Link Prediction in Large Scale Social Networks Using Hadoop

Agis Chartsias

Department of Electronic and Computer Engineering
Technical University of Crete

Advisor Professor: Professor Minos Garofalakis

Evaluation Committee:
Professor Minos Garofalakis
Assistant Professor Antonios Deligiannakis
Assistant Professor Michail G. Lagoudakis

July 23, 2010

Acknowledgements

This thesis would not have been possible without the guidance and the help of several individuals who in one way or another contributed and offered their valuable assistance in the preparation and completion of this study.

First and foremost, my utmost gratitude to my advisor, Professor Minos Garofalakis for his supervision, advise and guidance from the very early stage of this research. I am grateful for his encouragement and precious contribution throughout the elaboration of this study. I would also like to thank him for giving me the opportunity to work on this very interesting field of research. I am indebted to him more than he knows.

I gratefully acknowledge Assistant Professor Antonios Deligiannakis for his valuable advice in science discussion, using his precious times to help me through various obstacles that were presented.

Also, I would like to thank Assistant Professor Michail G. Lagoudakis who agreed to evaluate my diploma thesis.

Moreover, I would like to thank my laboratory colleagues for their patience and constructive comments.

I would like to thank all my friends for these great years we spent together and for many wonderful memories.

Most of all, I would like to thank my family for their enormous help, understanding and support throughout all these years as a student.

Abstract

One fundamental data mining task in the structural analysis of a social network is the ability to predict future links between its members. This task is formalized as the "link prediction problem". State-of-the-art solutions to the link prediction problem rely on analyzing the topological properties of the social graph and can be computationally expensive; coupled with the massive size of real-world social networks, this mandates the use of scalable parallel algorithms. In this thesis, we present the novel design and implementation of two state-of-the-art link prediction methods (based on node-neighborhood intersections and path counting) in the highly-scalable Map/Reduce parallel framework, using the Hadoop open-source implementation. Our experimental results with several large, real-life social graphs on SoftNet's Hadoop cluster verify the effectiveness of our approach.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Network science is a scientific discipline that studies network representations of physical, biological and social phenomena and seeks to discover common principles that govern network behavior. A network is a set of entities, which are pairwise connected with links. In computer science, networks are represented as graphs, where the entities correspond to vertices which are connected with edges. Networks model different forms of associations among entities and we see many examples in the real world. Examples include the World Wide Web, where the vertices are the web pages and links from one page to another form edges; social networks, where the vertices are people and edges express some sort of acquaintance like friendship or relativity; co-authorship networks, where the vertices are scientists in a particular discipline and edges connect those who have co-authored a paper; collaboration networks, where the vertices are employees and edges are formed between those who have worked in common projects; biological networks that express relations among proteins or neurons. (Newman [3] has a thorough study on networks).

The goal of network analysts is to model the interactions among the entities and discover interesting patterns, by focusing on the properties of real world networks. This is the area of research knows as graph mining. Patterns that have been discovered include the small world effect, the shrinking diameter and many others. One important property of networks is that they evolve over time with edges appearing or disappearing. The problem of predicting a future snapshot of a graph is formulated as the link prediction problem. Many algorihms have been introduced that solve this problem taking advantage of the structure of the graph.

One problem with the study of real world networks is that they are extremely large, extending from hundreds of edges to billions of edges (Yahoo Web). Obviously, it is difficult to apply sequential algorithms to analyze these graphs. This size restraint has lead to the development of parallelization architectures. One recent and effective framework that permits the development of parallelized algorithms is Hadoop. Hadoop provides us with a distributed filesystem and the implementation of the map/reduce progamming model, as well as all the necessary libraries that are needed in order for a compute cluster to function. Its main advantage is that it separates the parallelization code from the business logic, thus making easy for anyone to create and execute a parallel algorithm. Additionally it poses no restrictions regarding the number of computer nodes that the cluster should have, something that has been an issue in older architectures.

The object of our work is solving the link prediction problem with the development of parallel algorithms in the map/reduce model. We implemented two algorithms who

take advantage of the topological structure of the graph, by examining paths. For every pair of vertices, they calculate a score, which refers to the probability that these two vertices will form a link. By defining a threshold, one can create a future snapshot of the graph, by adding links between vertices that their proximity score exceeds this threshold. The chanllenge, has not been to create and evaluate a new link prediction method, but to parallelize an existing one in Hadoop API, so that it can be used for large datasets.

The first algorithm is based on the Jaccard coefficient. For two nodes, the Jaccard coefficient compares their neighborhoods. It is defined as the fraction of the intersection of their neighborhoods, divided by the union of their neighborhoods. We extended this definition by extending the neighborhoods of the nodes at a given depth d. The new neighborhoods will contain vertices that are d hops away of the source vertex. In this way we have improved the quality of the predictor, as the new probability takes into account longer paths. If there is a probability of connection between two nodes based on paths of length 1, then the probability is heightened if we examine longer paths.

The second algorithm is based on the Katz index. For two vertices, the Katz index provides a score by counting the total number of paths between them. The paths are given a weight according to their length.

This thesis is organized as follows. Chapter 2 describes the background knowledge and related work regarding our work. This includes graph theory (2.1), graph mining (2.2), link prediction 2.3, Jaccard Coefficient (2.4), Katz Index (2.5) and the Hadoop framework (2.6). The description of our Map/Reduce algorithm of Extended Jaccard Coefficient lies in Chapter 3 and the one of Katz Index in Chapter 4. Finally, Chapter 5 contains the experiments for every Map/Reduce algorithm that we conducted in several real graphs, along with interesting results.

Chapter 2

Background and Related Work

2.1 Graph Theory

Graphs are mathematical structures used to model pairwise relations between objects from a certain collection. A graph is a collection of objects, where some pairs of the objects are connected by links. The objects are called vertices or nodes and the links are called edges or arcs. The edges may be directed (assymetric) or undirected (symmetric). The corresponding graphs are called directed or digraphs and undirected graphs. Of course, we can represent an undirected graph as directed if we have two edges between every pair of nodes, one for each direction. The edges may carry weights, that could represent costs, length, capacities or other quantities depending on the problem. These edges define a graph as weighted. Graphs can be either cyclic, meaning they contain closed loops of edges or acyclic meaning they do not. Also, we can define a subgraph of a graph G, which is a graph whose vertex set is a subset of G, and whose adjacency relation is a subset of G restricted to the new vertex subset. In the other direction, a supergraph of a graph G is a graph of which G is a subgraph. Graphs may also be partitioned in natural ways. Bipartite graphs are graphs whose vertices are divided into two disjoint sets U and V, such that every edge connects a vertex from U to one in V.

2.1.1 Graph Representation

The two most commonly used data structures for representing a graph $G = (V, E)$ are the adjacency list and the adjacency matrix.

The adjacency list is implemented as an array of $|V|$ lists, with one list of destination nodes for each source node. The vertices in each adjacency list are typically stored in an arbitrary order. In directed graphs the sum of the lengths of all adjacency lists is $|E|$, while in undirected graphs it is $2 \cdot |E|$. This happens because in undirected graphs, each edge (u, v), is stored in the list of node u, as well as in the list of node v.

The adjacency matrix is a two dimensional boolean matrix, of length $|V| \times |V|$. It is assumed that the identities of vertices vary from $0 \ldots |V|$. A matrix entry (i, j) indicates if there is an edge from vertex i to j. Formally, we define the adjacency matrix $A = (a_{ij})$, where $a_{ij} = \begin{cases} 1 & if\ (i,j) \in E, \\ 0 & otherwise \end{cases}$. Adjacency matrices of undirected graphs are symmetric, as for every edge (i, j), there also exists an edge (j, i). The

transposed adjacency matrix of $A = (a_{ij})$ is the matrix $A^T = (a_{ij}^T) = (a_{ji})$.

Adjacency lists are usually preferable for sparse matrices, where $|E| << |V|^2$ [8], because they occupy less space, as they do not use any space for edges that are not present. Respectively, adjacency matrices are preferred when the graph is dense and $|E| \simeq |V|^2$. Because each entry of matrix requires one bit, they can be represented in a compact way occupying $|V|^2/8$ bytes. The adjacency matrix requires $\Theta(V^2)$ memory, independent of the number of the edges in the graph, while the adjacency list requires $\Theta(V + E)$ memory. Although the adjacency list representation is asymptotically at least as efficient as the adjacency matrix representation, the simplicity of an adjacency matrix may make it preferable when graphs are reasonably small. For a more extensive study on graph theory see [7].

2.1.2 Definitions and Graph Metrics

Some important definitions and metrics that are used in network analysis are:

- **path**: an alternating sequence of vertices and edges, beginning and ending with a vertex, where each vertex is incident to both the edge that precedes it and the edge that follows it in the sequence. The length of a path is the number of edges traversed.

- **degree**: number of edges incident to the vertex. The degree is not necessarily equal to the number of vertices adjacent to a vertex, since there may be more than one edge between any two vertices. Such a graph is called a multigraph. In directed graphs, there is in-degree and out-degree for every vertex, which are the numbers of incoming and outgoing edges respectively.

- **diameter**: the greatest distance between any pair of vertices; it is equal to the length of the longest shortest path between any two vertices.

- **betweenness of node i**: the number of shortest paths between pairs of other vertices that run through i. It is a measure of the influence of a node over the flow of information between other nodes, especially in cases where information flow over a network follows the shortest available path. [9].

- **clustering coefficient**: the probability that a connected triple of nodes is actually a triangle. It describes the tendency to form clusters (fully connected subgraphs) in a graph and is a measure of the likelihood that two associates of a node are associates themselves. [10, 11]. Formally, it is equal to:

$$C = \frac{3 \times number\ of\ triangles}{number\ of\ connected\ triples\ of\ vertices} \tag{2.1}$$

- **connected component**: a maximal connected subgraph, meaning a subgraph in which any two vertices are connected to each other by paths. In a directed graph each vertex has both an out-component and in-component, which are the sets of vertices that it can reach and it can be reached from.

2.2 Graph Mining

Graph mining is the process of finding and extracting useful information like patterns from structured data that can be represented as a graph. It helps us in interpreting

the laws under which behave real-world networks and understand the structure of links.

Important patterns regarding the structure and function of networks arise from the study of the basic metrics that we described in Section 2.1.2.

Milgram developed a theory in 1967 [12, 13] suggesting that human society is a small world type network characterized by short path lengths. The Small World Effect, which is associated with the phrase "six degrees of separation", is the phenomenon that the average diameter of a graph is small, typically around 6 [14, 15, 16]. Leskovec et al. [17] proved that in real graphs the diameter shrinks and stabilizes over time as the graph grows. Recent work by Kang et al. [18, 30] has used the Hadoop framework in order to compute the diameter and effective diameter of massive graphs of Tera and Peta-byte size, achieving excellent scale up.

Typically, real networks have degree distributions which are highly right skewed, meaning that most of the vertices have low-degree, but there is a small number of nodes with high degree and are known as "hubs". Often, the degree distributions of some networks like the World Wide Web or social networks follow a power law distribution or exponential form [19, 20, 21, 22, 23]. The densification power law is described by the relation $P(k) \sim k^{-a}$, where a is a constant. Time evolving graphs also follow the densification power law, with the equation $e(t) \sim n(t)^a$, where $e(t)$ are the edges and $n(t)$ are the nodes, according to [17].

As can be seen from Equation (2.1), in order to compute the clustering coefficient for a graph, first we need to count the triangles that exist. Several algorithms have been proposed for counting triangles in graphs [32]. Tsourakakis et al. [33] developed a hadoop algorithm, named "Doulion" that counts triangles in large scale graphs.

A giant connected component (GCC) is formed nearly at all real networks; that is a set of nodes that consists the majority of the graph. In the graph generator of Erdos-Renyi (1960), edges are randomly inserted between nodes and at a critical point there is a high probability for a GCC to emerge in the graph ([24]). Kumar et al. [25] and McGlohon et al. [31] study components other than the GCC. Some large-scale algorithms for detecting connected components include [26, 27, 28, 29].

Recently, Faloutsos et al. [30] released "PEGASUS" (Peta-Scale Graph Mining System), which implements several algorithms in Hadoop, one of which is detecting connected components. The other graph mining algorithms that are implemented in this project are diameter estimation, PageRank and Random Walks with Restarts (RWR). The main idea is that all the four algorithms can be solved using matrix-vector multiplications. So, the "heart" of their project is the parallelization of this operation. Using the parallelized matrix-vector multiplication in hadoop, they managed to implement these graph mining tasks.

2.3 Link Prediction

Link prediction is a graph mining task that aims to predict the occurence of new edges in a graph after a certain interval of time. Given a snapshot of a graph at a timestamp t, the goal is to predict the evolution of the graph at a timestamp $t' > t$. The heart of the problem is to compute scores for every pair of non-connected nodes, which makes it an $O(n^2)$ problem.

The structural features of a graph can provide sufficient information that can be used to predict new links. Liben-Nowell and Kleinberg have published a survey [34] through which, they analyze and compare some of the most important topological metrics that can be used for link prediction. Each metric computes a score for each pair of unconnected vertices. This score stands for the possibility that two vertices

will connect in the future. The methods are divided into two categories; those that are based on neighborhoods and those that are based on paths. The first are also called local and the second global methods. Local methods predict a link occurence by examining the number of common neighbors or paths of length 2. Global methods explore longer paths and predict a link occurence based on the sum of the total number of paths, which are weighted by their length. Hence, the global methods are natural extensions of the local ones. Let $G(V, E)$ be a graph. In order to define the methods that are based on the neighborhoods of the nodes, let us suppose that $\Gamma(x)$ and $\Gamma(y)$ are the neighborhoods of two nodes x, $y \in G(V, E)$.

The methods that are based on the neighborhoods of the nodes include:

- Common Neighbors: $score(x, y) = |\Gamma(x) \cap \Gamma(y)|$.
- Jaccard Coefficient: $score(x, y) = \frac{|\Gamma(x) \cap \Gamma(y)|}{|\Gamma(x) \cup \Gamma(y)|}$.
- Adamic/Adar Coefficient ([35]): $score(x, y) = \sum_z \frac{1}{\log(frequency(z))}$, where z is a feature shared by x, y.
- Preferential Attachment ([36, 37]): $score(x, y) = |\Gamma(x)| \cdot |\Gamma(y)|$

The methods that are based on paths include:

- Katz [5]: $score(x, y) = \sum_{l=1}^{\infty} \beta^l \cdot |paths_{x,y}^{\langle l \rangle}|$, where l is the length of a path, β a user defined variable and $|paths_{x,y}^{\langle l \rangle}|$ is the number of paths of length l between x and y.
- Hitting Time: $H_{x,y}$ is the expected number of steps for a random walk starting from x to reach y.
- Symmetric Commute Time: $C_{x,y} = H_{x,y} + H_{y,x}$
- rooted PageRank: the probability of y in a random walk that returns to x with probability a at each step, moving to a random neighbor with probability $1 - a$.
- SimRank ([38]): $score(x, y) = \gamma \cdot \frac{\Sigma_{a \in \Gamma(x)} \Sigma_{b \in \Gamma(y)} s(a,b)}{|\Gamma(x)| \cdot |\Gamma(y)|}$, where $\gamma \in [0, 1]$ and $s(a, b) = 1$, if $a = b$.

We elaborate further on Jaccard Coefficient and Katz, as they play a critical role in the thesis.

