Mining Tera-Scale Graphs: Theory, Engineering and Discoveries

U Kang

May 2012

CMU-CS-12-112

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:
Christos Faloutsos, chair
Tom Mitchell
Garth Gibson
Robert Grossman, University of Chicago

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This work was partially sponsored by the U.S. Army Research Office (ARO) and Defense Advanced Research Projects Agency (DARPA) under Contract Number W911NF-11-C-0088. Research was also partially sponsored by the Defense Threat Reduction Agency under contract No. HDTRA1-10-1-0120, the Army Research Laboratory under Cooperative Agreement Number W911NF-09-2-0053, a Google Focused Research Award, a Yahoo Research Alliance Gift, and the Gordon and Betty Moore foundation.

Keywords: graph mining, MAPREDUCE, HADOOP, graph structure analysis, radius plot, diameter, connected component, inference, spectral graph analysis, eigensolver, tensor analysis, graph management, graph indexing, graph compression

To my family

Abstract

How do we find patterns and anomalies, on graphs with billions of nodes and edges, which do not fit in memory? How to use parallelism for such Tera- or Peta-scale graphs? In this thesis, we propose PEGASUS, a large scale graph mining system implemented on the top of the HADOOP platform, the open source version of MAPREDUCE. PEGASUS includes algorithms which help us spot patterns and anomalous behaviors in large graphs.

PEGASUS enables the structure analysis on large graphs. We unify many different structure analysis algorithms, including the analysis on connected components, PageRank, and radius/diameter, into a general primitive called GIM-V. GIM-V is highly optimized, achieving good scale-up on the number of edges and available machines. We discover surprising patterns using GIM-V, including the 7-degrees of separation in one of the largest publicly available Web graphs, with 7 billion edges.

PEGASUS also enables the inference and the spectral analysis on large graphs. We design an efficient distributed belief propagation algorithm which infer the states of unlabeled nodes given a set of labeled nodes. We also develop an eigensolver for computing top k eigenvalues and eigenvectors of the adjacency matrices of very large graphs. We use the eigensolver to discover anomalous adult advertisers in the who-follows-whom Twitter graph with 3 billion edges. In addition, we develop an efficient tensor decomposition algorithm and use it to analyze a large knowledge base tensor.

Finally, PEGASUS allows the management of large graphs. We propose efficient graph storage and indexing methods to answer graph mining queries quickly. We also develop an edge layout algorithm for better compressing graphs.

Acknowledgments

First, I would like to thank my advisor Prof. Christos Faloutsos for guiding me through all the journeys in my Ph.D. study. His advice and encouragement made the study truly enjoyable, exciting, and fruitful. I also thank all the CMU Database group members and visitors for the wonderful moments we shared: Leman Akoglu, Alex Beutel, Rishy Chandy, Polo Chau, Robson L. F. Cordeiro, Sang-Wook Kim, Sunhee Kim, Danai Koutra, Yasuko Matsubara, Mary McGlohon, Vagelis Papalexakis, Aditya Prakash, Yasushi Sakurai, Pedro Olmo Vaz de Melo, and Hyung-Jeong Yang.

I had great collaborators and co-authors from whom I learned a lot about research: Ana Paula Appel, Misha Bilenko, Jure Leskovec, Ching-Yung Lin, Brendan Meeder, Spiros Papadimitriou, Jimeng Sun, Hanghang Tong, Babis Tsourakakis, and Dengyong Zhou.

I had amazing internship experiences at Microsoft Research and IBM T.J. Watson Research. I want to thank my mentors for the wonderful experiences and advices: Misha Bilenko, Manuel Reyes Gomez, Spiros Papadimitriou, Jimeng Sun, and Hanghang Tong.

I want to thank my thesis committee members for their advices and comments: Tom Mitchell, Garth Gibson, and Robert Grossman.

I also like to thank staff members at the CMU School of Computer Science for making my Ph.D. life run smoothly: Marilyn Walgora, Charlotte Yano, and Deborah Cavlovich.

Finally, I would like to thank my parents and sisters for their endless support and love.

Contents

Chapter 1

Introduction

Graphs are ubiquitous: computer networks, social networks, mobile call networks, the World Wide Web [Broder et al., 2000], protein regulation networks to name a few. The large volume of available data, the low cost of storage and the stunning success of online social networks and Web2.0 applications all lead to graphs of unprecedented size. Typical graph mining algorithms silently assume that the graph fits in the memory of a typical workstation, or at least on a single disk; the above graphs violate these assumptions, spanning multiple Giga-bytes, and heading to Tera- and Peta-bytes of data. As a consequence, the vast majority of large graphs has remained untouched.

This thesis aims to unleash the potential by making extremely scalable graph mining algorithms on distributed platforms. Specifically, we use MAPREDUCE [Dean and Ghemawat, 2004] and its open source version, HADOOP, for their extreme scalability, fault tolerance, and cheap cost of maintenance. This thesis address the answers to the following two questions.

Theory and Engineering. *How can we design and develop efficient* MAPREDUCE *algorithms for mining very large graphs with billions of nodes and edges?* There are several challenges to answer the question. First, how can we formulate many graph mining algorithms using simple operations that can be efficiently implemented on MAPREDUCE? Second, how to manage graphs efficiently so that storage spaces are minimized and graph mining queries can be answered quickly?

Discovery. *What are the patterns and anomalies that we can discover in very large, real-world graphs with billions of nodes and edges?* Large graphs have interesting patterns or regularities with regard to radius, connected components, triangles, etc. Discovering the patterns helps us spot anomalies which can be useful for applications ranging from cyber-security (computer networks), fraud-detection (phone companies), and spammer detection (social networks).

1.1 Overview

Toward the goal of enabling extremely scalable graph mining analysis, this thesis describes the theory, engineering, and discoveries on mining very large graphs. As a main contribution, we propose a carefully selected set of fundamental operations, that help answer the questions in the previous section, including diameter estimation, connected components, inference on graphs, solving eigenvalues, and tensor decomposition. We package all these operations in PEGASUS (`http://www.cs.cmu.edu/~pegasus`), which, to the best of our knowledge, is the first such library, implemented on the top of the HADOOP platform, the open source version of MAPREDUCE.

The works in this thesis divide into three groups: basic graph algorithms, advanced graph algorithms, and graph management.

1.1.1 Basic Graph Algorithms

What are the structures of very large graphs with billions of nodes and edges which do not fit in the memory of a single machine? How do we study them? In the first part we describe distributed algorithms based on MAPREDUCE/HADOOP for analyzing the structure of very large graphs.

Radius Plot (Chapter 3). We describe HAdoop DIameter and radii estimator (HADI), a carefully designed and fine-tuned distributed algorithm to compute the radii and the diameter of massive graphs. We employ HADI to study large, real world graphs, and report interesting discoveries including the 7-degrees of separation in the Web graph. Our optimization on HADI leads to 7.6$\times$ faster performance than the naive algorithm.

Generalized Iterative Matrix-Vector Multiplication (Chapter 4). We generalize the HADI operation (Chapter 3) to propose a graph mining primitive called Generalized Iterated Matrix-Vector multiplication (GIM-V) which unifies many different graph mining operations including PageRank, diameter estimation, and connected components. GIM-V is highly optimized, achieving good scale-up on the number of available machines, and linear running time on the number of edges. Our optimization on GIM-V leads to more than 5$\times$ faster performance than the naive algorithm.

1.1.2 Advanced Graph Algorithms

The second part deals with advanced graph mining algorithms to find patterns and anomalies in large graphs. The algorithms include inference in graph, spectral graph analysis, and tensor analysis.

Inference In Graph (Chapter 5). Given a graph and a set of labeled nodes, how can we infer the labels of initially unlabeled nodes? There is a standard graph inference algorithm called belief propagation; however, existing belief propagation algorithm is limited in its scalability. We propose HADOOP LINE GRAPH FIXED POINT (HA-LFP), an efficient distributed algorithm for inference in billion-scale graphs, using HADOOP platform. HA-LFP scales up linearly on the number of edges and machines.

