where \( k \) is the number of iterations and \( m \) is the number of edges) and the space (\( O(m) \)) complexity is relatively very low compared to the
computation in original graph [30].

Pre-Iteration processes:

An initial random vector of size equal to total number of edges is generated by using a Map operator in Flink. The initialized rank vector I of the form (eId, r) is shown in the bottom of the Figure 4-3, where eId represents the edgeId and r indicates a randomly generated value.

Bulk Iteration in Flink:

A bulk iteration operator is initialized with maximum number of iterations as shown in the code snippet.

The stopping criterion of the iteration can be specified during initialization of iteration. In simple case, a termination condition can be designed to check if the difference of current and previous iteration’s value is less than a given threshold. But mathematically, the power method finds L1 norm of the vectors in the current and previous iteration. In our implementation we follow the later scenario as a termination condition of the iteration. The final convergence check is achieved by JOIN operator which gets the solution set and the current work sets as inputs and hence checks the condition as shown in Figure 4-3 and C in the data flow stands for closing signal of iteration.

```java
IterativeDataSet<Tuple2<Long, Double>> iteration = edgeScores.
    iterate(maxIterations)
    .registerAggregationConvergenceCriterion(
        L1_NormDiff.AGGREGATOR_NAME,
        DoubleSumAggregator.class,
        L1_NormConvergence.class)
    .name("EdgeScoreVector_BulkIteration");
```

The step function in the iteration comprises of the following operation in sequence, 1. Hadamard Product

Hadamard product of two vectors is a direct multiplication of components of the vectors which results in a new vector with same dimensions. The following example illustrates the Hadamard product of two vectors named d and v with dimensions (5 × 1).
\[
\begin{pmatrix}
2 \\
3 \\
3 \\
5 \\
2
\end{pmatrix}
\begin{pmatrix}
3 \\
6 \\
1 \\
3 \\
2
\end{pmatrix}
= 
\begin{pmatrix}
6 \\
18 \\
3 \\
15 \\
4
\end{pmatrix}
\]

As illustrated above, the elements of these two vectors are multiplied at the same index to produce the result. To do this a JOIN operator in Flink is used to get the elements that resides in the same index of the two vectors. The intuition of this operation is to produce normalized vector which will then be used for matrix vector multiplication in successive iterations. In the Figure 4-3, the first red box from the bottom represents this functionality.

2. Matrix vector multiplication
The most critical operation of power iteration is the matrix-vector multiplication of transition probability matrix of L(G) and a random vector that turns out to be eigenvector after certain number of iterations. Since we only have \(S(G)\) and \(T(F)\), this operation is executed in two steps as follows,

- \(v_2 \leftarrow dS(G)^T\)
  
  A JOIN operation between \(S(G)\) (\(edgeId, srcId, weight\)) and \(v_1\) (\(edgeId, value\)) on edgeId attribute will be equivalent to multiplication part of the matrix-vector multiplication and then emits (\(srcId, partial-mul\)). The aggregation operation followed by JOIN operation is equivalent to summation in matrix-vector multiplication. Thus partial multiples from JOIN is grouped on (\(srcId\)) using a REDUCE operation. This is illustrated in second and third red boxes in the flow diagram 4-3 from the bottom.

- \(v_3 \leftarrow T(G)v_1\)
  
  Here, the results of \(v_2\) has to be joined with target incidence matrix \(T(G)\) and it is termed as \(v_3\). As elaborated before, the aggregation followed by multiplication is not required as \(T(G)\).
4.3.4 Aggregation of incident edges

As a final step in the computation of LineRank of the vertices, the incident edges scores which are computed using the power iteration are aggregated to come up with the final betweenness value. The implementation follows the right side of the equation 

\[(S(G) + T(G))^T \times V = (S(G)^T \times V) + (T(G)^T \times V)\].

The multiplication and addition operation in the right side equations are replaced by a Join operator in Flink. 

\((S(G)^T \times V)\) is obtained by joining on edgeId in \((\text{edgeId,srcId,weight})\) and edgeId in \((\text{edgeId,eigenvec-value})\). The same procedure is followed to compute \((T(G)^T \times V)\).

At last, the sum is computed by joining the results of two partial joins.

4.4 LineRank in Apache Giraph

As an initial rule of thumb, input format of compute method in Giraph encompasses each vertex, the neighbor nodes and other required information depending on the logic of the considered algorithm.

The input format of unweighted LineRank algorithm is designed in such a way that each line contains a source vertex with incoming and outgoing edges also identified as incident and adjacent edges respectively. The source code for this initial data processing task can be found at GiraphInputFormatForLineRank

LineRank algorithm is grounded on power iteration method for computing score of edges. In other words it finds ranks of edges in a graph and aggregates these ranks on vertices it connects to and from. This is accomplished by generating two sparse matrices and multiplying it with a random initial vector in a step-wise order. This process is repeated until the random vector is converged and the resulting converged vector is known to hold the stationary probability of edges. By generating sparse matrices, partially it means transforming input of power iteration method from vertices to edges. Since the incoming and outing edges are decomposed into two matrices, it inherently becomes sparse when compared to line graph.

In this thesis, the idea in Giraph implementation of LineRank algorithm is to exploit the concept of vertex centric model without generating line graph and verify
the ease of programmability to accomplish this computation in Giraph. There were
two clear things before starting to implement and they are as follows,

- As one of the goals of LineRank algorithm is to compute the score of edges, it is
  very obvious that it is not possible to store it as state of the vertex in Giraph’s
  compute method. This is analogous to ask the vertex to store state of the edges
  in the vertex centric model.

- Even though the score of edges are computed in the final step, the algorithm
  requires to aggregate edge’s score for each vertex. After a deep analysis of the
  algorithm in order to map with vertex centric model, the following solution is
  implemented first to carry out the two main computations in the power iteration.

\[
\begin{align*}
v_2 & \leftarrow dS(G)^T; \\
v_3 & \leftarrow T(G)v_2; \\
v_1 & \leftarrow cv_3 + (1-c)r;
\end{align*}
\]

For the sake of understanding, normalization that has to be done before these
computations are discussed later in the explanation progressively. For now we focus
on the two steps of computations (v2 and v3) and how to fit this computation into
vertex centric model. For the sake of simplicity we decided consider only un-weighted
graph.