The clustering coefficient quantifies the effect of clustering or transitivity that networks have. This property states that two vertices that are both neighbors to a third vertex have a higher probability of also being neighbors to one another. So, the clustering coefficient is a natural predictor of links. Another topological measure that can be used to predict links more effectively than the clustering coefficient is the generalized clustering coefficient according to [39]. Unlike the clustering coefficient (2.1), it measures the formation of cycles instead of triangles. It also expresses the proximity of two nodes by examining paths of length k instead of 2. It is defined as:

$$C(k) = \frac{number\ of\ cycles\ of\ length\ k}{number\ of\ paths\ of\ length\ k} \tag{2.2}$$

Tong et al. defined in [40] a node-node and a group-group proximity based on random walks, where the second refers to the proximity of two groups of nodes. Let $G(V, E)$ be a graph. A random walk starts from a node x and randomly moves

to a neighbor of y with a probability c. The proximity $Prox(x, y)$ of two nodes $x, y \in G(V, E)$ is defined as the probability for a random walk that starts from x, to visit y before it returns to x. Suppose that the graph is directed. Then, the prediction of a link between nodes x and y would depend on the relation $Prox(x, y) + Prox(y, x) > threshold$. If the group-group proximity is used, then a link between x and y would depend on $Prox(N(x), N(y)) + Prox(N(y), N(x)) > threshold$, if $N(x)$ and $N(y)$ are the neighborhoods of x and y resprectively. Finally, the direction of an edge between a linked pair can be predicted by the result of the two equations $Prox(x, y) > Prox(y, x)$ and $Prox(x, y) > Prox(y, x)$. In the first case, the edge goes from x to y and in the second it goes backwards.

O'Madadhain et al. [41] construct local conditional probability models, based on attribute and structural features. They predict the participation of actors in events. The structural features that are used for prediction are the one described in [34].

A number of approaches define a probabilistic model over the graph that can be used for link prediction among other applications. These approaches perform probabilistic inference and capture the correlations among the links. The models could be based on Markov random fields [43] or on relational representations like Relational Markov Networks [44] and Markov Logic Networks [45].

2.4 Jaccard Coefficient

The Jaccard similarity coefficient, also known as the Jaccard index (coefficient de communauté by Paul Jaccard [4]), is a statistical measure used for comparing the similarity of sample sets and it is widely used in information retrieval [6].

Let s_1 and s_2 be two sets. Then, the jaccard coefficient measures the number of "features" that both s_1 and s_2 have, compared to the number of features that either s_1 or s_2 has. Let $G(V, E)$ be a graph and two nodes $x, y \in G(V, E)$. If the two sets s_1 and s_2 represent the neighborhoods $\Gamma(x)$ and $\Gamma(y)$ of x and y respectively, then we can consider as "features" the neighbors of the nodes. In this case we define the jaccard coefficient as the cardinality of the union of the neighborhoods of x and y, divided by the cardinality of the intersection of the neighborhoods of x and y.

$$J(x, y) = \frac{|\Gamma(x) \cap \Gamma(y)|}{|\Gamma(x) \cup \Gamma(y)|} \tag{2.3}$$

J. Bank and B. Cole [46] have created a map/reduce algorithm that calculates the jaccard similarity coefficient. They use as dataset the pages of wikipedia and run their algorithm twice. The first computes the similarity between pages and the second computes the similarity between users.

As explained in section 2.3, the Jaccard coefficient can be used for link prediction. It is a similarity score between two nodes, that expresses the possibility of the appearance of a potential edge between these nodes [34].

2.5 Katz Score

The Katz index was originally used in the field of sociology as a popularity index between two subjects. Examining a social group, we can create a graph representing it with nodes to be people and edges to appear between individuals that are somehow related. The Katz index provides a score for every pair of actors, by counting the total connections between them. Each connection, however, is given a weight, according to its length. The greater the length, the weaker the connection. How much weaker

the connection becomes with increasing length depends on an attenuation factor. Mathematically, the Katz index for two nodes x and y that belong to a graph $G(V, E)$ is expressed by the formula:

$$score(x, y) = \sum_{l=1}^{\infty} \beta^l \cdot |paths_{x,y}^{\langle l \rangle}| \tag{2.4}$$

As explained in section 2.3, the Katz index can be used in link prediction. This status is indicative of the similarity of two nodes and can determine a potential edge [34].

Acar et al. [47] used the Katz method for link prediction in weighted and unweighted bipartite graphs. Matrix and tensor based methods were used. A low-rank approximation based on truncated Singular Value Decomposition was the basis upon the matrix-based methods. For the tensor-based methods they considered a CANDECOMP/PARAFAC (CP) decomposition. Note that the Katz scores for all pairs of nodes can be expressed in matrix terms as follows. Let $\hat{A}$ be the adjacency matrix of the graph and $\hat{S}$ the matrix with the Katz scores, then $\hat{S} = \sum_{l=1}^{\infty} \beta^l \cdot \hat{A}^l = (I - \beta \cdot \hat{A})^{-1} - I$.

The above matrix solution of the Katz index is interpreted in the following way. For matrices whose elements are 0 or 1, powers of A have as elements the numbers of chains of corresponding lengths going from i through intermediaries to j. Thus, $A^2 = (a_{ij}^2)$, where $a_{ij}^2 = \sum_k a_{ik} \cdot a_{ki}$; each component, $a_{ik} \cdot a_{ki}$, of a_{ij}^2 is equal to one if and only if i chooses k and k chooses j, i.e., there is a chain of length two from i to j. Higher powers of A have similar interpretations. The column sums of A give the numbers of direct links between all the nodes and the individual corresponding to each column. Also, the column sums of A^2 give the numbers of two-step links; column sums of A^3, numbers of three-step links etc. As a result, the Katz index may be constructed by adding to the direct links all the two-step, three-step links etc, using appropriate weights, which are represented by the parameter β^l. Hence, the matrix $\hat{S}$ that we seek is given by the relation $\hat{S} = \beta \cdot A + \beta^2 \cdot A^2 + \beta^3 \cdot A^3 + \ldots + \beta^l \cdot A^l = \hat{A})^{-1} - I$

2.6 Map/Reduce and Hadoop

Many computations that process a large amount of raw data, need to be distributed across several machines in order to finish in a reasonable amount of time. This issue can be addressed by an open source framework, called Hadoop [1] that supports data intensive distributed applications. Hadoop is a project of the Apache Software Foundation that parallelizes data processing across many nodes in a compute cluster, speeding up large computations and hiding I/O latency through increased concurrency. The advantages of this model lie in its ability to deal with the issues of distributing the data, handling failures, load balancing among the cluster, thus separating the business logic from the parallelization code; hence, developers are free to focus on application logic.

The hadoop project includes various subprojects that provide complemetary services. These are:

- Common: a set of utilities that support the other subprojects. It provides a distributed filesystem, RPC (Remote Procedure Call), persistent data structures, serialization libraries and support for the MapReduce distributed computing metaphor.

- MapReduce: a distributed data processing model and execution environment that runs on compute clusters.

- HDFS: a distributed filesystem that provides high throughput access to application data.

- Chukwa: a distributed data collection and analysis system.

- Hive: a data warehouse infrastructure that provides a query language based on SQL.

- Pig: a high level data flow language and execution framework for parallel computation. It is built on top of Common.

- ZooKeeper: a high performance coordination service for distributed applications.

Hadoop implements the MapReduce programming model [2]. The user of this library needs to implement two functions – map and reduce – to perform a computation. Each input record is converted into a key/value pair. A map operation is applied to each input record and produces a set of intermediate key/value pairs. The map outputs are grouped and sorted by key. A reduce operation is applied to all values that share the same key, in order to combine the derived data appropriately.

HDFS is a file system designed to store large files across multiple machines. Storage reliability is achieved with the data replication on several nodes. Three processes control the HDFS services. *Namenode* manages the filesystem namespace and regulates access to files by clients. It is a single point of failure for an HDFS installation, as if it goes down the system is offline. It is responsible for operations like opening, closing and renaming of files and directories available via an RPC interface. Also, it determines the mapping of blocks to Datanodes. *Secondary Namenode* is a process that regularly connects to the Namenode and downloads a snapshot of its directory information, which is then saved to a directory. The Secondary Namenode is used together with the edit log of the Namenode to create an up-to-date directory structure. *Datanode* is a process that provides block storage and retrieval services like serving read/write requests from clients and performing block creation, deletion and replication upon instruction from the Namenode.

The Hadoop framework provides two processes that handle the execution of MapReduce jobs. *TaskTracker* manages the execution of individual map and reduce tasks on a compute node in the cluster and *JobTracker* accepts job submissions, provides job monitoring and control and manages the distribution of tasks to the TaskTracker nodes.

When a MapReduce job is submitted by the user, it is decomposed into a number of tasks. The user is responsible for submitting the job configuration in order to provide the framework with a series of necessary parameters regarding the job, like the input and output destination in HDFS, the input and output format, the classes that contain the map and reduce functions and the JAR file(s) that contain the map and reduce functions and any support classes. Then, the input is splitted according to the HDFS block size (typically 64 MB) and distributed across the map tasks. If the input is N files, then at least there will be N map tasks. The map tasks are executed and produce the intermediate key/value pairs according to the map function that is specified by the user. Each map function receives one record (line) from the split and process it accordingly. Then, follows the shuffle phase where the map outputs are partitioned and sorted. The shuffle output for each partition is sorted. Afterwards,

the reduce tasks start with input the data that correspond to their partition. Each reduce function is called once for each input unique key with all the values that share that key. The reduce tasks emit key/value pairs, which are written to output directory. The number of output files in the directory will be as many as the number of reduce tasks that were executed.

A large variety of input formats are supplied by the framework. The major distinctions are between textual and binary input formats. The available formats are:

- KeyValueTextInputFormat: key/value pairs, one per line.
- TextInputFormat: the key is the byte offset of the line and the value is the line.
- NLineInputFormat: similar to KeyValueTextInputFormat, but the splits are based on N lines of input rather that Y bytes of input.
- MultiFileInputFormat: an abstract class that lets user implement an input format that aggregates multiple files into one split.
- SequenceFileInputFormat: the input file is a Hadoop sequence file, containing serialized key/value pairs.

Hadoop provides its own set of data types that are optimized for network serialization and correspond to the known Java built-in data types. Of course, the user can define custom data types if necessary. The data types that are used as keys need to implement the WritableComparable and the data types that are used as values need to implement the Writable interface, which is a subset of WritableComparable. The Writable interface implements the methods that are used for serialization and deserialization of the objects and the WritableComparable implements additionally the methods that are used for the comparison of the keys. The most common Hadoop data types are:

- Text: equivalent to String.
- IntWritable: equivalent to Integer.
- VIntWritable: used for integer values stored in variable-length format. Such values take between 1-5 bytes. Smaller values take fewer bytes.
- LongWritable: equivalent to Long.
- VLongWritable: used for long values stored in variable-length format. Such values take between 1-5 bytes. Smaller values take fewer bytes.
- FloatWritable: equivalent to Float.
- DoubleWritable: equvalent to Double.
- ByteWritable: equivalent to Byte.
- BytesWritable: used for byte arrays.
- BooleanWritable: equivalent to Boolean.
- NullWritable: equivalent to null.

The map and the reduce functions have 4 parameters. The key, the value, the output collector and the reporter. The output collector is the object used to emit the key/value pairs. The reporter object provides the mechanism for informing the framework of the current status of the job. If a job takes too long to complete, it is useful to inform the framework that it is still working through the reporter, so that the framework will not kill it.

Chapter 3

Extended Jaccard Algorithm

3.1 Introduction

As already mentioned in section 2.4, the Jaccard similarity score can be used as a proximity measure, providing us with the probability of two nodes to become neighbors. Thus, it can help us predict a future snapshot of the graph by linking the vertices that have a probability higher than a threshold.

The Jaccard coefficient is one of the measures that are based on the topological structure of the graph. This is shown by the fact that the score is depending on the neighborhoods of the nodes. Let $G(V, E)$ be a graph, x, y be two nodes such that $x, y \in G(V, E)$ and $\Gamma(x)$ and $\Gamma(y)$ be the neighborhoods of x and y. We define the Jaccard coefficient as the cardinality of the union of the neighborhoods of nodes x and y, divided by the cardinality of the intersection of the neighborhoods of x and y.

$$J\left(x, y\right) = \frac{|\Gamma\left(x\right) \cap \Gamma\left(y\right)|}{|\Gamma\left(x\right) \cup \Gamma\left(y\right)|} \tag{3.1}$$

The Jaccard coefficient is considered to belong to the local methods of link prediction. Recall here that local methods are those that examine paths of length 2, while global methods explore longer paths. We have extended the classic definition of the algorithm by calculating extended neighborhoods $\Gamma_d(x)$ at $1 \dots d$ hops for each node x, thus converting this predictor to a global method. The notion behind that is that, if links are likely to occur between nodes connected by a short path (of length 2), they might be more likely to occur between nodes connected by longer paths as well.

The new neighborhoods for each node, will contain the vertices that are $1 \dots d$ hops away. For example, a node $z \in G(V, E)$ belongs to the neighborhood of a node x if there is an outgoing edge from x to z ($x \rightarrow z$), or an outgoing path from x to z of some length ($x \rightarrow \dots \rightarrow z$). Furthermore, our work is specialized in directed graphs. However, it can easily be applied in undirected ones, simply by treating an undirected graph as directed; each edge between two vertices becomes two edges, one from the first vertex to the other and vice versa. However, this technique does not take advantage of the characteristic of symmetry that applies to an undirected network graph. The new equation that describes our algorithm is:

$$J(x, y) = \frac{|\Gamma_d(x) \cap \Gamma_d(y)|}{|\Gamma_d(x) \cup \Gamma_d(y)|} \tag{3.2}$$

The program runs with 6 parameters; the input file, the number of hops d, the number of nodes of the graph, the number of nodes that participate in edges, the

number of the reducers and the option to compress the output data or not. The input file must be in the form of an edge file e.g. $node1$ $node2$, where each line of the file declares an edge from the first node to the second. It is assumed that the identities of the nodes of the graph are positive integers. The number of hops d determines the depth of the neighborhoods to extend. The option of compression is applied to the output data that are binary, which can easily be compressed in order to save space.

The algorithm is programmed in chained Map/Reduce jobs. The fact that complicated calculations are needed in order to find and compare the neighborhoods at extended depths has imposed this code style. As described in section 3.2 we have divided our program in four Map/Reduce stages, where the output of each stage becomes the input for the following.

The notation that is used in the following sections to express the key/value pairs of the map/reduce framework is: the keys are underlined and the values are inside a parenthesis e.g. key, (value). Additional key notation is summarized in Table 3.1

Symbol	Definition
G	Directed graph
G_d	Directed graph with extended neighborhoods at $1 \dots d$ hops
V	Vertices
E	Edges
n	$\|V\|$ - number of nodes of the graph
m	$\|E\|$ - number of edges of the graph
n_x	number of edges that start from node x, i.e. number of outgoing neighbors of node x
n_R	number of reducers
d_{max}	max hop of neighborhoods
$d_{current}$	current hop of neighborhoods
$\Gamma(x)$	neighborhood of node x
$\Gamma_d(x)$	extended neighborhood of node x at d hops
A	Adjacency matrix of graph G
A_d	Adjacency matrix of graph G_d
$\vec{r}$	row of matrix A
$\vec{c_d}$	column of matrix A_d
$\vec{c_d^T}$	transposed column of matrix A_d
$c_d\vec{(k)}$	column k of matrix A_d
$c_d^T\vec{(k)}$	transposed column k of matrix A_d
M_k	Matrix that is produced from the outer product of $c_d\vec{(k)}$ and $c_d^T\vec{(k)}$
M	Sum of all matrices M_k
$r\vec{_M}$	row of matrix M
$r\vec{_{M_k}}$	row of matrix M_k
S_x	$\|\Gamma(X)\|$ - size of neighborhood of node x
$J(x, y)$	Jaccard coefficient of node pair x, y
P_i	Partition i that corresponds to reducer i

Table 3.1: Definions of Symbols and Acronyms

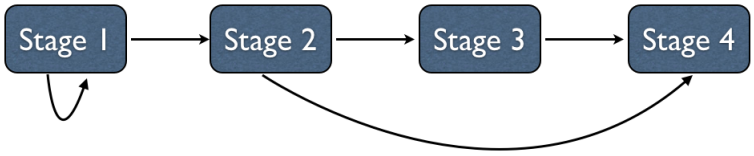

Figure 3.1: Extended Jaccard Coefficient Execution Flow

3.2 Description of the Algorithm

In this algorithm we want to find the extended neighborhoods of all nodes and calculate the intersection and the union of these neighborhoods for each pair.