Spectral Graph Analysis (Chapter 6). We design and implement algorithms for spectral analysis of graphs: i.e., to study the eigenvalues and eigenvectors of graph adjacency matrices. Spectral analysis on graphs leads to many interesting applications including triangle counting, and our proposed HEIGEN algorithm handles 1000$\times$ larger matrices than the state of the art.

Tensor Analysis (Chapter 7). We study tensors, or multi dimensional arrays: e.g., predicates (subject, verb, object) in knowledge bases, and hyperlinks/anchor texts in Web graphs. We generalize the spectral analysis algorithm to multiple dimensions, and propose GIGATENSOR, a large scale tensor decomposition algorithm which solves more than 100$\times$ bigger problems than existing methods. We study a large knowledge base tensor, and present interesting findings which include the discovery of potential synonyms among millions of noun-phrases.

1.1.3 Graph Management

Given a very large graph which do not fit in the disk or the memory of a single machine, how to store, compress, and index it so that graph mining queries can be answered quickly? In the third part we tackle the graph management problem.

Graph Storage and Indexing (Chapter 8). We describe GBASE, a scalable and general graph management system. GBASE provides a parallel indexing mechanism for graph mining operations that both saves storage space, as well as accelerates queries. GBASE reduces the storage space and the running time up to 50×.

Edge Layout (Chapter 9). We study edge layout problem: given a graph, reorder nodes so that the nonzeros (edges) of the adjacency matrix are well-clustered. Better layout of edges of graphs leads to better compression; however, existing methods based on the assumptions of the existence of clear-cut communities do not work nicely on real world graphs. We propose SLASHBURN, an edge layout algorithm which utilizes the characteristic of real world graphs to compress graphs well. SLASHBURN outperforms all the state of the art algorithms in terms of the compression ratio, and the running time for graph mining queries.

1.2 Contribution

We summarize the contribution of this thesis.

Basic Graph Algorithms:

- We develop HADI, a large scale radius/diameter computation algorithm. HADI scales linearly on the number of edges, and the optimized version of HADI is 7.6× faster than the naive algorithm.
- We develop GIM-V, a general primitive for many different graph mining operations. GIM-V scales linearly on the number of edges, and the optimized version of GIM-V is more than 5× faster than the naive algorithm.
- We are the first to discover the 7-degrees of separation of the Web.

Advanced Graph Algorithms:

- We develop HA-LFP, an efficient distributed belief propagation algorithm. HA-LFP is the first inference algorithm to handle billion-scale graphs.
- We develop HEIGEN, an eigensolver to perform spectral analysis on large graphs. HEIGEN analyzes 1000× larger matrices than the state of the art.
- We develop GIGATENSOR, a large scale tensor decomposition algorithm. GIGATENSOR handles more than 100× larger tensors than the state of the art.

Graph Management:

- We develop GBASE, a scalable and general graph management system. GBASE reduces the storage space and the running time up to 50×.
- We develop SLASHBURN, an edge layout algorithm for better compressing graphs. SLASHBURN outperforms all the state of the art algorithms in terms of compression ratio, and the running time for graph mining queries.

1.3 Technology Transfer

Our work in large graph mining and the PEGASUS system have impacts in academia as well as in industry. We summarize the technology transfers of our work.

- The PEGASUS system has been downloaded more than 410 times from 83 countries. It led to two U.S. patents, and won the award at the open source software world challenge.
- Microsoft included PEGASUS as part of their HADOOP distribution for Windows Azure.
- PEGASUS system is used as one of the core systems for several DARPA projects including Anomaly Detection At Multiple Scale (ADAMS).

Chapter 2

Survey

In this chapter we review the related works in large scale graph mining and MAPREDUCE/HADOOP. We also summarize the symbols and the dataset used.

2.1 Large Scale Graph Mining

Given a very large graph spanning Terabytes or Petabytes, how to find patterns and anomalies? Large scale graph mining poses challenges in dealing with massive amount of data. We review several approaches to large graph mining.

Single Machine. An obvious option is to use a single machine to analyze graphs. However, this option is not viable for graphs whose size is larger than the memory or disks of a single machine.

Sampling. One might consider using a sampling approach to decrease the amount of data. However, sampling from a large graph can lead to multiple nontrivial problems that do not have satisfactory solutions [Leskovec and Faloutsos, 2006]. For example, which sampling methods should we use? Should we get a random sample of the edges, or the nodes? Both options have their own share of problems: the former gives poor estimation of the graph diameter, while the latter may miss high-degree nodes. For this reason, we avoid using sampling methods.

Distributed Computing. For handling large graphs which span several disks of multiple machines, we need distributed computing platforms. Among many distributed computing platforms, we chose MAPREDUCE/HADOOP, a programming framework [Dean and Ghemawat, 2004] for processing Web-scale data, for its nice scalability, fault tolerance, and low maintenance costs. In the next subsection we provide a brief overview of MAPREDUCE/HADOOP.

2.2 MAPREDUCE and HADOOP

MAPREDUCE is a programming framework [Dean and Ghemawat, 2004, Aggarwal et al., 2004] for processing huge amounts of unstructured data in a massively parallel way. MAPREDUCE has two major advantages: (a) the programmer is oblivious of the details of the data distribution, replication, load balancing etc. and furthermore (b) the programming concept is familiar, i.e., the concept of functional programming.

Symbol	Definition
G	a graph
V	set of nodes in a graph
E	set of edges in a graph
d	diameter of a graph
$H(.)$	Shannon entropy function
GCC	giant connected component of a graph

Table 2.1: Table of symbols.

Briefly, the programmer needs to provide only two functions, a *map* and a *reduce*. The typical framework is as follows [Lämmel, 2008]: (a) the *map* stage sequentially passes over the input file and outputs (key, value) pairs; (b) the *shuffling* stage groups of all values by key, and (c) the *reduce* stage processes the values with the same key and outputs the final result.

HADOOP is the open source implementation of MAPREDUCE. HADOOP provides the Distributed File System (HDFS) and PIG, a high level language for data analysis [Olston et al., 2008]. Due to its power, simplicity, fault tolerance, and low maintenance costs, HADOOP is a very promising tool for large scale graph mining applications, something already reflected in academia (e.g., see [Papadimitriou and Sun, 2008, Kang et al., 2009, 2010, 2011e,d,a,b, Cordeiro et al., 2011, Kang et al., 2011c]). In addition to PIG, there are several high-level language and environments for advanced MAPREDUCE-like systems, including Sphere [Grossman and Gu, 2008], SCOPE [Chaiken et al., 2008], and Sawzall [Pike et al., 2005].

2.3 Table of Symbols

We list the common symbols used in this thesis in Table 2.1. Each chapter also defines chapter-specific symbols.

2.4 Table of Dataset

We summarize the dataset used in this thesis in Table 2.2 with the following details. Each chapter also contains the list of the data used.

- YahooWeb: Web pages and their links, crawled by Yahoo! at year 2002.
- Twitter: social network (who follows whom) extracted from Twitter, in June 2010 and Nov 2009.
- VoiceCall: phone call records (who calls whom) during Dec. 2007 to Jan. 2008 from an anonymous phone service provider.
- LinkedIn: social network (who connects to whom) from LinkedIn.
- SMS: short message service records (who sends to whom) during Dec. 2007 to Jan. 2008 from an anonymous phone service provider.
- Patents: U.S. patents citations from 1975 to 1999 (`http://www.nber.org/patents`).
- LiveJournal: friendship social network from LiveJournal.