Since Giraph follows BSP model, iteration in Giraph is called as super step and
the compute method represents a step function that will be executed by each worker
in the cluster in parallel and the barrier of synchronization is handled by the internals
of Giraph.

We here propose a new approach for computing three values as listed above (v1,
v2 and v3). The main objective here is to program the above two computations as
step functions in Giraph. This is modeled and implemented as follows:

1. Current state of the vertex is assigned with computation result of v2
2. The messages that are distributed or exchanged in the iteration are considered to the edge score (v3). Before exchanging edge score value, it is updated with damping factor calculation which is then re-assigned as v1 in the end of compute function.

3. Of course during initial iteration, v1 is initialized with random values and will be updated iteratively until the values reach a steady state.

**Pseudo-code:**

**Step 0:** Each vertex’s state is initialized with value equivalent to fraction of total number of edges \((1/\text{numEdges})\) in the graph. (The number of edges can be passed as a runtime argument during job submission). The current vertex value is send to its incident vertices by equally dividing it by the number of adjacent vertices.

if Step > 0: Current state of the vertex is updated by aggregating incoming values that are received via messages argument in compute method (v2). This value \(v2\) is updated with damping factor computation (v3) Finally before re-computing \(v1\), \(v3\) is divided by the number of adjacent vertices of the current vertex as same as in step 0 and distributed to its incident edges.

**Termination Condition:** As explained in LineRank in Apache Flink section, the termination criterion is finding the L1 Norm of the edge score vector in the iteration at the end of each super step. This is accomplished in Giraph using Master Compute abstraction in Giraph.

Master compute helps to initialize a global aggregator which will be accessed by all the worker nodes and can be updated as well. The compute method of Master Compute is called once for each super step before the worker compute is called in the parallel execution. So in this implementation this features best suits to check our termination criterion and for aggregating L1 norm difference of the values in all the workers. While re-computing the value of edges (v1) that are distributed as send
messages in the iteration it is subtracted from the previous iteration’s edge score (v1) which is computed from the current vertex value. The difference is considered to be the partial value of L1-Norm and it is aggregated using "DoubleSumAggregator" of Giraph. It is important to note that the current vertex’s state is updated after afore-mentioned step.

At the same time, the implementation includes a condition in the compute method of the Master Compute abstraction to check whether the computed L1 Norm in the aggregation is less than certain threshold if so it stops the entire iteration. Illustration of pseudo-code: Consider the sample graph shown in the figure 3-1 for the illustration. The graph has three vertices and 5 edges. Source incidence matrix S(G) of the graph 3-1 entails set of all out-going edges in that graph. Rows in the below matrix represent the adjacent edges and the columns represent the vertices.

\[
S(G) = \begin{bmatrix}
a & 1 & 0 & 0 \\
b & 0 & 1 & 0 \\
c & 0 & 0 & 1 \\
d & 1 & 0 & 0 \\
e & 0 & 1 & 0 \\
\end{bmatrix}
\]

The algorithm computes v2 by doing matrix vector multiplication, \( v2 = S(G)^T v1 \). Here v1 is an initial vector with values initialized with fraction of the total number of edges available in the graph. Also, v1 represents edge score vector. In other words this is the vector which will be converged and identified as principal dominant eigenvector in power iteration method. The first iteration of power method for v2 computation is illustrated below, where the initial vector and source incidence matrix are multiplied.
to get v2.

\[
v2 = \begin{pmatrix}
1 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0
\end{pmatrix}
\begin{pmatrix}
0.5 \\
0.5 \\
0.5
\end{pmatrix}
\]

Figure 4-4:

If we try to understand the intuition behind this matrix-vector multiplication, it reveals the fact that each vertex updates its current value based on the values of the other vertices that it connects to (out-going edges). For example, consider figure 1-1 and matrix-vector multiplication. Vertex 1 has two out-going edges a and d and it re-computes its value in the first iteration based on the values that it receives via ‘a’ and ‘d’ (which are highlighted here). But it is being said that a vertex computes it value based on the values received through its out-going edges.

Logically speaking a vertex can’t receive its value via an out-going edge and it can only receive messages via its incoming edges. This made things harder and raised a question, How to achieve this computation in Giraph? (change this para) One thing is very clear that we can’t achieve this by considering that each vertex sends its updated value via its outgoing edges (as similar to page rank algorithm), because for vertex 2, expected to receive updated value from 1 and 3 which will not possible.

If we take a closer look into Figure 1-1, it is achievable if each vertex sends its
current value to its in-coming edges to compute v2. For example, if vertex 1 sends its value to 2 and 3, it will be received by the corresponding vertices in the next iteration. In the same way vertex 2 sends its value to 1. In the end, vertex 2 receives from 1 and 3, vertex 1 receives from 2 and 3 and vertex 3 from 1.

As we know that vertices equally distribute its value to its neighbors, it is required to divide current vertex’s value by number of edges it distributes to. Since we achieve the concept receiving neighbor vertices value via out-going edges we divide current vertex’s value by total number of adjacent vertices though we distribute it to incident edges. (Also, this process is equivalent to the normalization part of the algorithm). The corresponding implementation classes can be found in Appendix.

1. UnweightedLineRank
2. Directions
3. IncidentVerticesIterator
4. LineRankMaster
5. UnweightedLineRankVertexWorkerContext

The distribution of the normalized value is send to its incident edges via the following line

```java
sendMessageToMultipleEdges(new IncidentVerticesIterator(vertex.getEdges().iterator()), sendingVal);
```

Here, ”IncidentVerticesIterator” retrieves the list of incoming edges of the iterator and send using ”sendMessageToMultipleEdges” method of Giraph. This operation is equivalent to v3 computation as shown below

\[
v3 = \begin{pmatrix}
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 0 & 0 \\
0 & 0 & 1 \\
0 & 0 & 1 \\
\end{pmatrix}
\begin{pmatrix}
1 \\
1 \\
1 \\
0.5 \\
\end{pmatrix}
\]

49
The final step of aggregation in the algorithm is $S(G) + T(G))T v$. The intuition behind this computation is that aggregating the computed edge scores of each vertex. More precisely, it aggregates the edge score of both incident and adjacent edges. If we again think about it, we already achieve a partial aggregation of the final step when we do $v2$ computation (because it basically represents the aggregation of incoming edges score). So in one step after the convergence, we need to send the current vertex value to both incident and adjacent edges to accomplish the final step of the algorithm. This is achievable only when we know the total number of iterations before in hand. This has been verified locally and it is not considered for experiments because of the aforementioned reason.