The basis of our algorithm has been the work of Bank et al. [46], who have also created a hadoop algorithm for the simple Jaccard coefficient. The difference is that they only compare the direct neighborhoods of the nodes, while we extend the neighborhoods at various hops.

A figure summarizing the execution flow of our algorithm is 3.1.

We use one map/reduce stage to find the extended neighborhoods. This task is done by combining neighbors of depth 1 with neighbors of depth d to find neighbors of depth $d+1$. This code is stage 1 (3.2.3). This stage is called iteratively as many times as the depth of the neighborhoods that we want to find.

After the completion of stage 1, we have at our disposal the extended neighborhoods for all nodes. Actually, we have key/value pairs that represent edges of length $1 \dots d$. For each pair, the key contains the identity of the source node and the value the identity of the destination node. Each node pair contains additional information about the length and the direction of their link. What is left to do is assemble the values for each key in order to create its neighborhood and then combine the neighborhoods of all keys in a way that will give us the desired similarity score. The first task is easily done by another map/reduce stage, which is stage 2 (3.2.4). Obviously, we will need another map/reduce stage to do the second task, as we need to combine the keys with each other.

There are many possible solutions, that can handle the problem of calculating the Jaccard coefficient, having as input the neighborhood of each node. There are divided into two categories. The first includes solutions based on adjacency list representations and the second are matrix based ideas.

Let us suppose that we have a map/reduce stage, whose reduce method assembles destination nodes (values) that are fetched to a source nodes (key). This method can easily create a list containing these destination nodes for every source node. A list of this type is the neighborhood at depth $1 \dots d$ for a source node. Next, we want to somehow compare every neighborhood with all the others in order to compute the Jaccard coefficient for every possible node pair.

A possible implementation is to create composite keys, that represent a node pair. So, each reducer that receives an input key u will create $|V| - 1$ output keys, of which the first argument will be u and the second argument will be the the identity of every other node in $G(V, E)$. The output value for every key/value pair will be the neighborhood of node u. The key/value pairs emitted should comply with the restraint that there is not a 1-hop edge between u and v. In this way, we would not compute scores for connected node pairs. As a result for every key (u, v), there would be two values. The first would be the neighborhood of u and the second the neighborhood of v. Using another map/reduce stage, where the map method would

simply read and emit, we could easily compute the desired score. The reduce method, would receive, as we said, two values for every key (u, v), that would represent the neighborhoods of u and v. The comparison of the two neighborhoods can give us the number of their intersection and union, hence the Jaccard coefficient. Nevertheless, if $|V|$ is the number of the nodes of the graph and $|E|$ is the number of edges, then the total number of keys would be $(|V|-1)^2 - |E|$, assuming that the graph is not a multigraph (there are not parallel edges that connect the same nodes). Unfortunately, this method is extremely inefficient because of the large number of key/value pairs emitted. Each node has its neighborhood emitted as many times as the number of the unconnected nodes that it has and this is inefficient, as the data transfered is enormous in comparison to the graph size. Even if the neighborhoods are encoded in bitmaps, the I/O operations for this amount of data is a bottleneck for the algorithm.

Another way is to use a single reducer that would read all the nodes with their neighborhoods and could then compare them in order to find the jaccard coefficient for the unconnected pairs. This method is also problematic since it places a limit to the size of the graph that we could use, depending on the available memory of the computer, as the reducer would have to store all the neighborhoods.

The procedure that we followed is slightly different and is based on the matrix representation of the graph $G(V, E)$. The neighborhoods can easily be represented as an adjacency matrix A_d, with size $[N \times N]$, if N is the number of the nodes of the graph. A_d holds the neighbors at depth $1 \dots d$ for each node. The goal is to create a matrix M, with size $[N \times N]$, that would contain the intersections of the neighborhoods.

Our proposal is this. Each element (i, j) of the matrix M is the inner product of a row $r_d(\vec{i})$ the transposed $r_d^T(\vec{j})$. So, all we need to do is multiply matrix A_d with its transposed A_d^T.

The matrix A_d, is made in stage 2 (3.2.4) and the matrix M in stage 3 (3.2.5).

The union of the neighborhoods is also needed in order to compute the jaccard coefficient. The union for two nodes x and y is found using the equation:

$$|\Gamma_d(x) \cup \Gamma_d(y)| = |\Gamma_d(x)| + |\Gamma_d(y)| - |\Gamma_d(x) \cap \Gamma_d(y)| \tag{3.3}$$

Consequently, we need to find $|\Gamma_d(x)|$ and $|\Gamma_d(y)|$. These are estimated in stage 2 (3.2.4), simply by adding the neighbors of each node.

Finally, we have the matrix M with the intersections of the neighborhoods and the number of neighbors S_v for each node v. All that is left to do is calculate the jaccard coefficient for extended paths. Stage 4 (3.2.6) outputs pairs of nodes accompanied by their connect probability, which is expressed by the jaccard coefficient. Combining the equations (3.2) and (3.3) we get:

$$J(x, y) = \frac{|\Gamma_d(x) \cap \Gamma_d(y)|}{|\Gamma_d(x)| + |\Gamma_d(y)| - |\Gamma_d(x) \cap \Gamma_d(y)|} \tag{3.4}$$

This stage also reads the adjacency matrix A, which was constructed in stage 2 (3.2.4). Matrix A is read so as the algorithm does not output pairs of nodes that are already neighbors.

Our matrix-based implementation differs from the implementation of Bank et. al [46] and can easily be applied to both directed and undirected graphs.

In the remainder of this chapter, we discuss the data preparation procedure, the detailed description of a custom Writable variable that we used and the description of each one of the four stages of the algorithm along with their pseudocode. In every map/reduce stage described further down, it is given the input and output types of

the variables and also an example of the key/value pairs that are read and written. A key is underlined and a value that goes with this key is inside a parenthesis.

3.2.1 Data Preparation

As discussed in Section (3.1), the input of the program is an edge file representing a graph, where the node are positive integers. If the graph has N nodes, their identities need to have values that vary from 0 to $N-1$, in order to construct the adjacency matrix. Because this is not often the case, we have implemented a simple map/reduce algorithm that assigns new identities to the nodes and creates a new edge file.

At first, taking as input the edge file we create a node file that contains only the identities of the nodes of the graph. Setting the number of reducers to 1, the one reducer receives the node identities sorted. Afterwards, we read the sorted node file and assign new identities from 0 to $N-1$ to the nodes. Finally, we read the new node file and the original edge file and create a new edge file, suitable for our jaccard algorithm.

3.2.2 NodePair Writable

Our need to categorize paths based on their direction and depth, led us to implement a custom Writable variable, which we named "$NodePair$". This variable is created in order to help us emit composite values and is used to express paths. It contains three fields; $depth, nodeId$ *and* $direction$. $Depth$ shows the distance (in number of hops) between two nodes, $nodeId$ is the identity of a node and $direction$ shows if the edge is outgoing ($direction$=1) or incoming ($direction$=0). So, in order to emit an path "1 2" of depth d, where the identities of the nodes are 1 and 2 respectively, in the form of a key/value pair, the key would be 1 and the value would be of type $NodePair$ with fields: $depth = d,\ nodeId = 2,\ direction = 1$.

The class $NodePair$ implements the following methods:

- associated constructors. All Writable implementations must have a default constructor so that the MapReduce framework can instatiate them.
- setters and getters for the variables.
- $toString$: returns the String representation of the variable. It is called to write the reducer' s output to the HDFS filesystem, if the output format is TextOutputFormat.
- $readFields$: deserializes the bytes from the input stream by delegating to each object. It is called by a mapper to read from the HDFS.
- $write$: serializes each object in turn to the output stream. It is called by the collect() function and writes the variable to the HDFS, if the output format is SequenceFileOutputFormat.
- $compareTo$: defines the comparison convention for two objects and imposes the ordering. It is created because NodePair is an implementation of WritableComparable.
- $equals$: defines the equality convention for two objects.
- $hashCode$: returns the hash code of an object. It is used by the HashPartitioner (default partitioner of MapReduce) to choose a reduce partition.

3.2.3 Stage 1

Stage 1 reads from an input edge file node pairs at 1 hop and creates pairs at $2 \dots d_{max}$ hops. This stage is executed iteratively and every time it extends the depth by 1, so in order to compute pairs at d_{max} hops, the stage is called d_{max}-1 times.

The pseudocode of this stage is Algorithms 3.2.1 and 3.2.2.

Map
Input Types: LongWritable, Text
Output Types: IntWritable, NodePair

$$\begin{array}{rcll} x \quad y & \rightarrow & \underline{x}, & (1 \quad y \quad 1) \\ & & \underline{y}, & (1 \quad x \quad 0) \end{array} \tag{3.5}$$

$$x \quad d_{cur} \quad y \quad dir \quad \rightarrow \quad \underline{x}, \quad (d_{cur} \quad y \quad dir)$$

The pseudocode for this section is Algorithms 3.2.3 and 3.2.4 The input of this map method is the input file, which represents a graph $G(V, E)$ and is in the form of an edge file e.g. "x y", where each line of the file declares an edge from the first node to the second. For each input edge "x y", where x and y are positive integers, it outputs two pairs; the first is the outgoing edge from x, and the second is the incoming edge to y. As this stage is executed again and again the output of every iteration becomes the input of the next one. The output value is of type NodePair, as we want to discriminate pairs of nodes of different distances and directions.

Algorithm 3.2.1: MAP - STAGE 1($line$)

```
1 : global finalDepth, currentDepth
2 : local outputLey, outputValue
3 : local id1, id2, d, direction
4 :
5 : if currentDepth = 1 then
6 :     id1, id2 ← parseValue(value)
7 :     outputKey ← id1
8 :     setNodeId(outputValue, id2)
9 :     setDepth(outputValue, 1)
10 :    setDirection(outputValue, 1)
11 :    output (outputKey, outputValue)
12 :
13 :    outputKey ← id2
14 :    setNodeId(outputValue, id1)
15 :    setDirection(outputValue, 0)
16 :    output (outputKey, outputValue)
17 : else
18 :    id1, id2, d, direction ← parseValue(value)
19 :    outputKey ← id1
20 :    setNodeId(outputValue, id2)
21 :    setDepth(outputValue, d)
22 :    setDirection(outputValue, direction)
23 :    output (outputKey, outputValue)
```

Reduce
Input Types: IntWritable, NodePair
Output Types: IntWritable, NodePair

$$\begin{array}{llllll} \underline{x} & \{(1 & y & direction)\} & & \underline{x} & \{(1 & y & direction)\} \\ \dots & & & & \rightarrow & \dots & & & \\ \underline{x} & \{(d_{cur} & y & direction)\} & & \underline{x} & \{(d_{cur+1} & y & direction)\} \end{array} \tag{3.6}$$

This reduce method takes as input, pairs of depth $1 \dots d_{current}$, depending on the current iteration. Its job is to combine the pairs of depth 1 with the ones of depth $d_{current}$, in order to create pairs of depth $d_{current+1}$. It also removes duplicate pairs if any, and eliminates pairs of depth $d_{current}$ that are also present at smaller depths.

In order to do so, for each node-key, it keeps in separate lists, its neighbors at 1-hop (list1), $1 \dots d_{current-1}$ hops (list2) and $d_{current}$ hops (list3). The list of nodes at $d_{current}$ hops contains the ones that were produced in the last iteration (lines 7-28 in pseudocode). At first, the reducer compares list3 with list1 and list2, in order to remove duplicate elements from list3 that also exist in list1 or list2 (lines 30-35). Afterwards, it combines the list of 1-hop neighbors with the list of $d_{current}$-hop neighbors, to produce pairs that are at a distance of $d_{current+1}$-hops. New pairs come from incoming 1-hop nodes to outgoing $d_{current}$-hop nodes (lines 41-57). The output is pairs of nodes at $1 \dots d_{current+1}$ hops. If we go through the first iteration, then in order to produce pairs of distance 2, we combine the incoming 1-hop neighbors

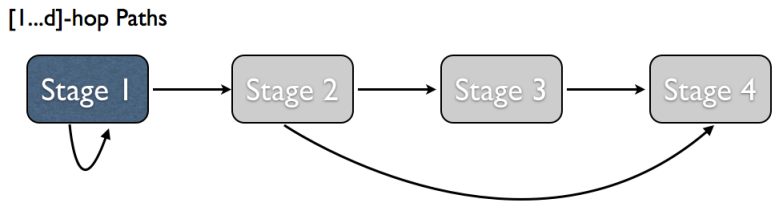

Figure 3.2: Extended Jaccard Coefficient Stage 1

with the outgoing 1-hop neighbors of a node. Again, the data type of the value is NodePair, as the pairs of nodes that are emitted differ in distance and direction.

Algorithm 3.2.2: REDUCE - STAGE 1($key, valueList$)

```
1 : global currentDepth
2 : local depth1InList, depth1OutList
3 : local depthMediumList, depthMediumList
4 : local depthMaxInList, depthMaxOutList
5 : local value, depth, direction, outputValue, node1, node2
6 :
7 : while hasNext(valueList)
8 :       value ← getNext(valueList)
9 :       direction ← getDirection(value)
10 :      depth ← getDepth(value)
11 :
12 :      if direction = in and depth = 1
13 :            insert(depth1InList, value)
14 :      else if direction = out and depth = 1
15 :            insert(depth1OutList, value)
16 :      else if direction = in and depth < currentDepth
17 :            insert(depthMediumInList, value)
18 :      else if direction = out and depth < currentDepth
19 :            insert(depthMediumOutList, value)
20 :      else if direction = in and depth = currentDepth
21 :            insert(depthMaxInList, value)
22 :      else if direction = out and depth = currentDepth
23 :            insert(depthMaxOutList, value)
24 :
25 :      if depth! = currentDepth
26 :            output (key, value)
27 :      end if
28 : end loop
29 :
30 : removeDuplicates(depthMaxInList, depthMediumInList)
31 : removeDuplicates(depthMaxOutList, depthMediumOutList)
32 : removeDuplicates(depthMaxInList, depth1InList)
33 : removeDuplicates(depthMaxOutList, depth1OutList)
34 : clear(depthMediumInList)
35 : clear(depthMediumOutList)
36 :
37 : for i ← 0 to size(depthMaxInList)
38 :      outputValue ← get(i, depthMaxInList)
39 :      output (key, outputValue)
40 :
41 : for each node2 ∈ depthMaxOutList
42 :      for each node1 ∈ depth1InList
43 :            if node1! = node2 then
44 :                  key ← getId(node1)
45 :                  setId(outputValue, getId(node2))
46 :                  setDirection(outputValue, out)
47 :                  setDepth(outputValue, currentDepth + 1)
48 :                  output (key, outputValue)
49 :
50 :                  key ← getId(node2)
51 :                  setId(outputValue, getId(node1))
52 :                  setDirection(outputValue, in)
53 :                  output (key, outputValue)
54 :            end if
55 :      end for
56 :      output (key, node2)
57 : end for
```

3.2.4 Stage 2

Stage 2 performs two operations. It calculates the size S_x of the neighborhood of each node x and constructs the two adjacency matrices A and A_d.

S_x will be used to find the size of union of the neighborhoods of each unconnected pair through the Equation (3.3) in stage 4 (3.2.6).

Matrix A represents the original graph G, whereas matrix A_d represents the graph G_d with extended neighborhoods at d_{max} hops. Matrix A is read in stage 4 (3.2.6) that calculates the jaccard coefficient. It is used to prevent the prediction of edges that already exist in G.

Matrix A_d is read in stage 3 (3.2.5) and is used to calculate the matrix M. Recall that the value of an element (i, j) of M is equal to the intersection of the neighborhoods of nodes i and j.