Graph	Nodes	Edges	File Size	Type	Description
YahooWeb	1.4 B	6.6 B	0.12 TB	real	WWW links in 2002
Twitter'10	104 M	3.7 B	0.13 TB	real	person-person
Twitter'09	63 M	1.8 B	56 GB	real	person-person
VoiceCall	30 M	260 M	8.4 GB	real	who calls whom
LinkedIn	7.5 M	58 MB	1 GB	real	person-person in 2006
	4.4 M	27 MB	490 MB		person-person in 2005
	1.6 M	6.8 MB	121 MB		person-person in 2004
	85 K	230 KB	4 MB		person-person in 2003
SMS	7 M	38 M	629 MB	real	who sends to whom
Patents	6 M	16 M	264 MB	real	patent-patent
LiveJournal	4.8 M	69 M	1.1 GB	real	friendship social network
Wikipedia	3.5 M	42 M	605 MB	real	doc-doc in 2007/02
	3 M	35 M	495 MB		doc-doc in 2006/09
	1.6 M	18.5 M	252 MB		doc-doc in 2005/11
DBLP	471 K	112 K	1 MB	real	document-document
WWW-Barabasi	325 K	1.5 M	20 MB	real	WWW links in nd.edu
Flickr	404 K	2.1 M	28 MB	real	person-person
Enron	80 K	313 K	11 MB	real	Enron email
Epinions	75 K	508 K	5 MB	real	who trusts whom
AS-Oregon	14 K	75 K	385 KB	real	router connetions
Kronecker	177 K	2 B	25 GB	synthetic	from Kronecker generator [Leskovec et al., 2005]
	121 K	1.1 B	13.9 GB		
	59 K	282 M	3.3 GB		
	20 K	40 M	439 MB		
Erdős-Rényi	177 K	2 B	25 GB	synthetic	random $G_{n,p}$
	121 K	1.1 B	13.9 GB		
	59 K	282 M	3.3 GB		
	20 K	40 M	439 MB		

Table 2.2: Datasets. B: Billion, M: Million, K: Thousand, TB: Terabytes, GB: Gigabytes, MB: Megabytes, KB: Kilobytes.

- Wikipedia: citation network from Wikipedia articles.
- DBLP: DBLP document to document network (`www.informatik.uni-trier.de/~ley/db/`).
- WWW-Barabasi: WWW links inside nd.edu [Albert et al., 1999].
- Flickr: social network from Flickr.
- Enron: email network (who sends to whom) [Klimt and Yang, 2004].
- Epinions: who trusts whom social network [Richardson et al., 2003].
- AS-Oregon: router connections network.
- Kronecker: synthetic Kronecker graph [Leskovec et al., 2005] with similar properties as real-world graphs.
- Erdős-Rényi: synthetic random graphs $G_{n,p}$ [Erdős and Rényi, 1959].

Part I

Basic Graph Algorithms

Part I - Basic Graph Algorithms: Overview

What are the structures of very large graphs with billions of nodes and edges which do not fit in the memory of a single machine? How do we study them? In this part we describe distributed graph structure analysis algorithms based on MAPREDUCE/HADOOP, and use them to analyze the structures of very large graphs.

First, we describe HAdoop DIameter and radii estimator (HADI), a carefully designed and fine-tuned distributed algorithm to compute the radii and the diameters of massive graphs. We employ HADI to study large, real world graphs, and report interesting discoveries including the 7-degrees of separation in the Web graph.

Second, we generalize the HADI operation to propose a graph mining primitive called Generalized Iterated Matrix-Vector multiplication (GIM-V) which unifies many different graph mining operations including PageRank, diameter estimation, and connected components. GIM-V is highly optimized, achieving good scale-up on the number of available machines, and linear running time on the number of edges. We employ GIM-V to study anomalous connected components in the Web graph.

Chapter 3

Radius Plot

Given large, multi-million node graphs (e.g., Facebook, Web-crawls, etc.), how do they evolve over time? How are they connected? What are the central nodes and the outliers? In this chapter we define the Radius plot of a graph and show how it can answer these questions. However, computing the Radius plot is prohibitively expensive for graphs reaching the planetary scale.

There are two major contributions in this chapter: (a) We propose HAdoop DIameter and radii estimator (HADI), a carefully designed and fine-tuned algorithm to compute the radii and the diameters of massive graphs, that runs on the top of the MAPREDUCE/HADOOP system, with excellent scale-up on the number of available machines, (b) We run HADI on several real world datasets including YahooWeb (*6B edges, 1/8 of a Terabyte*), one of the largest public graphs ever analyzed.

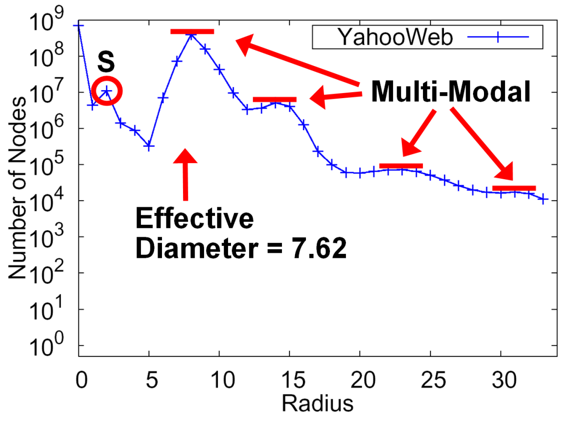
Thanks to HADI, we report fascinating patterns on large graphs, like the surprisingly small effective diameter, the multi-modal/bi-modal shape of the Radius plot, and its palindrome motion over time.

3.1 Introduction

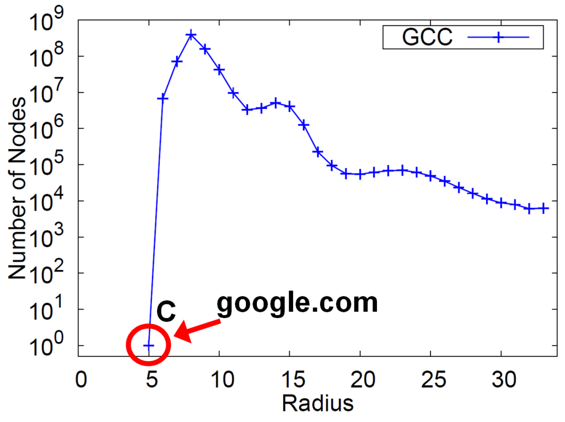
How do real, Terabyte-scale graphs look like? Is it true that the nodes with the highest degree are the most central ones, i.e., have the smallest radius (defined in Section 3.2)? How do we compute the diameter and node radii in graphs of such size?

This chapter addresses the answers to the questions above. The contributions of this chapter are the following:

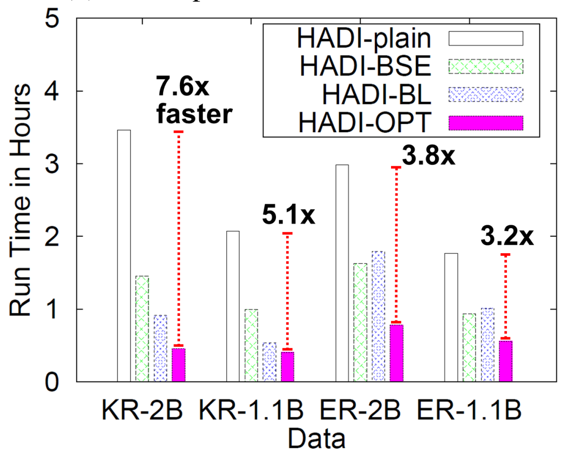
1. **Design.** We propose HADI, a scalable algorithm to compute the radii and diameter of network. As shown in Figure 3.1 (c), our method is *7.6*$\times$ faster than the naive version.
2. **Optimization and Experimentation.** We carefully fine-tune our algorithm, and we test it on one of the largest public Web graph ever analyzed, with several *billions* of nodes and edges, spanning 1/8 of a Terabyte.
3. **Observations.** Thanks to HADI, we find interesting patterns and observations, like the "Multi-modal and Bi-modal" pattern, and the surprisingly small effective diameter of the Web. For example, see the Multi-modal pattern in the radius plot of Figure 3.1, which also shows the effective diameter and the center node of the Web ('google.com').