This chapter started by describing the implementation of two algorithms Effective Closeness and LineRank in two different platform. Also we proposed a new approach for implementing LineRank algorithm in Apache Giraph. We conclude this chapter by saying that, programming complex algorithms in Giraph is way more difficult compared to Flink. The next chapter describes the experiments and the evaluation conducted for the implementations explained in this chapter.
Chapter 5

Evaluation

This chapter encompasses one of the main objectives of this thesis, namely a thoroughly designed approach to analyze effectiveness, performance and scalability of centrality measures (closeness and betweenness) implemented as discussed in previous chapter. The focus of experiments conducted has three main goals. First, the accuracy of obtained results are validated against the standard measures of centralities. Next, we wanted to compare the effects of runtime performance and scalability of these two algorithms on different data sets by varying the execution parameters. Finally, a descriptive analysis on these centrality measures is studied and demystified. After briefing about the runtime environment and dataset used in the experiments, the rest of this chapter focuses on each of these goals separately.

5.1 Experimental Setup and Datasets

The experiments are executed on IBM Power cluster maintained at TU Berlin premises. The cluster hosts 10 nodes of machines each equipped with POWER7 architectured CPUs (48 cores) and 48 GB RAM. 1 node acts as master node and the rest 9 are set as slave nodes. Consequently the available number of parallelism at the cluster is 432. Both Giraph and Flink use HDFS as its storage layer and for the experiments we used Hadoop version 1.2.1. In the beginning of this thesis, Flink was previously named as Stratosphere and it is now in Apache incubator. Hence the Flink based
implementation part of this thesis follows the stable release of Stratosphere-0.5.1 and the Giraph based user code follows Giraph version 1.1.0-SNAPSHOT release. The data sets used for the experiments are shown in table 5.1.

The first group of graph datasets (DBLP-PA, Enron, Protein, arXiv) in the table are smaller in volume (number of edges) relative the other datasets. Scalability experiments are conducted on the second group of datasets [33] which ranges from 5 to 93 millions in their volume and they represent the sources from diverse domains as mentioned in the description column of the table 5.1. The graphs are represented as adjacency list containing single edge information for each line (for e.g., source_vertex, target_vertex) and the same format is applicable for all the datasets.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP-PA</td>
<td>3 K</td>
<td>23 K</td>
<td>DBLP Prolific Authors</td>
</tr>
<tr>
<td>Protein</td>
<td>20 K</td>
<td>82 K</td>
<td>Protein Interaction Network</td>
</tr>
<tr>
<td>Enron</td>
<td>37 K</td>
<td>368 K</td>
<td>Email Network</td>
</tr>
<tr>
<td>arXiv</td>
<td>27 K</td>
<td>705 K</td>
<td>arXiv Citation Network</td>
</tr>
<tr>
<td>Google</td>
<td>88 K</td>
<td>5 M</td>
<td>Web Pages Network</td>
</tr>
<tr>
<td>Youtube</td>
<td>3 M</td>
<td>12 M</td>
<td>Social Network</td>
</tr>
<tr>
<td>EU-Web</td>
<td>8 M</td>
<td>38 M</td>
<td>EU domain Web Pages Network</td>
</tr>
<tr>
<td>DBLP</td>
<td>1 M</td>
<td>19 M</td>
<td>Citation Network</td>
</tr>
<tr>
<td>Live Journal</td>
<td>5 M</td>
<td>68 M</td>
<td>Social Network</td>
</tr>
<tr>
<td>Facebook</td>
<td>60 M</td>
<td>93 M</td>
<td>Social Network</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of Datasets and its main properties

5.2 Preliminaries of accuracy estimation

The accuracy of our experimental results are assessed against standard centrality measures by applying the novel correlation techniques. Since the standard measures that we have used for benchmarking, doesn’t scale for very large datasets (explained here 5.2) we decided to avail graphs with less number of edges for accuracy evaluation. We executed our experiments on the datasets placed in the upper box of the table 5.1. The strategy for computing standard closeness and betweenness measures are given below:
Standard Closeness (BitField Closeness)

For benchmarking Effective Closeness, we adopt an existing work for computing the actual closeness measure. This work basically follows the same concept of Effective Closeness except that, instead of approximating the count of neighbors in each iteration (as in Flajolet Martin sketch), this method maintains the actual count of neighbor nodes by using integer arrays. By this way we could able to compute the actual score of closeness of a node in a network. But this method is memory restrictive since we try to maintain the actual count using array this method doesn’t scale up for graph with millions of edges. The results obtained by running BitField closeness in Giraph are considered as the base for closeness accuracy estimation.

Standard Betweenness

The original LineRank algorithm is implemented in MatLab and the computations are executed sequentially in local machine. The MatLab script can be found in the Appendix A.1.

Three kinds of correlation methods are used for evaluating accuracy of the results computed by both the systems with the standard results.

5.3 Accuracy

We computed two rank correlations (Kendall’s tau and Spearman’s Rho) and one linear correlation (Pearson). Centrality measures are typically associated with a rank order. Kendall’s tau is used to find the degree of concordance between the results from our experiments and the standard measures [31]. Kendall’s tau will be low if the ranking order between the compared entities has greater number of inversions and also it is insensitive to detect any unusual discrepancies present in the rank order. Alternatively, we compute Spearman’s Rho co-efficient to find such anomalies as a second method to verify the accuracy. Moreover, Spearman’s co-efficient identifies the monotonic relationship that exists between the compared entities and
rather disregards linearity. Thus Spearman’s co-efficient plays a predominant role in rank correlation. Finally, Pearson correlation is calculated to compare the linear association between the actual values of the results.