The pseudocode for this section is Algorithms 3.2.3 and 3.2.4

Map
Input Types: Text, Text
Output Types: IntWritable, NodePair

$$\begin{array}{llll} \underline{x} & \{(1 & y & direction)\} \\ \dots & & & \\ \underline{x} & \{(d_{max} & y & direction)\} \end{array} \rightarrow \begin{array}{llll} \underline{x} & \{(1 & y & direction)\} \\ \dots & & & \\ \underline{x} & \{(d_{max} & y & direction)\} \end{array} \tag{3.7}$$

This map method reads the output of stage 1. It reads pairs of nodes of depth $1 \dots d_{max}$ and emits them unchanged.

Algorithm 3.2.3: MAP - STAGE 2$(key, value)$

```
1 : local outputKey, outputValue
2 :
3 : outputKey ← key
4 : outputValue ← value
5 : output (outputKey, outputValue)
```

Reduce
Input Types: IntWritable, NodePair
Output Types: Text, BytesWritable

$$\begin{array}{llll} \underline{x} & \{(1 & y\ direction)\} \\ \dots & & & \\ \underline{x} & \{(d_{max} & y\ direction)\} \end{array} \rightarrow \begin{array}{llllll} \underline{x} & (0 & c_d(\vec{x})) & (directory\ intersection) \\ \underline{blockI\ x} & (1 & x & r(\vec{x})) & (directory\ union - \ row) \\ \underline{0\ 0} & (2 & x & S_x) & \\ \dots & & & & \\ \underline{nblocks\ 0} & (2 & x & S_x) & \end{array} \tag{3.8}$$

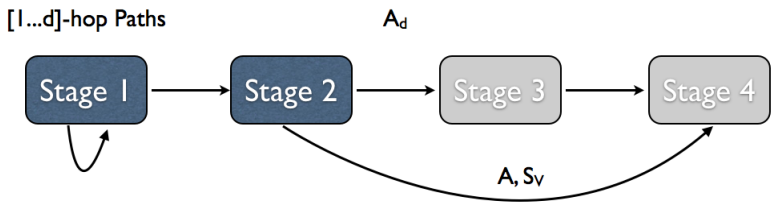

Figure 3.3: Extended Jaccard Coefficient Stage 2

This reduce method receives as input, pairs of nodes of depth $1 \dots d_{max}$. As mentioned earlier, we need the adjacency matrices A and A_d. Actually, we need the rows of A and the columns of A_d. Because of the fact that the graph is directed, a row i of the adjacency matrix contains the outgoing neighbors of node i and a column j contains the incoming neighbors of node j. Let the input key of a reducer be k. The outgoing 1-hop neighbors of k will form the row k of the adjacency matrix A (lines 14-16) and the incoming $1 \dots d_{max}$-hop neighbors of k will form the column k of the adjacency matrix A_d (lines 12-13).

Simply counting all the outgoing neighbors of a node k, we get the size of the neighborhood S_k (line 21).

Matrix A is divided into blocks of rows. The key of the output key/value pair representing a row $\vec{r}$ of the adjacency matrix A, is the block identity that this row belongs to and the value is the row (lines 33-35).

The neighborhood size of a node k, S_k must be available to all nodes, hence to all partitions. As a result, the key/value pair representing S_k is replicated to all partitions (lines 24-27). Because of the fact that the output key of the rows $\vec{r}$ of A is the block identity of the row, we realize that we will have as many reduce functions as the number of blocks. As a result the number of partitions that the S must be replicated to is equal to the number of blocks.

The rows $\vec{r}$ of A and the neighborhood sizes S are emitted and stored in the directory "union-rows", so that only stage 4 reads them. The columns $\vec{c_d}$ of A_d are emitted and stored in directory "intersections", so that only stage 3 reads it. The rows $\vec{r}$ and the columns $\vec{c_d}$ are constructed with the use of a byte array; its length is equal to $n/8$ bytes, as we need 1 bit for every node in the graph. A bit b in the byte array is 1, if and only if there is an edge between node-key and the node with identity b. In order to distinguish the three output types of key/value pairs (rows of A, columns of A_d and the neighborhood size S_k), we have made the convention that the first number of the output value serves as an identifier. If it is equal to 0, then the value is a column of A_d, if it is equal to 1 it is a row of A and if it is 2 the value is of type S_k.

Algorithm 3.2.4: REDUCE - STAGE 2($key, valueList$)

```
1 : global nReducers
2 : local rowBitmap, columnBitmap, neighborhoodBitmap, nblocks
3 : local value, nodeId, direction, depth, blocki
4 : local outputKey, outputValue, outputValue2
5 :
6 : while hasNext(valueList)
7 :       value ← getNext(valueList)
8 :       nodeId ← getId(value)
9 :       direction ← getDirection(value)
10 :       depth ← getDepth(value)
11 :
12 :      if direction = in then
13 :            setBit(columnBitmap, nodeId)
14 :      else if depth = 1 then
15 :            setBit(rowBitmap, nodeId)
16 :      else
17 :            setBit(neighborhoodBitmap, nodeId)
18 :      end if
19 : end loop
20 :
21 : neighborhoodSize ← countBits(rowBitmap)+
                         countBits(neighborhoodBitmap)
22 : outputValue ← concat(2, key, neighborhoodSize)
23 :
24 : for i ← 0 to nblocks
25 :      outputKey ← concat(i, 0)
26 :      output (outputKey, outputValue)
27 : end for
28 :
29 : outputKey ← key
30 : outputValue ← concat(0, columnBitmap)
31 : output (outputKey, outputValue)
32 :
33 : blocki ← nblocks/key
34 : outputKey ← concat(blocki, key)
35 : outputValue ← concat(1, key, rowBitmap)
36 : output (outputKey, outputValue)
```

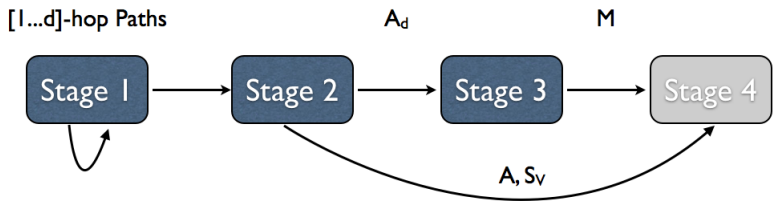

Figure 3.4: Extended Jaccard Coefficient Stage 3

3.2.5 Stage 3

In stage 3 we read the replicated matrices A_d that were produced in stage 2 (directory "intersections") and create the intersection matrix M. An element (i, j) of M comes from the product $(i, j) = \sum_{k=0}^{N}(i, k) \times (k, j)$, where (i, k) and (k, j) are elements of A_d and A_d^T respectively. This is explained with the following notion. (i, k) shows whether node k has i as an incoming neighbor, or not. Respectively, (k, j) shows whether k has j as an incoming neighbor, or not. If the above statements are both true, then i and j have one common neighbor, which is node k. Repeating, for all $k \in [0, N]$, we get the intersection of i and j.

Matrix A_d is divided in blocks of rows. Every reducer receives a block and calculates and emits the same block of rows of matrix M. In order to calculate a row of M, we need the whole matrix A_d^T, as a row of M comes from the multiplication of a row of A_d with all the columns of A_d^T. As a result, matrix A_d^T is replicated as many times as the number of blocks.

The pseudocode for this section is Algorithms 3.2.5 and 3.2.6

Map
Input Types: Text, Text
Output Types: Text, BytesWritable

$$\underline{k} \quad (0 \; \vec{c_d(k)}) \quad \rightarrow \quad \begin{array}{ll} \underline{0} & (0 \quad \vec{c_d(x)}) \\ \ldots & \\ \underline{nblocks} & (0 \quad \vec{c_d(x)}) \end{array} \tag{3.9}$$

The input of this map method is the directory "intersection" (output of stage 3). The input is key/value pairs that correspond to columns $\vec{c_d}$ of the matrix A_d. The output key is the identity of a block-i and the output value the column $\vec{c_d}$. Every column must be replicated to all blocks in order for the reducer to make the multiplication (lines 6-9).

Algorithm 3.2.5: MAP - STAGE 3$(key, value)$

```
1 : global nnodes, nreducers, nblocks
3 :
4 : outputValue ← value
5 :
6 : for blocki ← 0 to nblocks
7 :       outputKey ← blocki
8 :       output (outputKey, outputValue)
9 : end for
```

Reduce
Input Types: IntWritable, BytesWritable
Output Types: Text, BytesWritable

$$\begin{array}{ll} \underline{block_i} & (\vec{c_d}(0)) \\ \cdots & \\ \underline{block_i} & (\vec{c_d}(N)) \end{array} \quad \rightarrow \quad \begin{array}{ll} \underline{block_i\ start} & (0\ start\ r_M(\vec{start})) \\ \cdots & \\ \underline{block_i\ stop} & (0\ stop\ r_M(\vec{stop})) \end{array} \tag{3.10}$$

Each reducer of stage 3 is designed to create a block of rows of the matrix M. If we have n_B blocks, each reducer receives $split = N/n_B$ lines. If $block_i$ is the block identity, then the reducer will calculate the lines from $start = split * block_i$ until $stop = start + split$.

The key of each reducer is a block identity. The values that are fetched to the key correspond to multiple columns of A_d. The reducer needs to calculate only the rows $start \ldots stop$ of matrix M. Hence, for a column $\vec{c}$ that arrives, the reducer multiplies the bits $start \ldots stop$ of $\vec{c}$ with $\vec{c}$ and produces rows $start \ldots stop$ of M. As more columns come the new rows $start \ldots stop$ are added to the previous ones (lines 10-16).

The output is the calculated submatrix of M and is read by stage 4 (3.2.6). Matrix M is split in the blocks of rows, in the same way as matrix A_d. The output key is composite and contains two attributes. The first is the block identity of a row $r_{\vec{M}}$ and the second is the row identity. The output value is the row $r_{\vec{M}}$ (lines 18-23).

Algorithm 3.2.6: REDUCE - STAGE 3($key, valueList$)

```
1 : global nodeNumber, nreducers, nblocks
2 : local blocki, startRowId, endRowId
3 :
4 : matrixSplit ← nodeNumber/nblocks
5 : startRowId ← matrixSplit ∗ key
6 : endRowId ← startRowId + matrixSplit
7 :
8 : local column, intersectionSubmatrix
9 :
10 : while HASNEXT(valueList)
11 :    column ← GETNEXT(valueList)
12 :    for rowid ← startRowId to endRowId
13 :       intersectionMatrix[rowid] ← intersectionMatrix[rowid]+
14 :                                   MULTIPLY(rowid, column)
15 :    end for
16 : end loop
17 :
18 : block_i ← key
19 : for rowid ← startRowId to endRowId
20 :       outputKey ← CONCAT(block_i, rowId)
21 :       outputValue ← CONCAT(0, key, intersectionMatrix[rowid])
22 :       output (outputKey, outputValue)
23 : end for
```

An Alternative Strategy
It is obvious that the matrix multiplication performed is based on the creation of horizontal blocks. Another strategy tested, was to split the matrix in rectangular blocks. Each reducer would then compute a block of the final matrix. This technique allows a higher level of parallelization, but also increases network traffic. Suppose that we have blocks (bi, bk) of matrix A and blocks (bk, bj) of matrix B and we want to compute blocks (bi, bj) of matrix C. Each block (bi, bj) needs all blocks (bi, bk) and all (bk, bj). Suppose that the adjacency matrix has NB blocks and N rows. A block (bi, bk) of A needs to be copied to all reducers $R(ib, kb, jb)$. This is $NB * N$ key/value pairs. Similarly, a block (bk, bj) of B needs to be copied to all reducers $R(ib, kb, jb)$. This is another $NB * N$ key/value pairs. In total we have $2 * NB * N$ key/value pairs.

In the strategy followed, a block (bi, bk) goes to reducer $R(bi, bk)$. A block (bk, bj) is copied to all reducers $R(bi, bk)$, so $NB * N$ key/value pairs. It has turned out that a lower level of parallelization with less network transfers performs better.

3.2.6 Stage 4

Stage 4 produces the Jaccard coefficient for every unconnected pair of nodes by comparing their extended neighborhoods at d_{max} hops. The formula of the Jaccard coefficient is shown in (3.2). The size of the intersections of the neighborhoods at $1 \dots d$ hops of two nodes i and j is given by the element (i, j) of matrix M that was produced in the previous stage. The union of the neighborhoods of two nodes x and y at $1 \dots d$ hops is given by the formula:

$$\Gamma_d(x \cup y) = \Gamma_d(x) + \Gamma_d(y) - \Gamma_d(x \cap y) \tag{3.11}$$

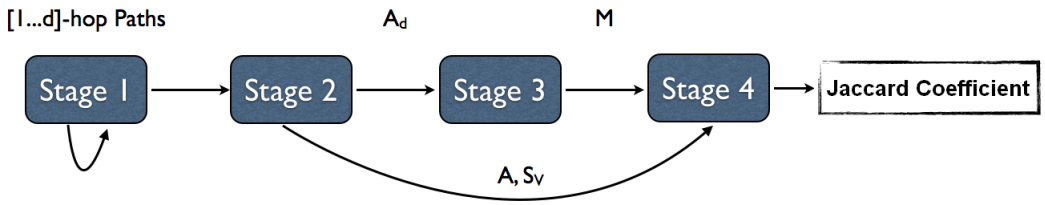

Figure 3.5: Extended Jaccard Coefficient Stage 4

Respectively the size of the unions of the neighborhoods is:

$$|\Gamma_d(x \cup y)| = |\Gamma_d(x)| + |\Gamma_d(y)| - |\Gamma_d(x \cap y)| \tag{3.12}$$

We have already calculated $|\Gamma_d(x)|$ (S_x) in stage 3, so we have all the necessary information available.

The pseudocode for this section is Algorithms 3.2.7, 3.2.8, 3.2.9, 3.2.10 and 3.2.11

Map
Input Types: Text, Text
Output Types: Text, Text

$$\begin{array}{llllll} \underline{blocki\ i} & (0 & i & \vec{r_M}) & \rightarrow & \underline{blocki\ i} \quad (0 \quad i \quad \vec{r_M}) \\ \underline{blocki\ i} & (1 & i & \vec{r}) & \rightarrow & \underline{blocki\ i} \quad (1 \quad i \quad \vec{r}) \\ \underline{blocki\ 0} & (2 & i & S_i) & \rightarrow & \underline{blocki\ 0} \quad (2 \quad i \quad S_i) \end{array} \tag{3.13}$$

This map method reads key/value pairs from two different directories. The first directory, "union-rows", was created in stage 2 and the second directory is the output of stage 3. "union-rows" contains the neighborhood sizes S and the rows $\vec{r}$ of the adjacency matrix A. The output of stage 3 contains the rows $\vec{r_M}$ of the matrix M. In this stage we have also created a custom partitioner, a key comparator and a group comparator. The data is read by the mappers and is passed into the partitioner, which in turn distributes it accordingly to the reducers.

The key of every input key/value pair is composite and contains two numbers separated by a blank character. The first is the block identity and the second is the identity of the node. The second argument is added in order for the key comparator to invoke the sorting of the rows. The output of this map method is the same as the input.

Algorithm 3.2.7: MAP - STAGE 4$(key, value)$

1 : $outputKey \leftarrow key$
2 : $outputValue \leftarrow value$
3 : **output** $(outputKey, outputValue)$

Partition

Our goal is to supply each reducer with a portion of rows of A, the same portion of rows of M and all the neighborhood sizes S. Because of the fact that the data do not come sorted by row/node identity into the reducer, obliges us to keep it in memory, which is impossible especially if we are dealing with extremely large datasets. For this reason, we implement a custom partitioner and a key comparator.

The partitioner reads the composite key of each pair and distributes the data to the reducers according to the first token of it (partition id). As already mentioned in the above map section, the partition identity corresponds to a block of matrix. A different way to partition the rows of the two matrices into the reducers could be with the use of a simple hash function of modulo. This was our initial approach, but was abandoned because it created many random I/O operations. This makes sense as every chunk of data, contains a sequential part of the matrix. A hash function of modulo means that a reducer needs to copy its input, row by row and usually from different mappers. As a result a large number of small reads is being performed and many random I/Os occur because it is highly likely that all the reducers need a part of the same mapper.