(a) Radius plot of YahooWeb



(b) Radius plot of GCC of YahooWeb



(c) Running time of HADI

Figure 3.1: **(a):** Radius plot (Count versus Radius) of the YahooWeb graph. Notice the "effective diameter" is surprisingly small. Also notice the peak (marked 'S') at radius 2, due to star-structured disconnected components.
(b): Radius plot of GCC (Giant Connected Component) of YahooWeb graph. The *only* node with radius 5 (marked 'C') is `google.com`.
(c): Running time of HADI with/without optimizations for Kronecker and Erdős-Rényi graphs with billions edges. Run on the M45 HADOOP cluster, using 90 machines for 3 iterations. HADI-OPT is up to **7.6**× faster than HADI-plain.

The HADI algorithm (implemented in HADOOP) is available at `http://www.cs.cmu.edu/~pegasus`. The rest of the chapter is organized as follows: Section 3.2 defines related terms and a sequential algorithm for the Radius plot. Section 3.3 describes large scale algorithms for the Radius plot, and Section 3.4 analyzes the complexity of the algorithms and provides a possible extension. In Section 3.5 we present timing results, and in Section 3.6 we observe interesting patterns. After describing backgrounds in Section 3.7, we conclude in Section 3.8.

3.2 Preliminaries; Sequential Radii Calculation

3.2.1 Definitions

In this section, we define several terms related to the radius and the diameter. Recall that, for a node v in a graph G, the *radius* $r(v)$ is the distance between v and a reachable node farthest away from v. The *diameter* $d(G)$ of a graph G is the maximum radius of nodes $v \in G$. That is, $d(G) = \max_v r(v)$ [Lewis., 2009].

Since the radius and the diameter are susceptible to outliers (e.g., long chains), we follow the literature [Leskovec et al., 2005] and define the *effective* radius and diameter as follows.

Definition 1 (Effective Radius). *For a node v in a graph G, the effective radius $r_{eff}(v)$ of v is the 90th-percentile of all the shortest distances from v.*

Definition 2 (Effective Diameter). *The effective diameter $d_{eff}(G)$ of a graph G is the minimum number of hops in which 90% of all connected pairs of nodes can reach each other.*

The effective radius is very related to *closeness centrality* that is widely used in network sciences to measure the importance of nodes [Newman., 2005]. Closeness centrality of a node v is the mean shortest-path distance between v and all other nodes reachable from it. On the other hand, the effective radius of v is 90% quantile of the shortest-path distances. Although their definitions are slightly different, they share the same spirit and can both be used as a measure of the 'centrality', or the time to spread information from a node to all other nodes.

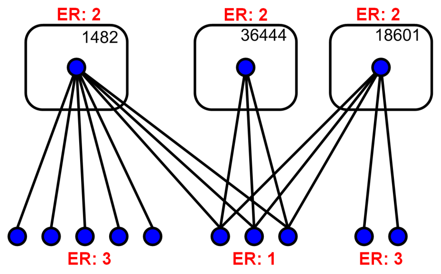
We will use the following three Radius-based Plots:

1. **Static Radius Plot** (or just "Radius plot") of graph G shows the distribution (count) of the effective radius of nodes at a specific time. See Figure 3.1 and 3.2 for the example radius plots of real-world and synthetic graphs.
2. **Temporal Radius Plot** shows the distributions of effective radius of nodes at several timestamps (see Figure 3.14 for an example).
3. **Radius-Degree Plot** shows the scatter-plot of the effective radius $r_{eff}(v)$ versus the degree d_v for each node v, as shown in Figure 3.12.

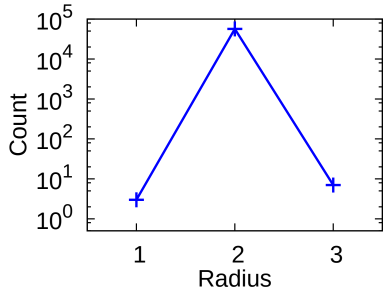
Table 3.1 lists the symbols used in this chapter.

3.2.2 Computing Radius and Diameter

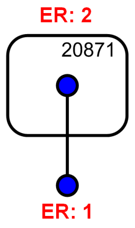
To generate the Radius plot, we need to calculate the effective radius of every node. In addition, the effective diameter is useful for tracking the evolution of networks. Therefore, we describe our algorithm for computing the effective radius and the effective diameter of a graph. As described in Section 3.7,



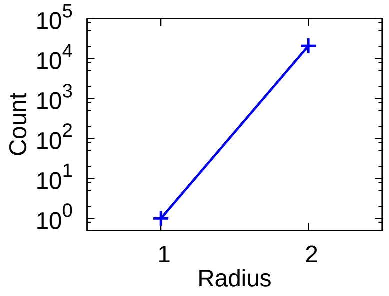
(a) Near-Bipartite-Core



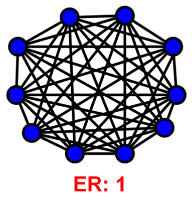
(b) Radius plot of the Near-Bipartite-Core graph



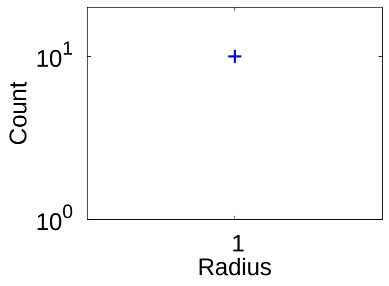
(c) Star



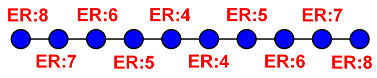
(d) Radius plot of the Star graph



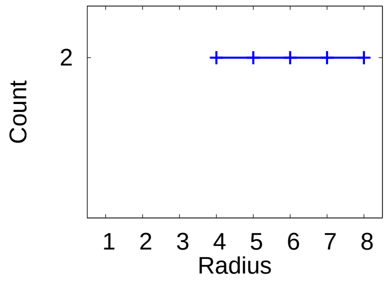
(e) Clique



(f) Radius plot of the Clique graph



(g) Chain



(h) Radius plot of the Chain graph

Figure 3.2: Radius plots of real-world and synthetic graphs. (a) and (c) are from the anomalous disconnected components of YahooWeb graph. (e) and (g) are synthetic examples to show the radius plots. ER means the Effective Radius of nodes. A node inside a rounded box represents N nodes specified in the top area of the box. For example, (c) is a compact representation of a star graph where the core node at the bottom is connected to 20,871 neighbors. Notice that Radius plots provide concise summary of the structure of graphs.

Symbol	Definition
n	number of nodes in a graph
m	number of edges in a graph
h	number of hops
$N(h)$	number of node-pairs reachable in $\leq h$ hops (neighborhood function)
$N(h,i)$	number of neighbors of node i reachable in $\leq h$ hops
$b(h,i)$	Flajolet-Martin bitstring for node i at h hops
$\hat{b}(h,i)$	Partial Flajolet-Martin bitstring for node i at h hops

Table 3.1: Table of symbols.

existing algorithms do not scale well. To handle graphs with billions of nodes and edges, we use the following two main ideas:

1. We use an approximation rather than an exact algorithm.
2. We design a parallel algorithm for MAPREDUCE/HADOOP (the algorithm can also run in a parallel RDBMS).

We first describe why exact computation is infeasible due to the huge output space of $O(n^2)$. Assume we have a 'set' data structure that supports two functions: *add()* for adding an item, and *size()* for returning the count of distinct items. With the set, radii of nodes can be computed as follows:

1. For each node i, make a set S_i and initialize by adding i to it.
2. For each node i, update S_i by adding one-step neighbors of i to S_i.
3. For each node i, continue updating S_i by adding 2,3,...-step neighbors of i to S_i. If the size of S_i before and after the addition does not change, then the node i reached its radius. Iterate until all nodes reach their radii.

Although simple and clear, the above algorithm requires too much space, $O(n^2)$, since there are n nodes and each node requires n space in the end. Since exact implementation is hopeless, we turn to an approximation algorithm for the effective radius and the diameter computation. For the purpose, we use the Flajolet-Martin algorithm [Flajolet and Martin, 1985, Palmer et al., 2002] for counting the number of distinct elements in a multiset. While many other applicable algorithms exist (e.g., [Beyer et al., 2007, Charikar et al., 2000, Garofalakis and Gibbons, 2001]), we choose the Flajolet-Martin algorithm because it gives an unbiased estimate, as well as a tight $O(\log n)$ bound for the space complexity [Alon et al., 1996].