### 5.3.1 Effective Closeness

<table>
<thead>
<tr>
<th></th>
<th>Kendall’s tau</th>
<th>Spearman’s Rho</th>
<th>Pearson Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP-PA</td>
<td>0.902</td>
<td>0.987</td>
<td>0.99</td>
</tr>
<tr>
<td>Enron</td>
<td>0.903</td>
<td>0.974</td>
<td>0.952</td>
</tr>
<tr>
<td>Protein</td>
<td>0.822</td>
<td>0.941</td>
<td>0.969</td>
</tr>
<tr>
<td>arXiv</td>
<td>0.932</td>
<td>0.994</td>
<td>0.996</td>
</tr>
</tbody>
</table>

Table 5.2: Effective Closeness in Flink vs BitField Closeness

Table 5.2 shows the three correlation coefficients between standard measure and the experimental results of Effective closeness in Flink, evaluated for four different dataset. It is observed that all three correlation coefficients do have positive and strong relationship and hence it is evident that the result are highly accurate for almost all experimented data sets. In the following table 5.3, the Giraph version of Effective Closeness results noticed to have highly correlated with Standard measure for Enron dataset relative to Flink counterpart. A remarkable point to note here is that, Effective Closeness algorithm is an attempt to approximate closeness computation and hence it is reasonable to have near linear correlation. A scatter plot between

<table>
<thead>
<tr>
<th></th>
<th>Kendall’s tau</th>
<th>Spearman’s Rho</th>
<th>Pearson Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP-PA</td>
<td>0.902</td>
<td>0.987</td>
<td>0.99</td>
</tr>
<tr>
<td>Enron</td>
<td>0.916</td>
<td>0.987</td>
<td>0.933</td>
</tr>
<tr>
<td>Protein</td>
<td>0.822</td>
<td>0.941</td>
<td>0.969</td>
</tr>
<tr>
<td>arXiv</td>
<td>0.932</td>
<td>0.994</td>
<td>0.996</td>
</tr>
</tbody>
</table>

Table 5.3: Effective Closeness in Giraph vs BitField Closeness

the standard closeness and Effective closeness results for DBLP-PA dataset is illustrated in Figure 5-1. Each point in chart represents a author-id in DBLP Prolific Author network.
5.3.2 LineRank

Since the complete LineRank implementation is restrictive in Giraph as discussed in previous chapter we computed, in Flink the implementation is modified to compute only the edge score vector, in order to have balanced comparison for both frameworks. It is recognized from the correlation analysis that the result of both Giraph and Flink for LineRank(v2) computation is perfectly linear as shown in the table 5.4 for DBLP-PA dataset and the corresponding graphical representation is shown in figure 5-2. Moreover, Enron data again shows strong rank and linear correlations with just few nodes difference with standard measure. We also tested to see how

<table>
<thead>
<tr>
<th></th>
<th>Kendall’s tau</th>
<th>Spearman’s Rho</th>
<th>Pearson Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP-PA</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Enron</td>
<td>0.998</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.4: Correlation coefficients of LineRank(v2) in both Giraph and Flink is realized to have same values and represented above against Standard LineRank(v2) from MatLab

similar the results of LineRank till v2 computation with the outcome of full LineRank implementation. It is evident from table 5.5 where the Kendall’s co-efficient of DBLP-PA says 99.6 percentage of match in the rank order between the results from LineRank(v2) and full LineRank whereas Spearman’s rho and Pearson correlation shows exact linear relationships. In case of Enron dataset it is again observed to have strong and linear correlations. From these experimental analysis, we propound that LineRank(v2) computation is a good approximation on full LineRank computation.
With a comprehensive study on accuracy we proceeded to further experiments by calculating correlation coefficients of LineRank(v2) vs Full LineRank. The results are as follows:

<table>
<thead>
<tr>
<th></th>
<th>Kendall’s tau</th>
<th>Spearman’s Rho</th>
<th>Pearson Correlation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBLP-PA</td>
<td>0.996</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Enron</td>
<td>0.741</td>
<td>0.91</td>
<td>0.881</td>
</tr>
</tbody>
</table>

Table 5.5: Correlation coefficients of LineRank(v2) vs Full LineRank

confirming that the same results will hold true for larger dataset with more accurate results.

### 5.4 Scalability

This chapter is divided into two parts each of which presents the results relating to the two centrality algorithms. In each part of this section we will report the individual experiments conducted on both platforms (Apache Giraph and Apache Flink). As pointed out earlier in this section, the datasets used for these experiments are real graph networks that covers social, citation and web networks.

#### 5.4.1 Effective Closeness

Figures 5-3 and 5-5 show the run time of Effective Closeness executed in Apache Giraph and Apache Flink respectively. For each dataset set considered for this evaluation (Google, Youtube, EU-WEB and Live Journal), we have executed the implementations of Effective Closeness separately in both the platforms.
Scalability - Fixed number of parallel tasks

The line chart in the figure 5-3 shows the run time of Effective Closeness in Apache Giraph for varying datasets that increases in volume sequentially and the number of parallel cores are fixed to 15. The run time apparently grows near linearly with respect to the increasing number of edges in the graph. Interestingly, the run time on EU-Web dataset drops to a small degree relative to its predecessor (from 135 seconds to 123 seconds) in contrast to the expected linear increase. One possible reason for this behavior can be observed from the bar chart located just below the line chart.

![Figure 5-3: Effective Closeness in Giraph: Runtime vs Edges](image)

The bar chart represents the same executions but the chart is plotted against the total number of iterations instead of the run time. It can be seen from this chart, the total number of iterations for EU-Web dataset is relatively less (15 iterations) compared to its predecessor dataset (Youtube).

As an attempt to find out the reason for less number of iterations in EU-Web, we
tried to compare the time taken to complete each superstep for both EU-Web and Youtube datasets. In general, according to the framework model, the run time should decrease with the increasing supersteps. This is because, the vertices in Giraph are programmed to the stop execution (vote to halt) once it has finished its assigned task. It will be re-activated only if the vertex receives messages in the future iterations as explained in chapter 2.