After the partitioning, a key comparator is executed. With the use of the comparator we impose the sorting of the data. In order to compute the Jaccard coefficient between a node with identity i and its neighbors, we need the row i of A, the row i of M and the S_j for the neighbors of i. Using the comparator, all this information come together, so each reducer does not need to store in memory all the rows for a block of the matrices A and M that receives. The comparator first compares the block identities, to ensure that each partition receives only the pairs that correspond to it. Afterwards, it does a secondary sort by the row identities, so that the elements will reach the reducer sorted. However, this secondary sort is not sufficient, because for every $r\vec{(}i)$ and $r_M\vec{(}i)$ that come, we need to have already received all the S_j, which does not happen for sure. For this reason, we created a "trick" for the partitioner. We have inserted number 0 to the composite key of the pairs of S_j, in the place of node identity (second argument of the composite key). In this way, the secondary sort will always bring first the S_j to the reducer. Hence, the calculations can be done on the fly, as when we have a $r\vec{(}i)$ and a $r_M\vec{(}i)$ at our disposal, the S_j have already arrived. To summarize, the partitioner is responsible for the correct distribution of the key/value pairs across the machines of our cluster.

Finally, a group comparator is executed. This comparator groups all the values for a key, so that the reducer receives all the values that share the same key together.

Algorithm 3.2.8: PARTITIONER - STAGE 4$(key, value)$

1 : **global** $partitionNumber$
2 : $blockId \leftarrow$ PARSEARGUMENTS(key)
3 : **return** $(blockId \ \% \ partitionNumber)$

Algorithm 3.2.9: KEY COMPARATOR - STAGE 4($var1, var2$)

```
1 : partitionId1, rowId1 ← parseArguments(var1)
2 : partitionId2, rowId2 ← parseArguments(var2)
3 :
4 : if partitionId1! = partitionId2 then
5 :       return (compare(partitionId1, partitionId2))
6 : end if
7 : return (compare(rowId1, rowId2))
```

Algorithm 3.2.10: GROUP COMPARATOR - STAGE 4($var1, var2$)

```
1 : partitionId1 ← parseArguments(var1)
2 : partitionId2 ← parseArguments(var2)
3 :
4 : return (compare(partitionId1, partitionId2))
```

Reduce
Input Types: Text, BytesWritable
Output Types: Text, FloatWritable

$$\begin{array}{lllll} \underline{blocki\ i} & (0 & i & r_M\vec{}(i)) & \\ \underline{blocki\ i} & (1 & i & r\vec{}(i)) & \rightarrow \quad J(i,j) = \frac{|\Gamma(i)\cap\Gamma(j)|}{|\Gamma(i)\cup\Gamma(j)|} \\ \underline{blocki\ 0} & \{(2 & j & S_j)\} & \end{array} \tag{3.14}$$

This reduce method receives key/value pairs that are: rows $r\vec{}(i)$ of the adjacency matrix A, rows $r_M\vec{}(i)$ of the matrix M and the sizes of the neighborhoods of all the nodes S_j. As mentioned in the above partition section, first arrive all the S_j and then come the rows sorted by their row identity. Again, this is very helpful as we only keep in memory one byte array for a row $\vec{r}$, one integer array for a row $\vec{r_M}$ and one integer array for the S_j. The first two arrays are overwritten when new $\vec{r}$ and $\vec{r_M}$ arrive.

The workflow of the reducer goes as follows. For every node i first arrive all the neighborhood sizes S and are stored (lines8-10). Then, come in turns a row $r\vec{}(i)$ (lines 14-17) and a row $r_M\vec{}(i)$ (lines 11-14). After the arrays that are used to store them are filled, we iterate through the elements j of $r_M\vec{}(i)$ and calculate the Jaccard coefficient for every pair (i, j) that are not present in $r\vec{}(i)$ (lines 19-31). Pairs with probability higher than a threshold θ are permitted to pass, meaning that they have a good chance of linking together in the future.

Algorithm 3.2.11: REDUCER - STAGE 5($key, valueList$)

```
1 : global reducersNumber, nodeNumber
2 : local rowBitmap, intersectionArray, unionArray
3 : local valueType, nodeId, isRow, isIntersection, jaccard
4 : local outputKey, outputValue
5 :
6 : while hasNext(valueList)
7 :       valueType ← analyzeValue(value)
8 :       if valueType ∈ S_i then
9 :           nodeId, neighborsNumber ← analyzeValue(value)
10 :          add(unionArray, nodeId, neighborsNumber)
11 :      else if valueType ∈ M⃗ then
12 :          nodeId, intersectionArray ← analyzeValue(value)
13 :          isIntersection ← true
14 :      else
15 :          rowBitmap ← value
16 :          isRow ← true
17 :      end if
18 :
19 :      if isIntersection = true and isRow = true
20 :          for id ← nodeId + 1 to nodeNumber
21 :              intersection ← getValue(intersectionArray, id)
22 :              union ← getValue(unionArray, id)+
                        getValue(unionArray, nodeId)
23 :              jaccard ← intersection/(union − intersection)
24 :              if isNeighbor(id, rowBitmap) = false then
25 :                  outputKey ← concatenate(nodeId, id)
26 :                  outputValue ← jaccard
27 :                  if jaccard > θ then
28 :                      output (outputKey, outputValue)
29 :                  end if
30 :              end if
31 :          end for
32 :          isIntersection ← false
33 :          isRow ← false
34 :          init(intersectionArray)
35 :      end if
36 : end loop
```

An Alternative Strategy

As mentioned above, the input key of the key/value pairs of the map method has been the block identity. However, this was not our initial approach, where the key was the node identity. Every reducer, would then receive a row $\vec{r}$ of A, a row $\vec{r_M}$ of M and all the neighborhood sizes S and the use of the custom partitioner would be unnecessary. Although this implementation is much simpler, the size of the intermediate data is enormous. The neighborhood sizes S, which are used for the calculation of the size of the union of the neighborhoods, must be replicated N times, if the input graph has N nodes. This means that the key/value pairs transferred are N^2, something that makes the algorithm very inefficient. In contradiction, in the best case that the number of blocks is equal to the number of reducers (n_R) available, the neighborhood

sizes S need to be replicated n_R, which is a great improvement in time and space complexity.

Chapter 4

Katz Algorithm

4.1 Introduction

Katz measure is a method that is based on the ensemble of all paths. It explores long paths for every node and relates the link occurence probability to the sum of total number of paths between two vertices weighed by the path length. This method directly sums over a collection of paths, exponentially dampened by length to count short paths more heavily. The Katz score is one of the measures that are based on the topological structure of the graph. Also, it is a global measure as it explore long paths.

Suppose that we have a graph $G(V, E)$, where V stands for the vertices and E for the edges of the graph. Then the katz score of a potential link between nodes x and y $\in G$ is defined as follows:

$$score(x, y) = \sum_{l=1}^{\infty} \beta^l \cdot |paths_{x,y}^{\langle l \rangle}| \tag{4.1}$$

$paths_{x,y}^{\langle l \rangle}$ is the set of all length-l paths from x to y and $\beta > 0$ is a user defined parameter.

Our algorithm is designed for directed graphs. This means that a path from x to y does not necessarily imply a path from y to x. However, it can also be applied to undirected graphs; each undirected edge from x to y is treated as two edges, one from x to y and one from y to x.

The program runs with 4 parameters; the input file, the path length L, β and the number of reducers. The input file must be in the form of an edge file e.g. $node1$ $node2$, where each line of the file declares an edge from the first node to the second. It is assumed that the identities of the nodes of the graph are positive integers. L tells the program to calculate paths of length-L for every node. If we want to examine paths between all nodes of the graph, a parameter L of about 6 should be sufficient according to the "Small World Effect". β is a number in the range (0, 1].

The algorithm is programmed in the form of chained map/reduce jobs. The fact that complicated calculations are needed in order to find and combine the length-l paths has imposed us this code style. As described in section 4.2 we have divided our program into two map/reduce stages.

The notation that is used in the following sections to express the key/value pairs of the map/reduce framework is: the keys are underlined and the values are inside a parenthesis e.g. <u>key</u>, (value). Additional key notation is summarized in Table 4.1

Symbol	Definition
G	Directed graph
V	Vertices
E	Edges
n	$\|V\|$ - number of nodes of the graph
m	$\|E\|$ - number of edges of the graph
L	maximum length of the paths that we calculate
$(x, y)_l$	path from node x to node y of length-l
l	length of a path
c	tha katz score of a path

Table 4.1: Definions of Symbols and Acronyms

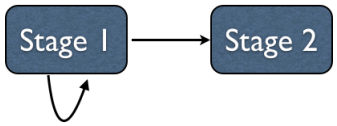

Figure 4.1: Katz Score Execution Flow

4.2 Description of the Algorithm

In this algorithm we want to find paths of length L and then apply the Katz formula (4.1) in order to find a score, which can be used as a predictor of future links.

A figure summarizing the execution flow of our algorithm is 4.3.

At First, for every pair of nodes x and y we need to calculate the number of paths of length $l = 0 \dots L$ that exist between them and then multiply each one with the factor β^l, so as to calculate every term of the sum. Finally, we add the terms and we get the Katz score for that pair.

The first stage (4.2.3) is executed iteratively and at each iteration the paths are extended by 1. Parallel paths of different lengths are grouped into one key/value pair in order to reduce the size of data. The key of every key/value pair is a source node and the value is composite. It contains the destination node of the path, the term $\Sigma\beta^l \cdot |paths|_l$ and the intermediate nodes of the path. The intermediate nodes are taken into account in order to avoid cycles.

After the first stage computes paths of length L, the second stage (4.2.4) is executed. The reducer of the second stage adds the different terms of the sum that contain the values that share the same key and produces the final score that corresponds to the probability of link occurence between the two nodes.

In the remainder of this chapter, we discuss the detailed description of two custom Writable variables that we used and the description of each one of the two stages of the algorithm along with their pseudocode. In every map/reduce stage described further down, it is given the input and output types of the variables and also an example of the key/value pairs that are read and written.

4.2.1 GraphPath1 Writable

Our need to categorize paths based on their length and direction and our need to count them, led us to implement a custom Writable variable, which we named "*GraphPath*". This variable is created in order to help us emit composite values and is used to express paths. It contains five fields; *nodeId*, *pathLength*, *count*, *direction*,

and *previousNodeIds*. *nodeId* is the identity of a destination node, *pathLength* is the length of the current path, *count* is the katz score for this path, *direction* shows if the path is outgoing (*direction*=1) or incoming (*direction*=0) considering as source the node-key and *previousNodeIds* holds the intermediate nodes of the path. *previousNodeIds* is used to avoid cycles. Suppose that we want to emit a path "1 2" of length 3, with intermediate nodes the 3 and 4, in the form of a key/value pair, with the use of GraphPath variable as value. Then, the key would be 1 and the fields of the value would be: $nodeId = 2$, $length = 3$, $count = \beta^3 \cdot 1$, $direction = 1$ *and* $previousNodeIds = (3\ 4)$

The class *GraphPath*1, implements the following methods:

- associated constructors. All Writable implementations must have a default constructor so that the MapReduce framework can instatiate them.
- setters and getters for the variables.
- *addPreviousNodeId*: adds a node identity to the list of the intermediate nodes of a path. This method is called at the construction and expansion of a path.
- *addPreviousNodeIds*: adds to the variable new intermediate nodes of a new path. It is called when two paths are concatenated to one.
- *toString*: returns the String representation of the variable. It is called to write the reducer' s output to the HDFS filesystem, if the output format is TextOutputFormat.
- *readFields*: deserializes the bytes from the input stream by delegating to each object. It is called by a mapper to read from the HDFS.
- *write*: serializes each object in turn to the output stream. It is called by the collect() function and writes the variable to the HDFS, if the output format is SequenceFileOutputFormat.
- *compareTo*: defines the comparison convention for two objects and imposes the ordering. It is created because GraphPath is an implementation of WritableComparable.
- *equals*: defines the equality convention for two objects.
- *hashCode*: returns the hash code of an object. It is used by the HashPartitioner (default partitioner of MapReduce) to choose a reduce partition.

4.2.2 GraphPath2 Writable

This variable serves the same purpose as the the variable *GraphPath*1. The difference is that we use this writable at stage 2, where we need less information for every path. So, this variable is identical to the one in 4.2.1, but has fewer fields. *GraphPath*2 contains two fields; *pathLength* and *count*. As above, *pathLength* is the length of the path and *count* is the Katz score for this path. Often, GraphPath2 carries paths of several lengths. In this occasion *pathLenth* is useless. This attribute helps us spot the original edges of the graph (*pathLength*=1). Hence, at stage 2 we can reject

paths that are already connected edges. The paths that reach stage 2 always have an outgoing direction, so we also do not need field *direction*. Finally, the intermediate nodes of the path serve no purpose as the cycles have been detected and avoided, so also the field *previousNodeIds* is useless. The class *GraphPath*2, implements the following methods:

- associated constructors. All Writable implementations must have a default constructor so that the MapReduce framework can instatiate them.
- setters and getters for the variables.
- *toString*: returns the String representation of the variable. It is called to write the reducer' s output to the HDFS filesystem, if the output format is TextOutputFormat.
- *readFields*: deserializes the bytes from the input stream by delegating to each object. It is called by a mapper to read from the HDFS.
- *write*: serializes each object in turn to the output stream. It is called by the collect() function and writes the variable to the HDFS, if the output format is SequenceFileOutputFormat.
- *compareTo*: defines the comparison convention for two objects and imposes the ordering. It is created because GraphPath is an implementation of WritableComparable.
- *equals*: defines the equality convention for two objects.
- *hashCode*: returns the hash code of an object. It is used by the HashPartitioner (default partitioner of MapReduce) to choose a reduce partition.

4.2.3 Stage 1

Stage 1 of the algorithm is executed iteratively; it reads various paths between all nodes and expands them by 1 if needed. During the first iteration, it reads the input file that represents a graph $G(V, E)$ and produces paths of length 2. If the input argument L is greater that 2, then it is executed again and again, as many times as L. Obviously the output of each iteration, becomes the input for the next.

Map
Input Types: LongWritable, Text
Output Types: IntWritable, GraphPath1

$$\begin{array}{lcl} x \quad y & \rightarrow & \underline{x}, \quad (y \quad 1 \quad 0 \quad 1 \quad ()) \\ & & \underline{y}, \quad (x \quad 1 \quad 0 \quad 0 \quad ()) \end{array} \tag{4.2}$$

$$x \quad y\ l\ c\ dir\ (prevIds) \quad \rightarrow \quad \underline{x}, \quad (y \quad l \quad c \quad dir \quad (prevIds))$$

The input of this map method is a file that contains the original edges of the graph and the generated paths, if any (from a previous iteration). In the first case, it reads data of type "$\underline{nodeId1}$ $(nodeId2)$", while in the second case, data of type

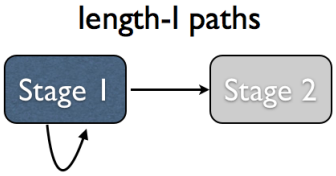

Figure 4.2: Katz Score Stage 1

"$\underline{nodeId1}$ ($nodeId2$ $pathLength$ $count$ $direction$ ($previousNodeIds$))". In the first case, it constructs two objects of data type $GraphPath1$ for every edge; one to declare the outgoing edge from $nodeId1$ ($direction$=1) and one to declare the incoming edge to $nodeId2$ ($direction$=0) (lines 4-17). In the second case, it constructs one object for every path and emits it unchanged (lines 18-27).

Recall that the data type GraphPath1 is of the form
$nodeId$ $length$ $count$ $direction$ ($previousNodeIds$).