The main idea of Flajolet-Martin algorithm is as follows. We maintain a bitstring *BITMAP*$[0 \ldots L-1]$ of length L which encodes the set. For each item to add, we do the following:

1. Pick an $index \in [0 \ldots L-1]$ with probability $1/2^{index+1}$.
2. Set *BITMAP*$[index]$ to 1.

Let R denote the index of the leftmost '0' bit in *BITMAP*. The main result of Flajolet-Martin is that the unbiased estimate of the size of the set is given by

$$\frac{1}{\varphi}2^R, \tag{3.1}$$

where $\varphi = 0.77351\cdots$. The more concentrated estimate can be get by using multiple bitstrings and averaging the R. If we use K bitstrings R_1 to R_K, the size of the set can be estimated by

$$\frac{1}{\varphi}2^{\frac{1}{K}\sum_{l=1}^{K} R_l}. \tag{3.2}$$

Picking an index from an item depend on a value computed from a hash function with the item as an input. Thus, merging the two set A and B is simply bitwise-OR'ing the bitstrings of A and B without worrying about the redundant elements. The application of Flajolet-Martin algorithm to radius and diameter estimation is straightforward. We maintain K Flajolet-Martin (FM) bitstrings $b(h, i)$ for each node i and current hop number h. $b(h, i)$ encodes the number of nodes reachable from node i within h hops, and can be used to estimate radii and diameter as shown below. The bitstrings $b(h, i)$ are iteratively updated until the bitstrings of all nodes stabilize. At the h-th iteration, each node receives the bitstrings of its neighboring nodes, and updates its own bitstrings $b(h - 1, i)$ handed over from the previous iteration:

$$b(h, i) = b(h - 1, i) \text{ BIT-OR } \{b(h - 1, j) | (i, j) \in E\}, \tag{3.3}$$

where "BIT-OR" denotes bitwise-OR function. After h iterations, a node i has K bitstrings that encode the *neighborhood function* $N(h, i)$, that is, the number of nodes within h hops from the node i. $N(h, i)$ is estimated from the K bitstrings by

$$N(h, i) = \frac{1}{0.77351}2^{\frac{1}{K}\sum_{l=1}^{K} b_l(i)}, \tag{3.4}$$

where $b_l(i)$ is the position of leftmost '0' bit of the l^{th} bitstring of node i. The iterations continue until the bitstrings of all nodes stabilize, which is a necessary condition that the current iteration number h exceeds the diameter $d(G)$. After the iterations finish at h_{max}, we calculate the effective radius for every node and the diameter of the graph, as follows:

- $r_{eff}(i)$ is the smallest h such that $N(h, i) \geq 0.9 \cdot N(h_{max}, i)$.
- $d_{eff}(G)$ is the smallest h such that $N(h) = \sum_i N(h, i) = 0.9 \cdot N(h_{max})$. If $N(h) > 0.9 \cdot N(h_{max}) > N(h - 1)$, then $d_{eff}(G)$ is linearly interpolated from $N(h)$ and $N(h - 1)$. That is, $d_{eff}(G) = (h - 1) + \frac{0.9 \cdot N(h_{max}) - N(h-1)}{N(h) - N(h-1)}$.

Algorithm 3.1 shows the summary of the algorithm described above.

The parameter K is typically set to 32 [Flajolet and Martin, 1985], and $MaxIter$ is set to 256 since real graphs have relatively small effective diameter. The NewFMBitstring() function in line 2 generates K FM bitstrings [Flajolet and Martin, 1985]. The effective radius $r_{eff}(i)$ is determined at line 21, and the effective diameter $d_{eff}(G)$ is determined at line 23.

Algorithm 3.1 runs in $O(dm)$ time, since the algorithm iterates at most d times with each iteration running in $O(m)$ time. By using approximation, Algorithm 3.1 runs faster than previous approaches (see Section 3.7 for discussion). However, Algorithm 3.1 is a sequential algorithm and requires $O(n \log n)$ space and thus can not handle extremely large graphs (more than billions of nodes and edges) which do not fit into a single machine. In the next sections we present efficient parallel algorithms.

Algorithm 3.1: Computing Radii and Diameter

```
Input: graph $G$,
    maximum iteration $MaxIter$, and
    number of bitstring $K$.
Output: $r_{eff}(i)$ of every node $i$, and
    diameter $d_{eff}(G)$.
1: for $i = 1$ to $n$ do
2:   $b(0, i) \leftarrow$ NewFMBitstring($n$);
3: end for
4: for $h = 1$ to $MaxIter$ do
5:   $Changed \leftarrow 0$;
6:   for $i = 1$ to $n$ do
7:     for $l = 1$ to $K$ do
8:       $b_l(h, i) \leftarrow b_l(h-1, i)$BIT-OR$\{b_l(h-1, j)|\forall j$ adjacent from $i\}$;
9:     end for
10:    if $\exists l$ s.t. $b_l(h, i) \neq b_l(h-1, i)$ then
11:      increase $Changed$ by 1;
12:    end if
13:  end for
14:  $N(h) \leftarrow \sum_i N(h, i)$;
15:  if $Changed$ equals to 0 then
16:    $h_{max} \leftarrow h$, and break for loop;
17:  end if
18: end for
19: for $i = 1$ to $n$ do
20:   // estimate eff. radii
21:   $r_{eff}(i) \leftarrow$ smallest $h'$ where $N(h', i) \geq 0.9 \cdot N(h_{max}, i)$;
22: end for
23: $d_{eff}(G) \leftarrow$ smallest $h'$ where $N(h') = 0.9 \cdot N(h_{max})$;
```

3.3 Proposed Method

In the next two sections we describe HADI, a parallel radius and diameter estimation algorithm. As mentioned in Section 3.2, HADI can run on the top of both a MAPREDUCE system and a parallel SQL DBMS. In the following, we first describe the general idea behind HADI and show the algorithm for MAPREDUCE. The algorithm for a parallel SQL DBMS is sketched in Section 3.4.

3.3.1 HADI Overview

HADI follows the flow of Algorithm 3.1; that is, it uses the Flajolet-Martin (FM) bitstrings and iteratively updates them using the bitstrings of its neighbors. The most expensive operation in Algorithm 3.1 is line 8 where bitstrings of each node are updated. Therefore, HADI focuses on the efficient implementation of the operation using MAPREDUCE framework.

It is important to notice that HADI is a disk-based algorithm; indeed, memory-based algorithm is not possible for Tera- and Peta-byte scale data. HADI saves two kinds of information to a distributed file system (such as HDFS (Hadoop Distributed File System) in the case of HADOOP):

- **Edge** has a format of ($srcid$, $dstid$).
- **Bitstrings** has a format of ($nodeid$, $bitstring_1$, ..., $bitstring_K$).

Combining the bitstrings of each node with those of its neighbors is very expensive operation which needs several optimizations to scale up near-linearly. In the following sections we will describe three HADI algorithms in a progressive way. That is we first describe HADI-naive, to give the big picture and explain why such a naive implementation should not be used in practice, then the HADI-plain, and finally HADI-optimized, the proposed method that should be used in practice.

3.3.2 HADI-naive in MAPREDUCE

HADI-naive is inefficient, but we present it for ease of explanation.

Data. The edge file is saved as a sparse adjacency matrix in HDFS. Each line of the file contains a nonzero element of the adjacency matrix of the graph, in the format of ($srcid$, $dstid$). Also, the bitstrings of each node are saved in a file in the format of ($nodeid$, $flag$, $bitstring_1$, ..., $bitstring_K$). The $flag$ records information about the status of the nodes (e.g., 'Changed' flag to check whether one of the bitstrings changed or not). Notice that we *do not know* the physical distribution of the data in HDFS.