![Figure 5-4: Effective Closeness in Giraph: Comparing the running time of each iteration for EU-Web and Youtube datasets](image)

In figure 5-4, as expected for both datasets the time of computation has a decreasing trend. But for the EU-Web graph at superstep 6, there is sudden fall in the computation time and later completes the execution with very less iterations. This reveals an interesting observation that the EU-Web data set is highly sparse and the nodes are loosely connected. Also, this evidence proves that the Effective Closeness algorithm in Giraph exploits the sparse computational dependency nature of the graphs in which during every iteration it only computes for the dependent nodes instead all the nodes in the graphs which is not possible with Apache Hadoop.

Turning now to the experiments of Effective Closeness algorithm In Apache Flink with the same datasets and number of cores (15). In figure 5-5, the runtime of Effective Closeness grows near linearly while increasing the dataset size. In addition to that, the number of iterations completed for each dataset are almost equal to the number of iterations we that have seen for Apache Giraph, But the run time of
Apache Flink is relatively greater than Apache Giraph for the same datasets.

![Figure 5-5: Effective Closeness in Flink: Runtime vs Edges](image)

**Scalability - Fixed graph data**

Another set of experiments are conducted to test the scalability by keeping the graph volume to be constant while modifying the number of parallel cores. When the available number of cores increase, then the expected runtime should regress linearly. The graph used for these experiments is a social network graph (Live Journal) with 68 millions of edges.

The graph 5-6 compares the rate of run time in each of the platforms. From the information shown in the graph, it is obvious that the run time of Giraph for these experiments are less compared to Flink. Two important patterns observed from the graph 5-6 are:

- For Giraph (in blue line), initially the run time declines linearly with 35% of drop when the number of cores increases from 12 to 16. Later the run time
remains nearly stable irrespective of the increase in number of parallelism. One possible reason for this behavior where there is no increase in performance by adding cores might be there is not enough work to do by the available number of workers. Since the latest version of Giraph has better optimizations with multi-threading\textsuperscript{1}, it might be a reason that the master may assign multiple partitions to a worker machine and tries to increase the load balancing among the partitions instead of utilizing the available cores.

- On the other hand Flink (in red line), shows gradual decrease in run time as the number of cores increases from 15 to 40. In the later case as similar to Giraph, the run time started to increase with increasing number of machines from 50. This trend in Apache Flink might be due to the additional work done by the system for increasing number of parallel tasks (for example network communication overhead ). We say that the behavior follows the Gustafson’s Law\textsuperscript{2} in which parallel overhead in the system increases with the increasing number of parallel processor.

\textsuperscript{1}https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920

\textsuperscript{2}http://en.wikipedia.org/wiki/Gustafson%27s_law
5.4.2 LineRank

To experiment the scalability of LineRank algorithm we used the implementation that computes the edge score vector (v2) of the algorithm in both platforms. As mentioned earlier in implementation chapter, the complete version LineRank computation is less feasible to implement in Apache Giraph. Moreover, as stated in the section that discusses the accuracy 5.3.2, the results of LineRank(v2) computation and full LineRank computation are proved to have almost similar results and hence going forward we refer LineRank(v2) computation as LineRank.

Scalability -Fixed number of parallel tasks

Figure 5-7, show the running time in both platforms for LineRank algorithm. The number of cores maintained constant for these experiments are 15. Again here, Giraph outperforms Flink by running in few minutes for all the data sets that are evaluated and also shows linear a increase with the increasing number of edges in the dataset. Flink takes comparatively longer time but the curve shows a linear increase. The square boxes near the points on the lines indicated the number of iterations required to complete the algorithm for each dataset.

![Figure 5-7: (a)LineRank in Giraph: Runtime vs Edges (b)LineRank in Flink: Runtime vs Edges](image-url)
Scalability - Fixed graph data

As similar to Effective Closeness experiments, we tested the performance of LineRank by maintaining the graph size as constant. Here we used two different datasets such as Live Journal (with 68 M of edges) and Facebook (93 M of edges) for evaluating Apache Giraph and Apache Flink respectively. There is no particular reason for choosing two different dataset since the main purpose of this experiment is just to measure the scalability of these two platforms. The two line graphs shown in the figure 5-8 are subjected to have similar results compared to Effective Closeness algorithm.

Figure 5-8: (a)LineRank in Giraph: Runtime vs Number of Machines (b)LineRank in Flink: Runtime vs Number of Machines

Comparison of Performance in Apache Giraph and Apache Flink

This section demonstrates the performance of Apache Giraph and Apache Flink with a bar chart as shown in 5-9. Figure 5-9 -[a] represents Effective Closeness and Figure 5-9 -[b] represents the LineRank algorithm. The observation from the bar chart are summarized as follows:

1. It can be observed that the Effective Closeness implemented using delta iteration operator in Apache Flink performs as equivalent to Apache Giraph for some datasets (Google, Youtube) as shown in the figure 5-9 -[a]. This reveals the fact that the advantage of using incremental iteration in Apache Flink in which the number of vertices that are processed in each iteration is reduced as similar to
Apache Giraph. For Google dataset, Flink (green bar) run relatively equal to Giraph and for Youtube dataset, Flink even has less run time with $1 \times$ faster than Giraph. On an average, Giraph performs $1.7 \times$ faster than Flink.

2. For LineRank algorithm 5-9 [b], the bulk iteration in Apache Flink is outperformed by Giraph’s implementation. The reason for this huge difference is due to the difference in implementation details of this algorithm in each of these platforms. As pointed out in chapter 4, the LineRank algorithm in Apache Flink undergoes multiple join operations in the power iteration method but Apache Giraph removes the pain of multiple join operations as the computing platform provides a different possibility of doing the same computation. In this way the load of computation is simplified in Apache Giraph and hence results in improved performance in case of LineRank algorithm. In the end we can infer from the chart that, on an average Giraph runs $3.5 \times$ faster than Flink.