Algorithm 4.2.1: MAPPER - STAGE 1($key, value$)

```
1 : global length
2 : local outputKey, outputValue
3 :
4 : if length = 2 then
5 :       id1, id2 ← PARSEVALUE(value)
6 :       outputKey ← id1
7 :       SETNODEID(outputValue, id2)
8 :       SETLENGTH(outputValue, 1)
9 :       SETCOUNT(outputValue, 0)
10 :      SETDIRECTION(outputValue, 1)
11 :      SETPREVIOUSIDS(outputValue, "()")
12 :      output (outputKey, outputValue)
13 :
14 :      outputKey ← id2
15 :      SETNODEID(outputValue, id1)
16 :      SETDIRECTION(outputValue, 0)
17 :      output (outputKey, outputValue)
18 : else
19 :      id1, id2, l, count, direction, previousIds ← PARSEVALUE(value)
20 :      outputKey ← id1
21 :      SETNODEID(outputValue, id2)
22 :      SETLENGTH(outputValue, l)
23 :      SETCOUNT(outputValue, count)
24 :      SETDIRECTION(outputValue, direction)
25 :      SETPREVIOUSIDS(outputValue, previousIds)
26 :      output (outputKey, outputValue)
27 : end if
```

Reduce
Input Types: $\underline{\text{IntWritable}}$, GraphPath1
Output Types: $\underline{\text{IntWritable}}$, GraphPath1

$$
\begin{array}{llllllllll}
x & y_i & 0 & c\ dir\ (prevIds) & \rightarrow & \underline{x}, & (y_i & 0 & c\ dir & ()) \\
\ldots & & & & & & & & & \\
x & y_i & l & c\ dir\ (prevIds) & \rightarrow & \underline{x}, & (y_i & l & c\ dir & ()) \\
\ldots & & & & & & & & & \\
x & y_i & L_{cur} & c\ dir\ (prevIds) & \rightarrow & \underline{x}, & (y_i & L_{cur}+1 & c\ dir & (prevIds))
\end{array}
\tag{4.3}
$$

This reduce method takes as input the output of the above map method (4.2). Let L be the maximum length of paths that we want to calculate and $L_{current}$ be the length of paths that we have calculated. Then each reducer receives all paths with length $0 \ldots L_{current}$ for a node. It maintains 4 lists; the first ("originalInPairs") stores the original incoming edges (length=1), the second ("originalOutPairs") the original outgoing edges (length=1), the third ("producedPairs") the paths that were produced in the last iteration (length=$L_{current}$-1) and the fourth ("intermediatePairs") the paths that have been produced in previous iterations (length< $L_{current}$) (lines 6-20).

Each record of the list "intermediatePairs", contains paths of various lengths. Their Katz measure is calculated and the field with the previous node identities is deleted as it is no more useful. The paths are then emitted (lines 22-27).

Afterwards, we combine the incoming length 1 edges with the paths of length $L_{current}$ to create new paths of length $L_{current} + 1$. For every node-key, we create a path from an incoming to an outgoing neighbor. For example, suppose that the key is node x and x has one incoming neighbor, which is node-y. Also, x has one path at z and one at w. Then, after the execution of the reducer, there will have been produced two new paths, which will be $y \rightarrow z$ and $y \rightarrow w$. Also, for every new path $y \rightarrow z$ we check to see if we have created a cycle, by comparing the route of the path $x \rightarrow z$, so that it does not contain node y. If a cycle is detected, then the path is rejected (lines 29-50).

All the paths of length $0 \ldots L_{current}+1$ are emitted.

Recall that the data type GraphPath1 is of the form
$nodeId \quad length \quad count \quad direction \quad (previousNodeIds)$.

Algorithm 4.2.2: REDUCER - STAGE 1($key, valueList$)

```
1 : global L, L_current
2 : local originalInPairs, originalOutPairs
3 : local intermediatePairs, producedPairs
4 : local outputKey, outputValue
5 :
6 : while hasNext(valueList)
7 :       value ← getNext(valueList)
8 :       if getLength(value) = 1 then
9 :           if getDirection(value) = 0 then
10 :              addList(originalInPairs, value)
11 :          else
12 :              addList(originalOutPairs, value)
13 :          end if
14 :      else  if getLength(value) = L_current then
15 :          addList(producedPairs, value)
16 :      else  if getLength(value) < L_current
17 :           and  not contains(originalOutPairs, value) then
18 :          addList(intermediatePairs, value)
19 :      end if
20 : end loop
21 :
22 : while hasNext(intermediatePairs)
23 :      outputKey ← key
24 :      outputValue ← getNext(intermediatePairs)
25 :      setPrevIds(outputValue, null)
26 :      output (outputKey, outputValue)
27 : end loop
28 :
29 : if L_current = 2 then
30 :      producedPairs ← originalOutPairs
31 : end if
32 :
33 : for node2 ∈ producedPairs do
34 :      for node1 ∈ originalInPairs do
35 :          if node1! = node2 then
36 :              outputKey ← node1
37 :              setId(outputValue, node2)
38 :              setDirection(outputValue, 1)
39 :              setLength(outputValue, L_current + 1)
40 :              setPrevIds(getPrevIds(node2))
41 :              addCheckPrevIds(outputValue, node1)
42 :              output (outputKey, outputValue)
43 :          end if
44 :      end for
45 :      if  not contains(originalOutPairs, node2) then
46 :          outputKey ← key
47 :          outputValue ← node2
48 :          output (outputKey, outputValue)
49 :      end if
50 : end for
```

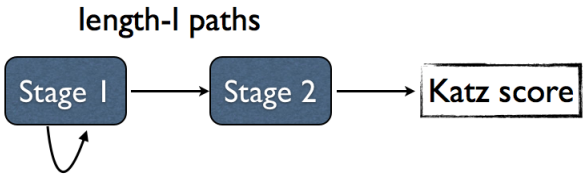

Figure 4.3: Katz Score Stage 2

4.2.4 Stage 2

After the successful extension of paths at the desired length by stage 1, stage 2 is called. Here, we assemble all the paths and calculate the final katz score.

Map

Input Types: Text, Text
Output Types: Text, GraphPath2

$$\begin{array}{llllll} x & y_i & 0 & c\ dir\ (prevIds) & \rightarrow & \underline{x\ y_i},\quad (0 \quad c) \\ \multicolumn{6}{c}{\ldots} \\ x & y_i & l & c\ dir\ (prevIds) & \rightarrow & \underline{x\ y_i},\quad (l \quad c) \\ \multicolumn{6}{c}{\ldots} \\ x & y_i & L & c\ dir\ (prevIds) & \rightarrow & \underline{x\ y_i},\quad (L \quad c) \end{array} \tag{4.4}$$

This map method reads paths of various lengths. The key is the identity of the node-source of the path and the value is the identity of the node-destination of the path. The value also contains information about the path such as its length, direction, Katz score and the intermediate nodes of the path. Since, the paths have been correctly calculated in stage 1 and all the paths are outgoing, we do not need the fields "*direction*" and "*previousNodeIds*". So, for every key/value pair that we receive as input, we modify the key and value as follows. The key becomes composite and contains both identities of the path (source and destination) and the value contains the path length and the Katz score. The attribute path length is valid only for length 1 paths, because one pair of nodes may represent paths of different lengths. The writable variable $GraphPath2$ is used as value in the key/valye pairs, as also described in 4.2.2.

Algorithm 4.2.3: MAPPER - STAGE 2$(key, value)$

```
1 : local outputKey, outputValue
2 :
3 : id1 ← key
4 : id2, length, count ← PARSEVALUE(value)
5 :
6 : outputKey ← CONCATENATE(id1, id2)
7 : SETLENGTH(outputValue, length)
8 : SETCOUNT(outputValue, count)
9 :
10 : output (outputKey, outputValue)
```

Reduce

Input Types: Text, GraphPath2

Output Types: Text, FloatWritable

$$\begin{array}{l} \underline{x\ y},\quad (l \quad c_0) \\ \ldots \\ \underline{x\ y},\quad (l \quad c_I) \end{array} \quad \rightarrow \quad score(x,\ y) = \sum_0^I c_i \tag{4.5}$$

This reduce method receives key/value pairs, of which the key contains the end nodes of a path and the values are paths of various lengths. Each value has its attribute *count* pointing to the Katz score. The reducer just adds the attributes *count* of all the values that receives, resulting to the final Katz score of the path (lines 3-9). It outputs the source and destination node identity, accompanied with the final Katz score, if the score is higher than a threshold θ.

Algorithm 4.2.4: REDUCER - STAGE 2($key, valueList$)

```
1 : local score, outputKey, outputValue
2 :
3 : while hasNext(valueList)
4 :       value ← getNext(valueList)
5 :       if getPathLength(value) = 1 then
6 :           return
7 :       end  if
8 :       score ← score + getCount(value)
9 : end  loop
10 :
11 : if score > θ then
12 :       outputKey ← key
13 :       outputValue ← score
14 :       output (outputKey, outputValue)
15 : end if
```

Chapter 5

Experiments and Results

5.1 Data Sets and Experimental Methodology

We run our experiments in our compute cluster, which is composed of 13 nodes. What we want to measure, is the performance of our algorithms and the accuracy of our predictions. The performance depends on many parameters, like the size of the input graph, the length of the paths that we examine and the number of the compute nodes running concurrently. Changing either of these, results in different execution times and different output sizes.

We used as input five graphs of different sizes. These are described in Table (5.1). There is a wide collection of graphs in [49] and [50]. "Wiki-Vote" was published by Leskovec et. al in [51, 52], "Cond-Mat" by Leskovec et. al in [53], "Cit-HepPh" by Leskovec et. al in [**?**] and Hehrke et. al in [**?**], "Trust Epinions" by Massa et. al in [56] and "Gnutella" by Leskovec et. al in [57] and Ripeanu et. al in [58].

Name	**Type**	**Nodes**	**Edges**	**Effective Diameter**	**Network Description**
Wiki-Vote	Directed	7115	103629	3.8	vote
CondMat	Undirected	23133	186936	6.6	co-authorship
HepPh	Directed	34596	421578	5	citation
TrustEpinions	Directed	49288	487183	5	social
Gnutella	Directed	62586	147892	6.7	peer to peer

Table 5.1: Networks used for experiments

In this section, we investigate how bigger graphs and larger paths affect the performance, and the scale-up that we achieve by adding more nodes in our cluster. As we can see from the Table 5.1, the effective diameter for the graphs that we examine varies from 3.8 to 6.7, which is explained by the "Small World Effect". In our experiments, we chose the average length of paths to be 3, in order to examine a sufficient amount of the graph but not the whole. Also, in experiments where the parameter that changes is the path length, we have set an upper limit of 4, so as to avoid examining all the possible paths between nodes.

Furthermore, we will try to measure the precision and recall of our two algorithms in order to evaluate the Jaccard and Katz methods of prediction. Given two timestamps of the co-authorship network condMat at year 2003 and 2005, we will try to predict the new edges of the graph at 2005, by using the graph of 2003 as input to our program.

The term precision in statistics is a measure of fidelity, while the term recall is a measure of completeness. In link prediction, precision is defined as the number of correctly predicted edges divided by the total number of predicted edges.

$$Precision = \frac{number\ of\ correctly\ predicted\ edges}{number\ of\ predicted\ edges} \tag{5.1}$$

Recall is defined as the number of correctly predicted edges divided by the actual number of new edges added in the graph.

$$Precision = \frac{number\ of\ correctly\ predicted\ edges}{number\ of\ new\ edges} \tag{5.2}$$

The structure of the rest of this section goes as follows. In subsection 5.2 and 5.3 lie the experiments for the Extended Jaccard Coefficient and the Katz Score algorithms that we conducted using the graphs of Table 5.1. In subsection 5.4 we try to explain the experiments and figure out the patterns behind their behavior.

5.2 Extended Jaccard Algorithm's Experiments

At first, we investigate how the size of the input graph affects the execution time. Running in our hadoop cluster of 13 nodes and examining neighborhoods of depth 3 we get the following results.

Stage	**Running Time (sec)**	**Output Data (MB)**
Stage 1 (iteration 1)	33	126.5
Stage 1 (iteration 2)	221	1533
Stage 2	167	21.4
Stage 3	100	202.8
Stage 4	70	162.5
Total	591	2046.2

Table 5.2: Jaccard Coefficient, Wiki-Vote, depth=3

Stage	**Running Time (sec)**	**Output Data (MB)**
Stage 1 (iteration 1)	42	131.7
Stage 1 (iteration 2)	259	1677
Stage 2	234	162.6
Stage 3	205	2141
Stage 4	93	85.6
Total	833	4197.9

Table 5.3: Jaccard Coefficient, Cond-Mat, depth=3

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (iteration 1)	98	263
Stage 1 (iteration 2)	356	1770
Stage 2	191	341
Stage 3	348	4775
Stage 4	76	97.4
Total	1069	7246.4

Table 5.4: Jaccard Coefficient, HepPh, depth=3

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (iteration 1)	40	1155
Stage 1 (iteration 2)	1157	31472
Stage 2	2740	668.7
Stage 3	3440	9700
Stage 4	220	32.7
Total	7597	43028.4

Table 5.5: Jaccard Coefficient, Trust Graph, depth=3

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (iteration 1)	32	21.2
Stage 1 (iteration 2)	41	81.9
Stage 2	840	1000
Stage 3	1080	15670
Stage 4	2028	0.15
Total	4021	16773.25

Table 5.6: Jaccard Coefficient, Gnutella Graph, depth=3

Figure 5.1 summarizes the results.

Let us choose graph HepPh and run our algorithm, extending each time the neighborhoods of the nodes at different depths. The results are:

Stage	Running Time (sec)	Output Data (MB)
Stage 1	35	16
Stage 2	37	341.5
Stage 3	100	4775
Stage 4	61	0.8
Total	233	5133.3

Table 5.7: Jaccard Coefficient, HepPh, depth=1

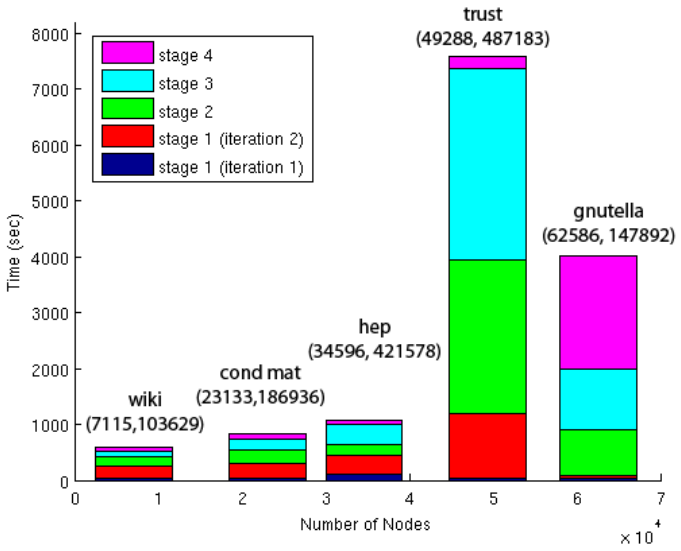

Figure 5.1: Graph Size vs Time (Jaccard Coefficient)

Stage	**Running Time (sec)**	**Output Data (MB)**
Stage 1	37	263
Stage 2	42	341.5
Stage 3	123	4775
Stage 4	71	17.2
Total	273	5396.7

Table 5.8: Jaccard Coefficient, HepPh, depth=2

Stage	**Running Time (sec)**	**Output Data (MB)**
Stage 1 (iteration 1)	98	263
Stage 1 (iteration 2)	356	1770
Stage 2	191	341
Stage 3	348	4775
Stage 4	76	97.4
Total	1069	7246.4

Table 5.9: Jaccard Coefficient, HepPh, depth=3

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (iteration 1)	38	263
Stage 1 (iteration 2)	853	1770
Stage 1 (iteration 3)	1980	6319
Stage 2	949	341.5
Stage 3	763	4775
Stage 4	212	254.4
Total	4795	13722.9

Table 5.10: Jaccard Coefficient, HepPh, depth=4

Figure 5.2 summarizes the results.