Main Program Flow. The main idea of HADI-naive is to use the bitstrings file as a logical "cache" to machines which contain edge files. The bitstring update operation in Equation (3.3) of Section 3.2 requires that the machine which updates the bitstrings of node i should have access to (a) all edges adjacent from i, and (b) all bitstrings of the adjacent nodes. To meet the requirement (a), it is needed to reorganize the edge file so that edges with a same source id are grouped together. That can be done by using an Identity mapper which outputs the given input edges in ($srcid$, $dstid$) format. The most simple yet naive way to meet the requirement (b) is sending the bitstrings to every reducer which receives the reorganized edge file.

Thus, HADI-naive iterates over two-stages of MAPREDUCE. The first stage updates the bitstrings of each node and sets the 'Changed' flag if at least one of the bitstrings of the node is different from the previous

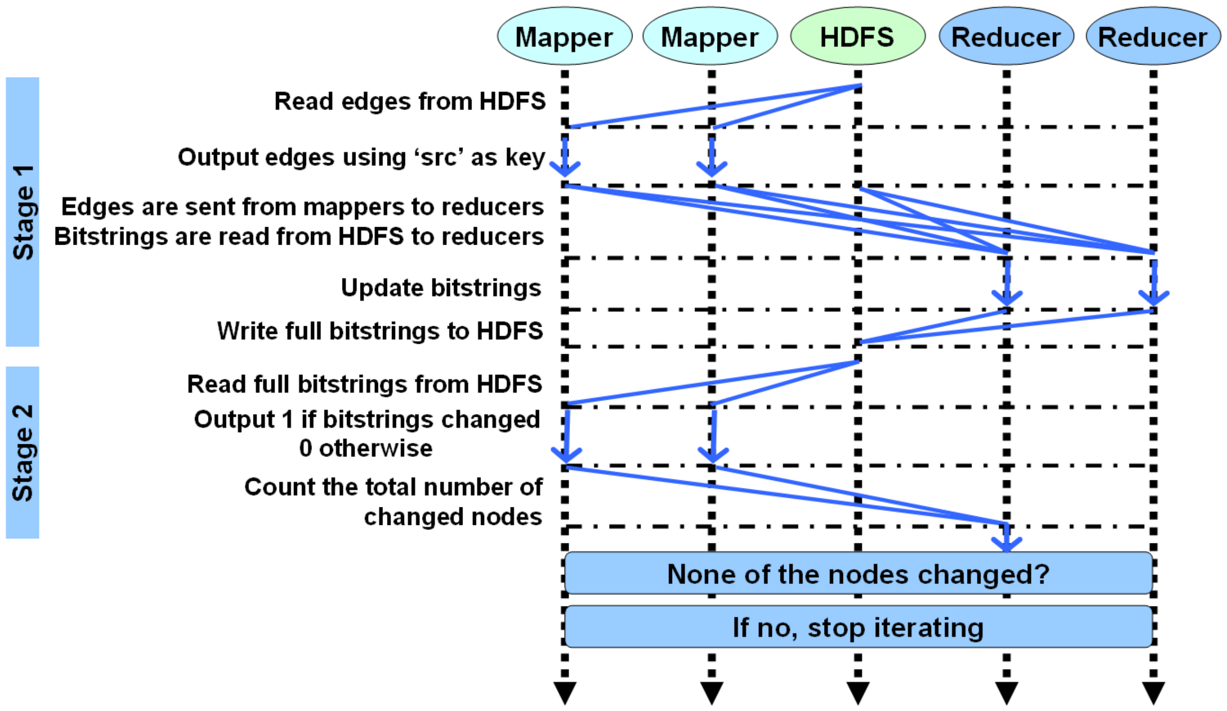

Mapper
Mapper
HDFS
Reducer
Reducer
Stage 1
Read edges from HDFS
Output edges using 'src' as key
Edges are sent from mappers to reducers
Bitstrings are read from HDFS to reducers
Update bitstrings
Write full bitstrings to HDFS
Stage 2
Read full bitstrings from HDFS
Output 1 if bitstrings changed
0 otherwise
Count the total number of
changed nodes
None of the nodes changed?
If no, stop iterating

Figure 3.3: One iteration of HADI-naive. *Stage 1*. Bitstrings of all nodes are sent to every reducer. *Stage 2*. Sums up the count of changed nodes.

bitstring. The second stage counts the number of changed nodes and stops iterations when the bitstrings stabilized, as illustrated in the swim-lane diagram of Figure 3.3.

Although conceptually simple and clear, HADI-naive is unnecessarily expensive, because it ships all the bitstrings to all reducers. Thus, we propose HADI-plain and additional optimizations, which we explain next.

3.3.3 HADI-plain in MAPREDUCE

HADI-plain improves HADI-naive by *copying only the necessary bitstrings to each reducer*. The details are next.

Data. As in HADI-naive, the edges are saved in the format of ($srcid$, $dstid$), and bitstrings are saved in the format of ($nodeid$, $flag$, $bitstring_1$, ..., $bitstring_K$) in files over HDFS. The initial bitstrings generation, which corresponds to line 1-3 of Algorithm 3.1, can be performed in a completely parallel way. The $flag$ of each node records the following information:

- **Effective Radii** and Hop Numbers to calculate the effective radius.
- **Changed** flag to indicate whether at least a bitstring has been changed or not.

Main Program Flow. As mentioned in the beginning, HADI-plain copies only the necessary bitstrings to each reducer. The main idea is to replicate bitstrings of node j exactly x times where x is the in-degree of node j. The replicated bitstrings of node j is called the *partial bitstring* and represented by $\hat{b}(h, j)$. The replicated $\hat{b}(h, j)$'s are used to update $b(h, i)$, the bitstring of node i where (i, j) is an edge in the