5.4.3 Summary

As a summary of these results we can conclude that for both the algorithms, Apache Giraph apparently proved to perform better than Apache Flink in terms of run time. As an attempt to find reasons for this performance results we think the following evidences are the possible reasons for this results:

- Apache Giraph incorporates much more efficient optimizations in case of message aggregations. As described in a recent article Large-scale graph partition-
ing with Apache Giraph\textsuperscript{3}, the hash based aggregations for exchanging messages between vertices instead of sorting efficiently reduces time spend on message exchange and more over provides a balanced overload on network traffic. But in case of Apache Flink, the platform uses sorting technique while aggregating the data in ‘sort and shuffle’ phase of MapReduce paradigm which result in comparatively poor performance in Apache Flink.

- Apache Giraph employs an efficient strategy while computing the global aggregations\textsuperscript{4} (for e.g., in our case the operation that computes L1 Norm of the vector in LineRank) in which it distributes the global aggregation across the workers and then the workers communicates with each other to access the updated global values.

- Currently, the optimizer in Apache Flink works by first estimating the amount of memory required for all the data flow operators in the plan. Then, based some heuristics, it tries to allocate approximately equal amount of memory for each of the operators in the data flow without recognizing the compute intensive operators which usually requires large amount of memory for computations compared to other tasks \textsuperscript{5}. This forces the system to do external memory computations (e.g., external memory sorting) and consequently slows down the execution. On the other in Apache Giraph, the platform provides an efficient mechanism for estimating the memory requirements depending on the data partition (for e.g., memory required = number of bytes per edge * total edges * 1.5X) and decides to perform out-of-core operations if necessary. In addition to that, Apache Giraph also supports efficient methods for partitioning data across the distributed storage\textsuperscript{6}.

\textsuperscript{3}\url{https://code.facebook.com/posts/274771932683700/large-scale-graph-partitioning-with-apache-giraph/}
\textsuperscript{4}\url{https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920}
\textsuperscript{5}verified directly discussing with Apache Flink team
\textsuperscript{6}\url{https://www.facebook.com/notes/facebook-engineering/scaling-apache-giraph-to-a-trillion-edges/10151617006153920}
5.5 Analysis

In this section, we tried to get the rank distributions of closeness and betweenness centralities. Figure 5-10 shows the line rank distribution of 27 thousand nodes in arXiv citation network. The betweenness measure of arXiv citation network observed to follow a long tail distribution.

![Figure 5-10: LineRank Distribution for arXiv Citation network](image)

A brief comparative analysis of PageRank and LineRank is shown in the figure 5-11. The figure portrays top 10 articles resulted from PageRank and LineRank executions on arXiv dataset with 705 thousand edges. It is obvious to see that, ranks associated to each of the articles by PageRank and LineRank algorithms are different for most of the articles. Article 9 has been identified to have higher PageRank score.

![Figure 5-11: LineRank vs PageRank for Top 10 Articles from arXiv Citation Network](image)
but with lower LineRank score. On the other hand Articles 3 is has higher betweenness centrality than being most popular in the network. It is clearer from the figure that both results have less rank correlations.

Overall, in this chapter we tried to answer our research question relating to the performance comparison for each of these platforms for both the algorithms and we observed that Apache Giraph performs better in term of scalability compared to Apache Flink. We also discussed the possible reasons for these evaluation results. In the following chapter we will only cover the related works in direction of two centrality algorithms as we have already discussed the literature overview related to the computing platform in detail in chapter 2.
Chapter 6

Related Works

Several attempts have been made to postulate an optimized multi-thread algorithms for evaluating betweenness centrality [1,4,5] on a highly powerful and expensive super computers (e.g., Cray XMT). This recent evidence suggests an increasing need towards assessing betweenness centrality in real time applications. Also, a significant analysis of many of these works reveals an emerging trend towards parallelizing the shortest path betweenness that is considered to the standard way of computing the key betweenness index of a vertex. Particularly the work from [1,7], emphasis on harnessing the sparse nature of real world graphs in that they reuse the time and space effective Brandes[8] betweenness algorithm and implemented it in multi-core parallel architecture. Nick Edmonds [7] claims that the algorithms based on fine grained parallelism are relatively complex to distribute the computation over distributed systems. Furthermore he attempts to design an algorithm that exploits the distributed memory infrastructure for finding betweenness measure. Since the real world graphs are scale-free (power law degree distribution), the distributed memory based techniques suffers from load balancing issues [42]. Nevertheless none of these approaches are applicable large-scale graphs since all these approaches are mainly based on multi-core systems. A similar work, related to this thesis has been recently studied in [23]. In this paper, authors target to compare six different graph processing platforms including the platforms in this thesis scope with multiple experiments. However, the list of algorithms that are experimented excluded centrality measures.
Chapter 7

Conclusion

This thesis has shown the significance of centrality measures in network analysis. We provided a comparative study on a selection of the existing platforms available for distributed parallel data processing with respect to large scale centrality computation. The present study makes several noteworthy contributions to understand the challenges involved in implementing and evaluating the computation intensive graph algorithms (Effective Closeness and LineRank) in two different computing platforms such as Apache Flink and Apache Giraph. We will now summarize the results and findings in a way to answer the questions introduced in the beginning of this thesis.

Apache Flink provides a comprehensive computing platform that supports data processing with a high user friendly programming interface. This flexibility provides a natural extension for data pre-processing activities while implementing graph iterative algorithms. On the other hand, since Apache Giraph is a graph processing platform it compels the user to first convert the raw input data into an implementation specific format by using other generic data processing platforms (Apache Hadoop, Apache Flink, etc.,).

While implementing these algorithms in Apache Flink and Apache Giraph, we recognized that Apache Flink provides a novel approach for handling custom data types that makes easier for the user to focus on computation logic without taking much care about the serialization part. In case of Apache Giraph, it requires the user to manage these serializations and also, the user needs to write implementation
specific custom input format.

The implementations of each of these centrality algorithms (Effective Closeness and LineRank) in Apache Flink are more straightforward and simple. In particular, the computation flow of Effective Closeness algorithm naturally fits into the delta-iteration operator in Apache Flink as discussed in Chapter 4. The LineRank algorithm is easier to implement with a programming interface that supports functional programs as it contains multiple join and reduce operations as the computation part of iteration. Since Apache Flink is more expressive for these kinds of operations, the effort for implementation is relatively much less compared to its counterpart in Apache Giraph.