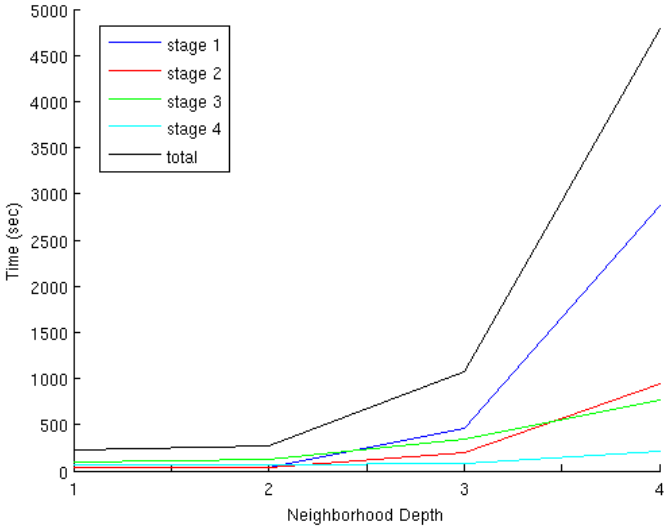

Figure 5.2: Neighborhood Depth vs Time (Jaccard Coefficient)

Next we want to measure the rank of parallelism that we have achieved. We run our algorithm with input the graph HepPh, extending its neighborhoods at depth 3 with various number of compute nodes, starting from 1 and reaching 5. The running times were:

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (Iteration 1)	271	263
Stage 1 (Iteration 2)	2817	1770
Stage 2	353	308.3
Stage 3	1455	4775
Stage 4	4832	97.4
Total	9728	6876.7

Table 5.11: Jaccard Coefficient, HepPh, depth=3, Nodes=1

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (Iteration 1)	105	263
Stage 1 (Iteration 2)	2017	1770
Stage 2	258	308.3
Stage 3	1129	4775
Stage 4	1378	97.4
Total	4887	6876.7

Table 5.12: Jaccard Coefficient, HepPh, depth=3, Nodes=2

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (iteration 1)	80	263
Stage 1 (iteration 2)	1125	1770
Stage 2	151	308.3
Stage 3	1093	4775
Stage 4	787	97.4
Total	3236	6876.7

Table 5.13: Jaccard Coefficient, HepPh, depth=3, Nodes=3

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (iteration 1)	65	263
Stage 1 (iteration 2)	729	1770
Stage 2	168	311
Stage 3	925	4775
Stage 4	495	97.4
Total	2382	6876.7

Table 5.14: Jaccard Coefficient, HepPh, depth=3, Nodes=4

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (iteration 1)	42	263
Stage 1 (iteration 2)	430	1770
Stage 2	123	311
Stage 3	598	4775
Stage 4	193	97.4
Total	1386	6876.7

Table 5.15: Jaccard Coefficient, HepPh, depth=3, Nodes=5

Figure 5.3 summarizes the results.

Finally, we evaluated our predictor by calculating the presicion and recall that is achieved. The input graph is the 2003 timestamp of the collaboration matrix CondMat. Using this, we will try to predict the 2005 timestamp of CondMat. The graph of 2003 contains 31163 nodes and 120029 edges. The graph of 2005 contains 40421 nodes and 175693 edges. The jaccard coefficient does not predict the appearance of new nodes, just the appearance of new edges among the existing nodes. The algorithm was run with different depth as input. The matrix summarizing the results is:

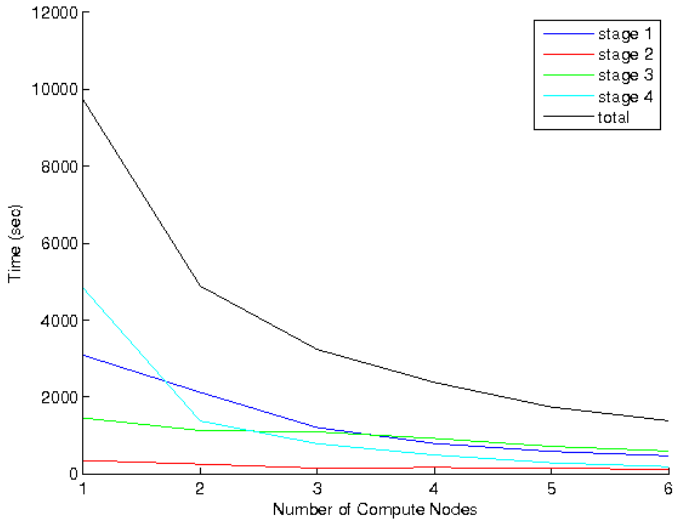

Figure 5.3: Number of Compute Nodes vs Time (Jaccard Coefficient)

Depth	**Precision**	**Recall (MB)**
depth=1	0.008	0.003
depth=2	0.007	0.006
depth=3	0.006	0.008
depth=4	0.004	0.011

Table 5.16: Jaccard Coefficient, Prediction Evaluation

The Figure of the precision/recall is 5.4.

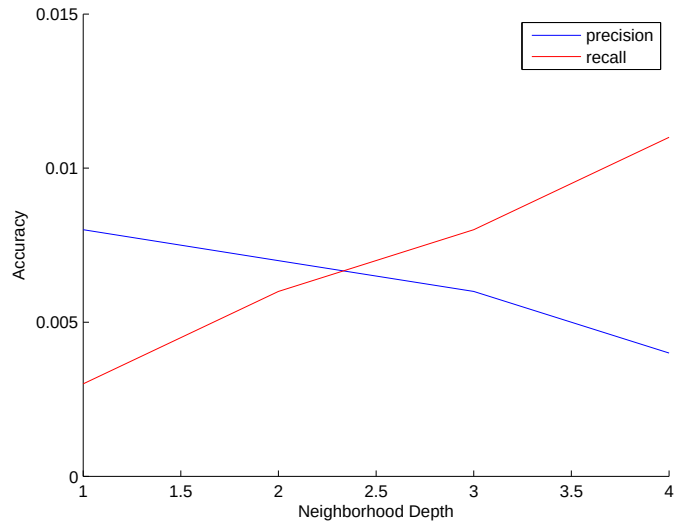

Figure 5.4: Depth vs Precision/Recall (Jaccard Coefficient)

5.3 Katz Algorithm's Experiments

There follow the same experiments for the katz score as the ones made in subsection 5.2 for the jaccard coefficient. Again the parameters that affect the performance is the size of the graph, the length of the paths that we examine and the number of compute nodes available in the cluster.

The following matrices show how the size of the input graph affects the execution time. These experiments were executed in 13 compute nodes.

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (Iteration 1)	26	160
Stage 1 (Iteration 2)	110	4146
Stage 2	180	191.4
Total	316	4497.4

Table 5.17: Katz Score, Wiki-Vote, length=3

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (Iteration 1)	36	158
Stage 1 (Iteration 2)	90	2981
Stage 2	70	0.1
Total	196	3139.1

Table 5.18: Katz Score, cond-Mat, length=3

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (Iteration 1)	73	253.2
Stage 1 (Iteration 2)	726	3111
Stage 2	183	63.5
Total	982	3427.7

Table 5.19: Katz Score, HepPh, length=3

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (Iteration 1)	54	1159
Stage 1 (Iteration 2)	3724	78669
Stage 2	8346	960
Total	12124	80788

Table 5.20: Katz Score, Trust Network, length=3

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (Iteration 1)	51	22.9
Stage 1 (Iteration 2)	98	94.4
Stage 2	107	0.1
Total	256	117.4

Table 5.21: Katz Score, Gnutella, length=3

Figure 5.5 summarizes the results.

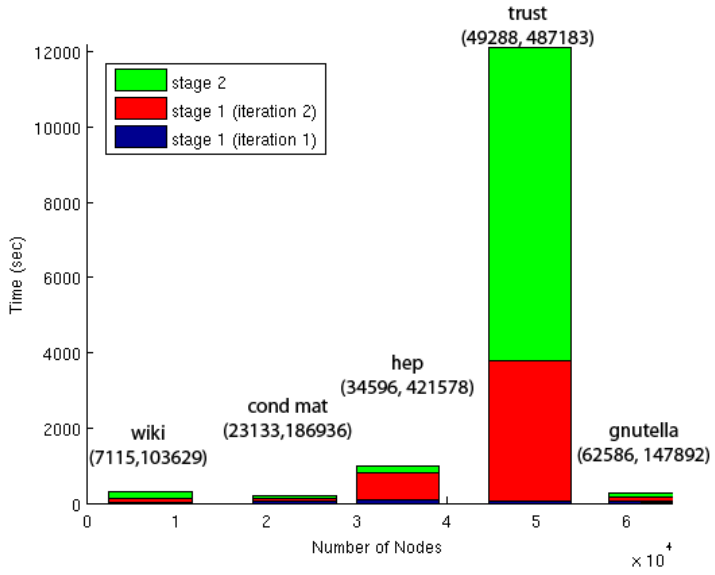

Figure 5.5: Graph Size vs Time (Katz Score)

The following matrices show how the length of the paths examined affect the execution time and the output size of the algorithm. These experiments were executed in 13 compute nodes.

Stage	**Running Time (sec)**	**Output Data (MB)**
Stage 1	51	72.9
Stage 2	35	4.5
Total	86	77.4

Table 5.22: Katz Score, HepPh, length=2

Stage	**Running Time (sec)**	**Output Data (MB)**
Stage 1 (Iteration 1)	55	253.2
Stage 1 (Iteration 2)	726	3111
Stage 2	183	63.5
Total	982	3427.7

Table 5.23: Katz Score, HepPh, length=3

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (iteration 1)	34	253.2
Stage 1 (iteration 2)	708	3123
Stage 1 (iteration 3)	6600	37915
Stage 2	6025	305.2
Total	13367	41596.4

Table 5.24: Katz Score, HepPh, length=4

Figure 5.6 summarizes the results.

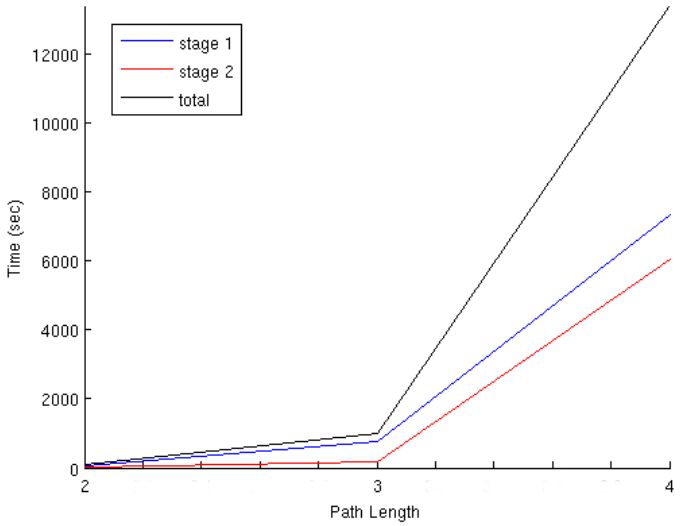

Figure 5.6: Path Length vs Time (Katz Score)

Next, we will measure the scale up that we have achieved. We run our algorithm with input the graph HepPh, examining paths of length=3 with a changing number of compute nodes, starting from 1 and reaching 5. The running times are:

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (Iteration 1)	72	253.2
Stage 1 (Iteration 2)	504	3111
Stage 2	4011	88.7
Total	4587	3452.9

Table 5.25: Katz Score, HepPh, length=3, Nodes=1

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (Iteration 1)	50	253.2
Stage 1 (Iteration 2)	155	3111
Stage 2	1719	88.7
Total	1924	3452.9

Table 5.26: Katz Score, HepPh, length=3, Nodes=2

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (iteration 1)	39	253.2
Stage 1 (iteration 2)	116	3111
Stage 2	1128	88.7
Total	1283	3452.9

Table 5.27: Katz Score, HepPh, length=3, Nodes=3

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (iteration 1)	34	253.2
Stage 1 (iteration 2)	87	3111
Stage 2	740	88.7
Total	861	3452.9

Table 5.28: Katz Score, HepPh, length=3, Nodes=4

Stage	Running Time (sec)	Output Data (MB)
Stage 1 (iteration 1)	27	253.2
Stage 1 (iteration 2)	79	3111
Stage 2	530	88.7
Total	636	3452.9

Table 5.29: Katz Score, HepPh, length=3, Nodes=5

Figure 5.7 summarizes the results

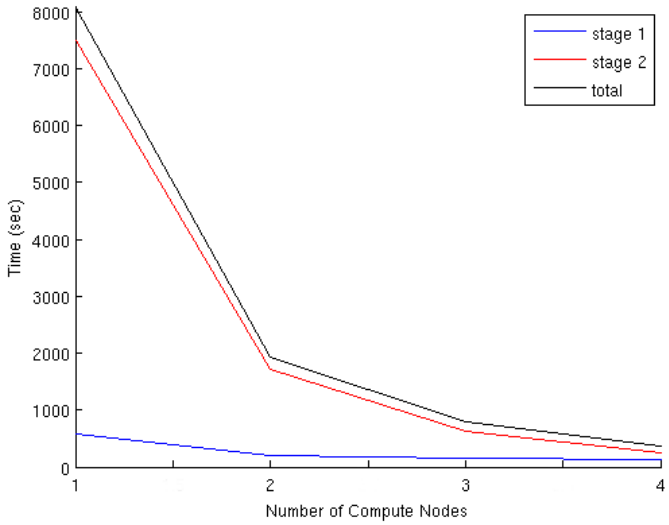

Figure 5.7: Number of Compute Nodes vs Time (Katz Score)

Finally, we will calculate the presicion and recall that is achieved by running our algorithm on an actual graph. The input graph is the 2003 timestamp of the collaboration matrix CondMat. Using this, we will try to predict the 2005 timestamp of CondMat. The graph of 2003 contains 31163 nodes and 120029 edges. The graph of 2005 contains 40421 nodes and 175693 edges. The Katz score doesn' t predict the appearance of new nodes, just the appearance of new edges among the existing nodes. The algorithm was run 3 times; at each time it examined paths of different length, so as to see how the length of the paths affect the quality of our predictions. There follow two matrices summarizing the results, one with $\beta = 0,3$ and one with $\beta = 0,5$.

Length	**Precision**	**Recall (MB)**
length=2	0.038	0.003
length=3	0.025	0.006
length=4	0.014	0.012

Table 5.30: Katz Score, Prediction Evaluation, β=0.3

Length	**Precision**	**Recall (MB)**
length=2	0.029	0.008
length=3	0.022	0.018
length=4	0.016	0.026

Table 5.31: Katz Score, Prediction Evaluation, β=0.5

The figure of precision/recall for the Katz measure is 5.8.

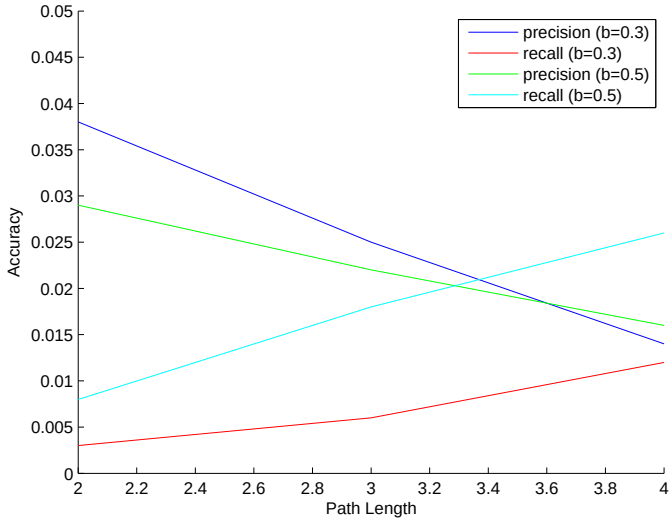

Figure 5.8: Path Length vs Precision/Recall (Katz Score)

5.4 Discussion of Results

Some very interesting conclusions can be deducted from the experiments in subsections 5.2 and 5.3.