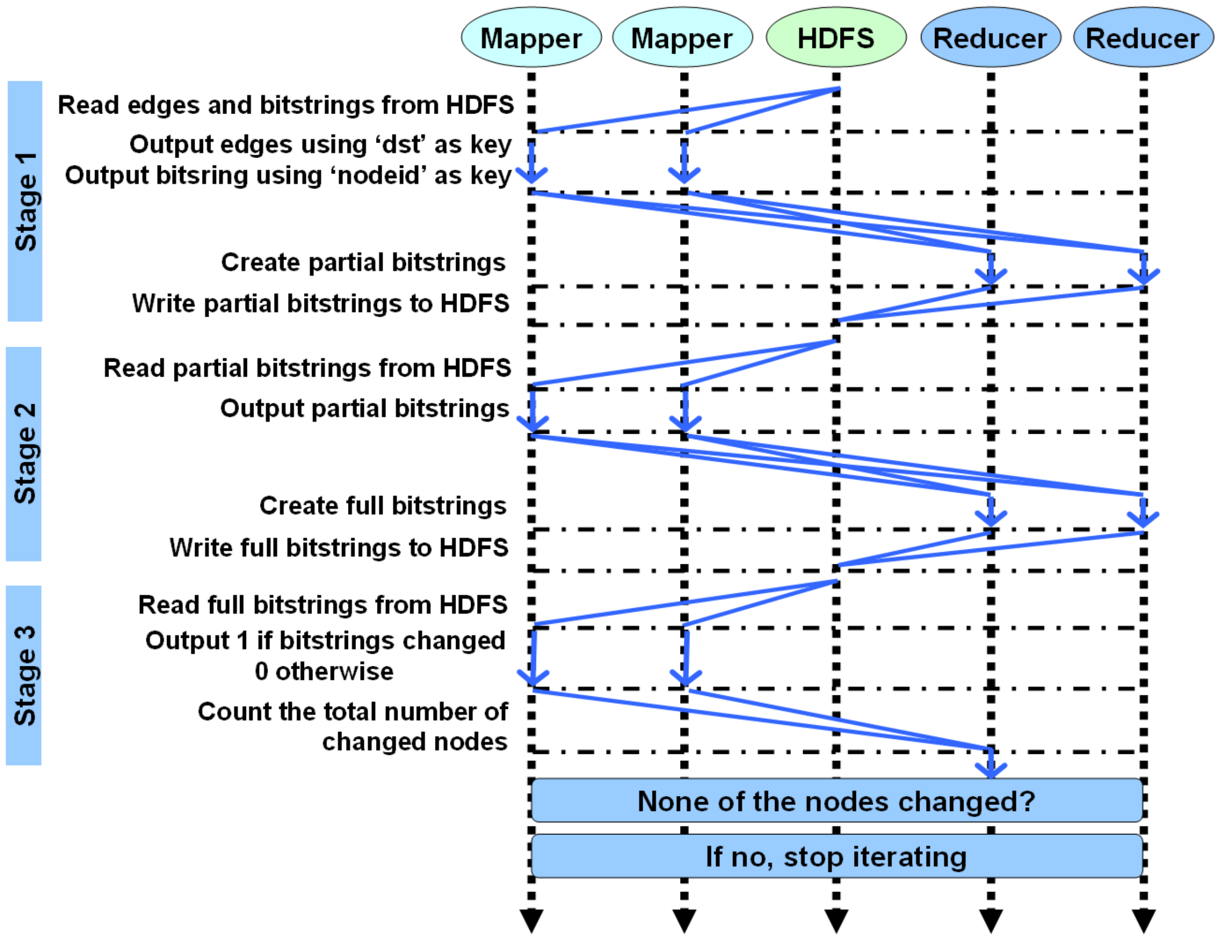

Mapper
Mapper
HDFS
Reducer
Reducer
Stage 1
Read edges and bitstrings from HDFS
Output edges using 'dst' as key
Output bitsring using 'nodeid' as key
Create partial bitstrings
Write partial bitstrings to HDFS
Stage 2
Read partial bitstrings from HDFS
Output partial bitstrings
Create full bitstrings
Write full bitstrings to HDFS
Stage 3
Read full bitstrings from HDFS
Output 1 if bitstrings changed
0 otherwise
Count the total number of
changed nodes
None of the nodes changed?
If no, stop iterating

Figure 3.4: One iteration of HADI-plain. *Stage 1.* Edges and bitstrings are matched to create partial bitstrings. *Stage 2.* Partial bitstrings are merged to create full bitstrings. *Stage 3.* Sums up the count of changed nodes, and compute N(h), the neighborhood function. Computing N(h) is not drawn in the figure for clarity.

graph. HADI-plain iteratively runs three-stage MAPREDUCE jobs until all bitstrings of all nodes stop changing. Algorithm 3.2, 3.3, and 3.4 shows HADI-plain, and Figure 3.4 shows the swim-lane. We use h for denoting the current iteration number which starts from h=1. Output(a,b) means to output a pair of data with the key a and the value b.

Stage 1. We generate (key, value) pairs, where the key is the node id i and the value is the partial bitstrings $\hat{b}(h, j)$'s where j ranges over all the neighbors adjacent from node i. To generate such pairs, the bitstrings of node j are grouped together with edges whose $dstid$ is j. Notice that at the very first iteration, bitstrings of nodes do not exist; they have to be generated on the fly, and we use the *Bitstring Creation Command* for that. Notice also that line 22 of Algorithm 3.2 is used to propagate the bitstrings of one's own node. These bitstrings are compared to the newly updated bitstrings at Stage 2 to check convergence.

Stage 2. Bitstrings of node i are updated by combining partial bitstrings of itself and nodes adjacent from i. For the purpose, the mapper is the Identity mapper (output the input without any modification). The reducer combines them, generates new bitstrings, and sets $flag$ by recording (a) whether at least a

Algorithm 3.2: HADI Stage 1

```
Input: edge data $E = \{(i, j)\}$, and
    current bitstring $B = \{(i, b(h-1, i))\}$ or Bitstring Creation Command $BC = \{(i, cmd)\}$.
Output: partial bitstring $B' = \{(i, b(h-1, j))\}$.
1: Stage1-Map(Key $k$, Value $v$):
2: if $(k, v)$ is of type B or BC then
3:     Output$(k, v)$;
4: else if $(k, v)$ is of type E then
5:     Output$(v, k)$;
6: end if
7:
8: Stage1-Reduce(Key $k$, Value $v[1..r]$):
9: SRC $\leftarrow$ [];
10: for $z \in v[1..r]$ do
11:     if $(k, z)$ is of type BC then
12:         $\hat{b}(h-1, k)$ $\leftarrow$NewFMBitstring();
13:     else if $(k, z)$ is of type B then
14:         $\hat{b}(h-1, k) \leftarrow z$;
15:     else if $(k, z)$ is of type E then
16:         Add $z$ to $SRC$;
17:     end if
18: end for
19: for $src \in SRC$ do
20:     Output$(src, \hat{b}(h-1, k))$;
21: end for
22: Output$(k, \hat{b}(h-1, k))$;
```

Algorithm 3.3: HADI Stage 2

```
Input: partial bitstring $B = \{(i, \hat{b}(h-1, j)\}$.
Output: full bitstring $B = \{(i, b(h, i)\}$.
Stage2-Map(Key $k$, Value $v$) : // Identity Mapper
Output($k, v$);

Stage2-Reduce(Key $k$, Value $v[1..r]$) :
$b(h, k) \leftarrow 0$;
for $z \in v[1..r]$ do
    $b(h, k) \leftarrow b(h, k)$ BIT-OR $z$;
end for
Update $flag$ of $b(h, k)$;
Output($k, b(h, k)$);
```

Algorithm 3.4: HADI Stage 3

```
Input: full bitstring $B = \{(i, b(h, i))\}$.
Output: number of changed nodes, and
    neighborhood $N(h)$.
Stage3-Map(Key $k$, Value $v$) :
Analyze $v$ to get ($changed$, $N(h, i)$);
Output($key\_for\_changed$,$changed$);
Output($key\_for\_neighborhood$, $N(h, i)$);

Stage3-Reduce(Key $k$, Value $v[1..r]$) :
$Changed \leftarrow 0$;
$N(h) \leftarrow 0$;
for $z \in v[1..r]$ do
    if $k$ is key_for_changed then
        $Changed \leftarrow Changed + z$;
    else if $k$ is key_for_neighborhood then
        $N(h) \leftarrow N(h) + z$;
    end if
end for
Output($key\_for\_changed$,$Changed$);
Output($key\_for\_neighborhood$, $N(h)$);
```

bitstring changed or not, and (b) the current iteration number h and the neighborhood value $N(h, i)$ (line 9 of Algorithm 3.3). This h and $N(h, i)$ are used to calculate the effective radius of nodes after all bitstrings converge, i.e., do not change. Notice that only the last neighborhood $N(h_{last}, i)$ and other neighborhoods $N(h', i)$ that satisfy $N(h', i) \geq 0.9 \cdot N(h_{last}, i)$ need to be saved to calculate the effective radius. The output of Stage 2 is fed into the input of Stage 1 at the next iteration.

Stage 3. We calculate the number of changed nodes and sum up the neighborhood value of all nodes to calculate $N(h)$. We use only two unique keys (key_for_changed and key_for_neighborhood), which correspond to the two calculated values. The analysis of line 2 of Algorithm 3.4 can be done by checking

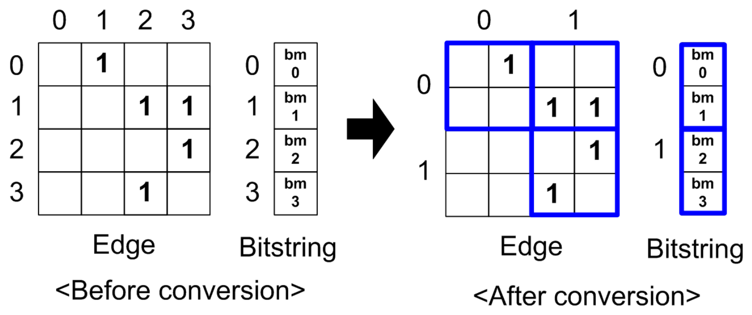


Figure 3.5: Converting the original edge and bitstring to blocks. The 4-by-4 edge and length-4 bitstring are converted to 2-by-2 super-elements and length-2 super-bitstrings. Notice the lower-left super-element of the edge is not produced since there is no nonzero element inside it.

the $flag$ field and using Equation (3.4) in Section 3.2. The variable $changed$ is set to 1 or 0, based on whether the bitmask of node k changed or not.