Implementation of LineRank algorithm in Apache Giraph requires an in-depth knowledge in both platform level and algorithmic level (strong background on graph mathematics). Although it requires much time to re-model the algorithm in order to fit into the vertex centric model, we found a novel approach to implement it in Apache Giraph in such a way that the implementation leverages the core logic of LineRank algorithm for the primary computations as discussed in Chapter 4. This consequently reduces the computation complexity of the LineRank algorithm in Apache Giraph as compared to Apache Flink.

The most obvious finding to emerge from the evaluation of these two algorithms on both the platforms is that, Apache Giraph performs better compared to the Apache Flink in terms of scalability. In case of Effective Closeness, Apache Flink is observed to perform near equivalent to Apache Giraph in some cases and in general the later $1.7 \times$ faster than the former. On the other hand, for LineRank algorithm in Apache Giraph runs $3.5 \times$ faster than LineRank in Apache Flink.

The findings suggest that, Apache Giraph currently supports an efficient mechanism (hash based) for message aggregations during iterations compared to Apache Flink which uses sorting technique for aggregations. Another reason is that the optimization strategy for estimating memory requirements of computations in Apache Giraph is more efficient compared to its counterpart as discussed in Chapter 5. In addition to that, since the computation complexity of LineRank algorithm is ab-
stracted by the novel approach that we have implemented in this thesis also adds more advantage to Apache Giraph in a way to execute much faster.

We conclude by saying that both the systems have limitations in different aspects while implementing complex algorithms. In case of computation intensive centrality algorithms (Effective Closeness and LineRank), Apache Giraph wins over Apache Flink and proves to the fact that it works very well for what it is designed for (graph processing). Apache Flink is more generic and the programmability with this platform is simple and flexible. It is also good to see that Apache Flink provides an extensive support for iterative algorithms unlike other generic data processing systems.

7.1 Future Work

The work on the LineRank algorithm implementation can be extended to verify whether it is feasible to compute the final aggregation part of the algorithm in Apache Giraph. Moreover, this thesis work can be extended to evaluate the computation intensive centrality algorithms on other parallel data processing systems such as Apache Spark and Distributed GraphLab. More research can be performed in the direction of memory utilization of the computation in each iteration to track the in-depth details of execution in both these systems.
Appendix A

Appendix

The Java code for the two centrality measures (Effective Closeness and LineRank) implemented in Apache Flink can be found following github link. It also includes the utility code snippets written for data preprocessing such as converting the raw data into the required format.

A.1 Effective Closeness in Flink

EffectiveCloseness

A.2 LineRank in Flink

LineRank_Flink

A.3 LineRank in Giraph

This implementation is build on top of previous work done by Sebastian Schelter that can be found at LineRank_Giraph
A.4 Generating Giraph input format

Giraph Input Format Generation

A.5 MATLAB Script

```matlab
load source_incidence.csv
load target_incidence.csv
S = spconvert(source_incidence);
T = spconvert(target_incidence);
n = 10;
m = 19;
d1 = S’ * ones(m, 1);
d2 = T * d1;
d = 1 ./ d2;
v = rand(m, 1);
r = ones(m, 1) / m;
diff = 1;
while diff > 0.00001
    v1 = d .* v;
v2 = S’ * v1;
v3 = T * v2;
v_next = .85 * v3 + .15 * r;
diff = norm(v - v_next, 2);
v = v_next;
end
lineranks = (S + T)’ * v;
```

Listing A.1: LineRank

A.6 Column Normalization in LineRank Algorithm

Consider the following adjacency matrix L(G) of the example graph 3-1.
Standard method of computing Column normalization

1. Consider the following adjacency matrix of \( L(G) \):

\[
\begin{array}{ccccc}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & 1 & 0 & 0 & 1 \\
2 & 1 & 0 & 1 & 0 & 0 \\
3 & 0 & 0 & 0 & 1 & 0 \\
4 & 1 & 0 & 1 & 0 & 0 \\
5 & 0 & 0 & 0 & 1 & 0 \\
\end{array}
\]

2. The corresponding transition probability matrix is obtained by transposing the above matrix and dividing each element by the sum of its corresponding column values.

\[
\begin{array}{ccccc}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & 1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0 & 0 & 0 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 0 & 1 & 0 & 1 \\
5 & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]

The sum of the columns = \( \begin{array}{c} 2 \quad 2 \quad 1 \quad 2 \quad 1 \end{array} \)

3. The transition probability matrix of \( L(G) \):

\[
\begin{array}{ccccc}
1 & 2 & 3 & 4 & 5 \\
1 & 0 & 0.5 & 0 & 0.5 & 0 \\
2 & 0.5 & 0 & 0 & 0 & 0 \\
3 & 0 & 0.5 & 0 & 0.5 & 0 \\
4 & 0 & 0 & 1 & 0 & 1 \\
5 & 0.5 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Column Normalization of $L(G)$ in LineRank:

Consider the same adjacency matrix $L(G)$. The author tried to compute the vector shown below next the matrix. This vector is the normalizing vector which will be multiplied with vector in the iteration during power iteration method. The result of the standard method and following representation are same.

$$
\begin{bmatrix}
1 & 0 & 1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0 & 0 & 0 \\
3 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 0 & 1 & 0 & 1 \\
5 & 1 & 0 & 0 & 0 & 0 \\
\end{bmatrix}
\begin{bmatrix}
0.5 \\
0.5 \\
1 \\
0.5 \\
1 \\
\end{bmatrix}
$$

But since the algorithm is trying to not compute the line graph, the author achieves the above computation efficiently with the decomposed matrices

$$T(G)$$

$$
\begin{bmatrix}
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0 \\
3 & 0 & 0 & 1 \\
4 & 1 & 0 & 0 \\
5 & 0 & 0 & 1 \\
\end{bmatrix}
$$

$$T(G)^* d1$$

$$
\begin{bmatrix}
1 & 2 & 3 \\
1 & 0 & 1 & 0 \\
2 & 1 & 0 & 0 \\
3 & 0 & 0 & 1 \\
4 & 1 & 0 & 0 \\
5 & 0 & 0 & 1 \\
\end{bmatrix}
\begin{bmatrix}
2 \\
2 \\
2 \\
1 \\
2 \\
1 \\
\end{bmatrix}
$$