Figure 5.1 shows how the size of the input graph affects the performance. As we can see, the overall performance is strongly affected by stage 3. Recall that stage 3 (3.2.5) does a matrix multiplication and produces the matrix with the intersections of the neighborhoods. The adjacency matrix is split in horizontal blocks, where each one contains some rows of the matrix. As the graph size increases, so does the adjacency matrix. Maintaining the same number of reducers, means that each reducer receives and outputs larger blocks. Hence, the in-memory computations and the I/O operations increase and as a result the overall time increases. The only case that stage 3 is not the most time consuming stage is for graph "Gnutella". Here, stage 4 takes more time to complete. This is explained by the fact that *Gnutella* is very sparse, hence the adjacency matrix A that is read by stage 3, is possible to contain many zero rows, which are not written in the filesystem. On the contrary, the output matrix (M) of stage 3 is complete, meaning that stage 3 reads more data, in comparison to the other graphs.

Also notice the small decrease of execution time for stage 2, in the graph *Gnutella* in comparison to the one of graph $Trust$. This is explained by the fact that this graph has a lot fewer edges. *Gnutella* has 147,892 edges, while $Trust$ has 487,183. This means that stage 2 receives less node-pairs at hops $1 \ldots 3$. This is also shown by the output data of stage 1 in Table 5.6, where is equal to 81.9 MB. In comparison, graph $Trust$ *Epinions* (5.5) outputs 31.5 GB. So, the density of a graph is a parameter affecting the execution time.

Observing Figure 5.2 we see that the neighborhood depth also affects the performance. Here, stage 1 is the one that more or less controls the overall execution time. As we extend more the neighborhood depth, so do the neighbors of each node and respectively the size of the output data of stage 1. We also see the effectiveness of using the matrix based implementation of our algorithm in comparison to the list

based implementation in the execution time of stage 3. As the depth increases, the time increases almost linearly, while for example the time of stage 1 increases at a higher rate. This is explained by the constant space that is required for the matrix storage.

Figure 5.2 shows the scale up, or in other words the rank of parallelization that we have achieved. The major improvement is observed in stage 4 (3.2.6). Stage 4 receives the adjacency matrix A of the graph (without the neighborhood extension), the matrix M with the intersections of the neighborhoods of the nodes and the size of the neighborhood of each node v, S_v. Let us remind that a custom partitioner is used, in order to sort the incoming data. Each reducer receives a portion of A, the same portion of M and the corresponding S_v. They are sorted, so as to come by row id and then the jaccard coefficient is calculated. Having less compute nodes and therefore less reducers means that each reducer receives a bigger portion of A and M. The bottleneck here that increases dramatically the execution time is the sorting of the data.

Also, we see that the fluctuation in time of the"heavy" stage of our algorithm, stage 3, is not big. This happens because there may be more in-memory computations, but the I/O operations are decreased drammatically. With one compute node and four reducers the adjacency matrix is replicated four times and stage 3 reads far less data. Given the fact that the input graph isn' t too big, the I/O operations play a more important role.

Finally, the execution time of the other stages decreases linearly, converging to a constant value. Here we have a tradeoff between parallelization and network traffic. If we run this experiment in more compute nodes, we may even observe a slight increase in time, which would be explained by the network traffic and the overhead of starting new tasks. A good balance between all these parameters offers the best performance.

The last task is to evaluate the extended Jaccard coefficient in real graphs. We run experiments extending the neighborhoods at depths varying from 1 to 4. The results match with those at the work of Z. Huang et al. in [48]. Precision and recall are defined by the equations 5.1 and 5.2.

As neighborhood depth increases, the precision decreases and the recall increases. When we explore longer paths, the size of the neighborhoods increases and so does the number of predicted edges, often exceeding the number of the actual new edges. As the number of the predicted edges goes bigger, we predict more correctly the appearance of new edges. However, predicted edges are increasing at a bigger rate than the correct edges. So, as we explore neighborhoods at bigger depths, the denominator of the precision increases more than the nominator and the precision decreases.

On the other hand the denominator of recall is always constant and equal to the actual number of new edges. Hence, bigger depths cause a bigger nominator and the recall increases.

In figure 5.5, we observe how the graph size affects the execution time of Katz algorithm. Here, we find paths of length 3 and the stage that controls the performance is stage 2 (4.2.4). The main reason behind the steep increase of the curve is the I/O operations. As we can see from Table 5.20, the output data of stage 1 (2^{nd} iteration) for *Trust Epinions* is 79 GB. Probably, there is the case of many random I/O operations and a lot of network traffic. This amount of data is expected as this graph is dense, having 49,288 nodes and 487,183 edges. Watch that graph *Gnutella* runs much faster than graph *trust*. This happens because *Gnutella* is very sparse, having only 147,892 edges in 62,586 and effective diameter equal to 6.7. So with paths of length 3, each vertex explores almost less than the half of the vertices.

Figure 5.6 shows the effect of the path length on the performance of the algorithm.

Again, the bottleneck here is the reads and writes of massive data in HDFS. The problem is that we emit paths of different lengths from one node to another, separately. For example, for length=4, in the third iteration of stage 1, we cannot merge paths of length 1, with the ones of length 2 and 3, because we need the length-1 and length-3 paths to compute paths of length 4. Also, every path contains additional information about the route of the path, in order to prevent the creation of cycles. Examining Table 5.24 we see that the intermediate data that are produced are 37.9 GB.

Figure 5.7 shows the scale up that we achieves our algorithm, by running experiments in $1 \ldots 5$ compute nodes. As we can see the time is decreasing linearly as we add more compute nodes, converging to a constant value.

Finally, we evaluated the reliability of the Katz score in link prediction. We measured the precision and recall extending paths from 2 to 4, using two values for β, 0.3 and 0.5. β assigns a weight to each path, with longer paths having lower weights. For both values of β, as we explore longer paths the number of our predictions becomes bigger and so do the correct predictions. However, the number of predictions rises at a higher rate than the one of the correct predictions. Hence, the precision decreases with the increase of path length. On the contrary, recall increases as the nominator (correct predictions) increases, while the denominator stays the same (new edges). We also observe that precision and recall decrease and increase linearly.

Chapter 6

Conclusions and Future Work

The motivation of our work has been to develop a parallel Map/Reduce algorithm in Hadoop to handle a graph mining task. This task was chosen to be link prediction, which predicts the occurence of future edges in a given graph. Link prediction has many applications in network theory according to the relations that a network expresses. Such include recommendation systems, social network analysis and the discovery of patterns in a computer network traffic.

However, sequential algorithms cannot address the problem of data that occurs in real world networks. Hadoop is a tool that offers us the possibility to easily write parallel algorithms without caring about parallelization details like the communication of machines, the distribution of data, the replication and fault tolerance. All that is needed for a programmer, is to supply the implementation of a map and a reduce function. Hadoop is a powerfull tool and has already been used in graph mining algorithms like counting triangles ([33]), detecting components, finding the diatemer ([30]), link prediction ([46]) etc.

The contribution of our work is to supply two scalable algorithms that predict links among nodes by exploring paths of several lengths; the extended Jaccard coefficient and the Katz score. We performed several experiments in graphs with sizes from 7,000 to 60,000 nodes and deducted interesting results.

The experimental process has led us to the conclusion that the performance of our parallel algorithms is totally controlled by the I/O operations which are pretty heavy due to the large size of intermediate data that they produce, especially in dense graphs. However, we have proved through experimental studies that they scale up well and can be used in massive graphs, if we have a big cluster at our disposal.

Future work could focus on the effort of reducing the size of the intermediate data by adopting techniques of data compression or implementations that compute matrix approximations. Furthermore, future work includes the development in Hadoop of more link prediction algorithms like the Adamic/Adar coefficient or more graph mining tasks like finding the betweeness centrality of nodes.

Bibliography

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

[2] J. Dean, S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. OSDI, 2004

[3] M. Newman. The Structure and Function of Complex Networks. SIAM Review 45: 167-256, 2003

[4] Paul Jaccard (1901). Étude comarative de la distribution florale dans une portion des Alpes et des Jura. Bulletin de la Société Vaudoise des Sciences Naturelles 37, 547-579

[5] L. Katz. A new status index derived from sociometric analysis. Psychometrika, 18(1), March 1953.

[6] G. Salton, M. McGill. Introduction to Modern Information Retrieval. McGraw-Hill, 1983

[7] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein. Introduction to Algorithms. MIT Press

[8] John E. Hopcroft and Robert E. Tarjan. Efficient algorithms for graph manipulation. Communications of the ACM, 16(6):372–378, 1973.

[9] L. Freeman. Centrality in social networks: Conceptual clarifications. Social Networks, 1:215–239, 1979.

[10] D. J. Watts, S. H. Strogatz, (1998) Nature (London) 393, 440–442.

[11] Strogatz, S. H. (2001) Nature (London) 410, 268–276.

[12] Stanley, Milgram. "The Small World Problem". Psychology Today, 1(1), May 1967. pp 60 – 67

[13] Jeffrey Travers, Stanley Milgram. 1969. "An Experimental Study of the Small World Problem." Sociometry, Vol. 32, No. 4, pp. 425-443.

[14] D. J. Watts. Six degrees : the science of a connected age. 2003.

[15] D. J. Watts, S. H. Strogatz. Collective dynamics of 'small-world' networks. Nature, 393:440–442, 1998.

[16] J. Leskovec and E. Horvitz. Worldwide buzz: Planetary-scale views on an instant-messaging network. Technical report, Microsoft Research, June 2007.

[17] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In Proc. of ACM SIGKDD, pages 177–187, Chicago, Illinois, USA, 2005. ACM Press.

[18] U. Kang, C. Tsourakakis, A. Appel, C. Faloutsos, J. Leskovec. HADI: Fast Diameter Estimation and Mining in Massive Graphs with Hadoop.

[19] R. Albert, A. L. Barabasi. Emergence of Scaling in Random Networks. Science, 286: 509-512, 1999.

[20] M. Faloutsos, P. Faloutsos, C. Faloutsos. On Power-Law Relationships of the Internet Topology. SIGCOMM, p. 251-262.

[21] D. Chakrabarti, Y. Zhan, and C. Faloutsos. R-MAT: A recursive model for graph mining. SIAM Int. Conf. on Data Mining, Apr. 2004.

[22] M. E. J. Newman. Power laws, Pareto distributions and Zipf's law, December 2004.

[23] M. E. J. Newman, D. J. Watts, S. H. Strogatz. Random Graph Models of Social Networks. Proc Natl Acad Sci USA 99:2566–2572, 2002.

[24] P. Erdös, A. Rényi. On the evolution of random graphs. Publication of the mathematical institute of the Hungarian academy of science, 5:17-61, 1960.

[25] R. Kumar, J. Novak, A. Tomkins. Structure and evolution of online social networks. In KDD '06: Proceedings of the 12th ACM SIGKDD International Conference on Knowedge Discover and Data Mining, pages 611–617, New York, 2006.

[26] Y. Shiloach and U. Vishkin, "An o(logn) parallel connectivity algorithm," Journal of Algorithms, pp. 57–67, 1982.

[27] B. Awerbuch and Y. Shiloach, "New connectivity and msf algorithms for ultracomputer and pram," ICPP, 1983.

[28] D. Hirschberg, A. Chandra, and D. Sarwate, "Computing connected components on parallel computers," Communications of the ACM, vol. 22, no. 8, pp. 461–464, 1979.

[29] J. Greiner, "A comparison of parallel algorithms for connected components," Proceedings of the 6th ACM Symposium on Parallel Algorithms and Architectures, June 1994.

[30] U. Kang, Charalampos E. Tsourakakis, Christos Faloutsos, "PEGASUS: A Peta-Scale Graph Mining System," icdm, pp.229-238, 2009 Ninth IEEE International Conference on Data Mining, 2009

[31] Mary McGlohon, Leman Akoglu, Christos Faloutsos, Weighted graphs and disconnected components: patterns and a generator, Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining, August 24-27, 2008, Las Vegas, Nevada, USA

[32] C. E. Tsourakakis. Fast counting of triangles in large real networks, without counting: Algorithms and laws. In ICDM, 2008.

[33] C. E. Tsourakakis, U. Kang, G. L. Miller, C. Faloutsos. Doulion: Counting triangles in massive graphs with a coin. KDD, 2009

[34] D. Liben-Nowell, J. Kleinberg. The link prediction problem for social networks. In International Conference on Information and Knowledge Management (CIKM), pages: 556-559, 2003.

[35] L. Adamic, E. Adar. Friends and neighbors on the web. Soc. Networks, 25(3), 2003.

[36] A. Barabasi, H. Jeong, Z. Néda, E. Ravasz, A. Schubert, T. Vicsek. Evolution of the social network of scientific collaboration. Physica A, 311(3–4), 2002.

[37] M. Newman. Clustering and preferential attachment in growing networks. Phys. Rev. E, 64(025102), 2001.

[38] G. Jeh, J. Wisdom. SimRank: A measure of structural-content similarity. In KDD, 2002.

[39] Z. Huang. Link Prediction Based on Graph Topology: The Predictive Value of Generalized Clustering Coefficient. LinkKDD 2006.

[40] H. Tong, C. Faloutsos, Y. Koren. Fast direction-aware proximity for graph mining. Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining, August 12-15, 2007, San Jose, California, USA

[41] J. O'Madadhain, J. Hutchins, P. Smith. Prediction and ranking algorithms for event-based network data. SIGKDD Explorations 7(2), December 2005.

[42] A. Popescul, L. H. Ungar. Statistical Relational Learning for Link Prediction. In IJCAI Workshop on Learning Statistical Models from Relational Data. 2003.

[43] R. Chellappa, A. Jain. Markov random fields: theory and applications. Academic Press, Boston, 1993.

[44] B. Taskar, M. F. Wong, P. Abbeel, and D. Koller. Link prediction in relational data. In Neural Information Processing Systems Conference, Vancouver, Canada, December 2003.

[45] P. Domingos, M. Richardson. Markov Logic: A unifying framework for statistical relational learning. In ICML-2004 Workshop on Statistical Relational Learning and its Connections to Other Fields, pages 49—54, 2004.

[46] J. Bank and B. Cole, Calculating the Jaccard Similarity Coefficient with Map Reduce for Entity Pairs in Wikipedia. December 2008.

[47] E. Acar, D. M. Dunlavy, T. G. Kolda. Link prediction on evolving data using matrix and tensor factorizations. ICDM Workshops, 262–269, 2009.

[48] Zan Huang, Xin Li, Hsinchun Chen. Link prediction approach to collaborative filtering, In Proceedings of the Joint Conference on Digital Libraries (JCDL05). ACM, 2005, p. 7-11.

[49] http://snap.stanford.edu/data/#socnets

[50] http://www.trustlet.org/wiki/Trust_network_datasets

[51] J. Leskovec, D. Huttenlocher, J. Kleinberg. Signed Networks in Social Media. CHI 2010.

[52] J. Leskovec, D. Huttenlocher, J. Kleinberg. Predicting Positive and Negative Links in Online Social Networks. WWW 2010.

[53] J. Leskovec, J. Kleinberg and C. Faloutsos. Graph Evolution: Densification and Shrinking Diameters. ACM Transactions on Knowledge Discovery from Data (ACM TKDD), 1(1), 2007.

[54] J. Leskovec, J. Kleinberg and C. Faloutsos. Graphs over Time: Densification Laws, Shrinking Diameters and Possible Explanations. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2005.

[55] J. Gehrke, P. Ginsparg, J. M. Kleinberg. Overview of the 2003 KDD Cup. SIGKDD Explorations 5(2): 149-151, 2003.

[56] Massa, P.; Avesani, P. (2007). "Trust-aware Recommender Systems (link)". Proceeding of ACM Recommender Systems Conference, Minneapolis, Minnesota, USA.

[57] J. Leskovec, J. Kleinberg and C. Faloutsos. Graph Evolution: Densification and Shrinking Diameters. ACM Transactions on Knowledge Discovery from Data (ACM TKDD), 1(1), 2007.

[58] M. Ripeanu and I. Foster and A. Iamnitchi. Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design. IEEE Internet Computing Journal, 2002.