The inverse of this highlighted vector is equal to our initial goal vector. This way the normalized vector is pre-computed and multiplied with the vector in the iteration that produces the edge score vector.
Contents

1 Introduction 5
   1.1 Motivation ........................................ 5
   1.2 Goal ............................................. 8
   1.3 Outline ......................................... 9

2 Distributed Graph Processing 10
   2.1 Generic Data Parallel Systems ....................... 11
      2.1.1 Apache Hadoop ................................ 11
      2.1.2 HaLoop ....................................... 12
      2.1.3 Twister ....................................... 13
      2.1.4 Apache Spark ................................ 13
      2.1.5 Apache Flink ................................ 14
   2.2 Graph Parallel Systems .............................. 17
      2.2.1 Pregel ........................................ 17
      2.2.2 Apache Giraph ................................ 19
      2.2.3 Distributed GraphLab ........................... 20
   2.3 Comparison of Large Scale Parallel Processing Systems .... 20

3 Scalable Centrality Measures 23
   3.1 Graph Theoretical Background ....................... 23
      3.1.1 Transition Probability Matrix .................. 24
      3.1.2 Stationary Distributions ....................... 25
      3.1.3 Power Iteration ................................ 25
3.1.4 Column Normalization ........................................ 26

3.2 Centrality Measures ............................................. 26
  3.2.1 Closeness .................................................. 27
  3.2.2 Scalable Closeness ....................................... 28
  3.2.3 Flajolet-Martin Sketch .................................. 29
  3.2.4 Betweenness .............................................. 30
  3.2.5 Scalable Betweenness .................................... 31

3.3 Analogy between PageRank and LineRank .................... 33

4 Implementation .................................................. 34
  4.1 Effective Closeness in Flink ................................ 34
  4.2 Effective Closeness in Giraph ............................. 38
  4.3 LineRank in Flink .......................................... 38
    4.3.1 Building Incidence Matrix ............................. 39
    4.3.2 Computing Normalized Vector ......................... 40
    4.3.3 Power Iteration ....................................... 41
    4.3.4 Aggregation of incident edges ......................... 44
  4.4 LineRank in Apache Giraph ................................ 44

5 Evaluation ...................................................... 51
  5.1 Experimental Setup and Datasets ........................... 51
  5.2 Preliminaries of accuracy estimation ....................... 52
  5.3 Accuracy .................................................... 53
    5.3.1 Effective Closeness .................................. 54
    5.3.2 LineRank .............................................. 55
  5.4 Scalability .................................................. 56
    5.4.1 Effective Closeness .................................. 56
    5.4.2 LineRank .............................................. 61
    5.4.3 Summary ............................................... 63
  5.5 Analysis ...................................................... 65
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>BSP</td>
<td>18</td>
</tr>
<tr>
<td>2-2</td>
<td>Compact Routing Example</td>
<td>19</td>
</tr>
<tr>
<td>3-1</td>
<td>Example Graph</td>
<td>24</td>
</tr>
<tr>
<td>3-2</td>
<td>Example graph of 6 nodes</td>
<td>28</td>
</tr>
<tr>
<td>3-3</td>
<td>Shortest Path Computation</td>
<td>28</td>
</tr>
<tr>
<td>3-4</td>
<td>Scalable Shortest Path Computation</td>
<td>29</td>
</tr>
<tr>
<td>3-5</td>
<td>Betweenness computation using Shortest Paths algorithm</td>
<td>31</td>
</tr>
<tr>
<td>3-6</td>
<td>Betweenness computation using Shortest Paths algorithm</td>
<td>32</td>
</tr>
<tr>
<td>4-1</td>
<td>The Effective Closeness algorithms as an Incremental Iteration</td>
<td>35</td>
</tr>
<tr>
<td>4-2</td>
<td>The Effective Closeness algorithms as an Incremental Iteration</td>
<td>36</td>
</tr>
<tr>
<td>4-3</td>
<td>The Effective Closeness algorithms as an Incremental Iteration</td>
<td>39</td>
</tr>
<tr>
<td>4-4</td>
<td></td>
<td>48</td>
</tr>
<tr>
<td>5-1</td>
<td>Correlation analysis of Effective Closeness for DBLP Prolific authors</td>
<td>55</td>
</tr>
<tr>
<td>5-2</td>
<td>Correlation analysis of LineRank(v2) for DBLP Prolific authors</td>
<td>56</td>
</tr>
<tr>
<td>5-3</td>
<td>Effective Closeness in Giraph: Runtime vs Edges</td>
<td>57</td>
</tr>
<tr>
<td>5-4</td>
<td>Effective Closeness in Giraph: Comparing the running time of each iteration for EU-Web and Youtube datasets</td>
<td>58</td>
</tr>
<tr>
<td>5-5</td>
<td>Effective Closeness in Flink: Runtime vs Edges</td>
<td>59</td>
</tr>
<tr>
<td>5-6</td>
<td>Comparison of Effective Closeness in Apache Giraph and Apache Flink: Runtime vs Number of cores</td>
<td>60</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Comparison of Large Scale Distributed Parallel Processing Systems in the context of support for iterative algorithms ................................................. 21

5.1 Summary of Datasets and its main properties ........................................... 52
5.2 Effective Closeness in Flink vs BitField Closeness ................................. 54
5.3 Effective Closeness in Giraph vs BitField Closeness ............................. 54
5.4 Correlation coefficients of LineRank(v2) in both Giraph and Flink is realized to have same values and represented above against Standard LineRank(v2) from MatLab ...................................................... 55
5.5 Correlation coefficients of LineRank(v2) vs Full LineRank ...................... 56
Bibliography


[38] RECOMMENDATION USING PRODUCT NETWORK. An improvement to e-commerce recommendation using product network analysis.