When all bitstrings of all nodes converged, a MAPREDUCE job to finalize the effective radius and diameter is performed and the program finishes. Compared to HADI-naive, the advantage of HADI-plain is clear: bitstrings and edges are evenly distributed over machines so that the algorithm can handle as much data as possible, given sufficiently many machines.

3.3.4 HADI-optimized in MAPREDUCE

HADI-optimized further improves HADI-plain. It uses two orthogonal ideas: "block operation" and "bit shuffle encoding". Both try to address some subtle performance issues. Specifically, HADOOP has the following two major bottlenecks:

- Materialization: at the end of each map/reduce stage, the output is written to the disk, and it is also read at the beginning of next reduce/map stage.
- Sorting: at the *Shuffle* stage, data is sent to each reducer and sorted before they are handed over to the *Reduce* stage.

HADI-optimized addresses these two issues.

Block Operation. Our first optimization is the block encoding of the edges and the bitstrings. The main idea is to group w by w sub-matrix into a super-element in the adjacency matrix E, and group w bitstrings into a super-bitstring. Now, HADI-plain is performed on these super-elements and super-bitstrings, instead of the original edges and bitstrings. Of course, appropriate decoding and encoding are necessary at each stage. Figure 3.5 shows an example of converting data into block-format.

By this block operation, the performance of HADI-plain changes as follows:

- *Input size* decreases in general, since we can use fewer bits to index elements inside a block.
- *Sorting time* decreases, since the number of elements to sort decreases.
- *Network traffic* decreases since the result of matching a super-element and a super-bitstring is a bitstring which can be at maximum $block\_width$ times smaller than that of HADI-plain.
- *Map and Reduce functions* take more time, since the block must be decoded to be processed, and be encoded back to block format.

For reasonable-size blocks, the performance gains (smaller input size, faster sorting time, less network traffic) outweigh the delays (more time to perform the map and reduce function). Also notice that the number of edge blocks depends on the community structure of the graph: if the adjacency matrix is nicely clustered, we will have fewer blocks. See Section 3.5, where we show results from block-structured graphs ('Kronecker graphs' [Leskovec et al., 2005]) and from random graphs ('Erdős-Rényi graphs' [Erdős and Rényi, 1959]).

Bit Shuffle Encoding. In our effort to decrease the input size, we propose an encoding scheme that can compress the bitstrings. Recall that in HADI-plain, we use K (e.g., 32, 64) bitstrings for each node, to increase the accuracy of our estimator. Since HADI requires $O(K(m+n)\log n)$ space, the amount of data increases when K is large. For example, the YahooWeb graph in Section 3.6 spans 120 GBytes (with 1.4 billion nodes, 6.6 billion edges). However the required disk space for just the bitstrings is $32 \cdot (1.4B + 6.6B) \cdot 8$ byte = 2 Tera bytes (assuming 8 byte for each bitstring), which is more than 16 times larger than the input graph.

The main idea of Bit Shuffle Encoding is to carefully reorder the bits of the bitstrings of each node, and then use Run Length Encoding. By construction, the leftmost part of each bitstring is almost full of one's, and the rest is almost full of zeros. Specifically, we make the reordered bit strings to contain long sequences of 1's and 0's: we get all the first bits from all K bitstrings, then get the second bits, and so on. As a result we get a single bit-sequence of length $K \cdot |bitstring|$, where most of the first bits are '1's, and most of the last bits are '0's. Then we encode only the length of each bit sequence, achieving good space savings (and, eventually, time savings, through fewer I/Os).

3.4 Analysis and Discussion

In this section, we analyze the time/space complexity of HADI and its possible implementation on RDBMS.

3.4.1 Time and Space Analysis

We analyze the time and the space complexities of HADI with s machines for a graph G with diameter d.

Lemma 3.1 (Time Complexity of HADI)**.** HADI *takes* $O(\frac{d(|V|+|E|)}{s}\log\frac{|V|+|E|}{s})$ *time.*

Proof. In all stages, HADI has O($|input\ size|/s$) running time. The Shuffle steps after Stage1 takes $O(\frac{|V|+|E|}{s}\log\frac{|V|+|E|}{s})$ time which dominates the time complexity. □

Notice that the time complexity of HADI is less than previous approaches in Section 3.7 ($O(|V|^2+|V||E|)$, at best). Similarly, for space we have:

Lemma 3.2 (Space Complexity of HADI)**.** HADI *requires* $O((|V|+|E|)\log|V|)$ *space.*

Proof. The maximum space $k \cdot ((|V|+|E|)\log|V|)$ is required at the output of `Stage1`-Reduce. Since k is a constant, the space complexity is $O((|V|+|E|)\log|V|)$. □

```
SELECT INTO B_NEW E.src, BIT-OR(B.b)
  FROM E, B
  WHERE E.dst=B.id
  GROUP BY E.src
```

Table 3.2: SQL statement for updating bitstrings in parallel RDMBS.

3.4.2 HADI in parallel DBMSs

Using relational database management systems (RDBMS) for graph mining is a promising research direction, especially given the findings of [Pavlo et al., 2009]. We mention that HADI can be implemented on the top of an Object-Relational DBMS (parallel or serial): it needs repeated joins of the edge table with the appropriate table of bit-strings, and a user-defined function for bit-OR-ing. We sketch a potential implementation of HADI in a RDBMS. The generalization of the HADI algorithm in a RDBMS will be discussed in Section 4.2.1.

Data. In parallel RDBMS implementations, data is saved in tables. The edges are saved in the table E with attributes src (source node id) and dst (destination node id). Similarly, the bitstrings are saved in the table B with id (node id) and b (bitstring) as its attributes.

Main Program Flow. The main flow comprises iterative execution of SQL statements with appropriate UDF (user defined function)s. The most important and expensive operation is updating the bitstrings of nodes. Observe that the operation can be concisely expressed by the following SQL statement:

The SQL statement requires BIT-OR(), a UDF function that implements the bit-OR-ing of the Flajolet-Martin bitstrings. The RDBMS implementation iteratively runs the SQL until B_NEW is same as B. B_NEW created at an iteration is used as B at the next iteration.

3.5 Scalability of HADI

In this section, we perform experiments to answer the following questions:

Q1 How fast is HADI?
Q2 How does it scale up with the graph size and the number of machines?
Q3 How do the optimizations help performance?

3.5.1 Experimental Setup

We use both real and synthetic graphs in Table 3.3 whose entries are repeated from Table 2.2 for convenience.

For the performance experiments, we use synthetic Kronecker (using a 3-nodes chain graph as an initiator) and Erdős-Rényi graphs. The reason of this choice is that we can generate any size of these two types of graphs, and Kronecker graphs mirror several real-world graph characteristics, including small and constant diameters, power-law degree distributions, etc. The number of nodes and edges of Erdős-Rényi graphs have been set to the same as those of the corresponding Kronecker graphs. The main difference of

Graph	Nodes	Edges	File Size	Type	Description
YahooWeb	1.4 B	6.6 B	0.12 TB	real	WWW links in 2002
LinkedIn	7.5 M	58 MB	1 GB	real	person-person in 2006
	4.4 M	27 MB	490 MB		person-person in 2005
	1.6 M	6.8 MB	121 MB		person-person in 2004
	85 K	230 KB	4 MB		person-person in 2003
Patents	6 M	16 M	264 MB	real	patent-patent
Kronecker	177 K	2 B	25 GB	synthetic	from Kronecker generator [Leskovec et al., 2005]
	121 K	1.1 B	13.9 GB		
	59 K	282 M	3.3 GB		
Erdős-Rényi	177 K	2 B	25 GB	synthetic	random $G_{n,p}$
	121 K	1.1 B	13.9 GB		
	59 K	282 M	3.3 GB		

Table 3.3: Datasets. B: Billion, M: Million, K: Thousand, TB: Terabytes, GB: Gigabytes, MB: Megabytes.

Kronecker compared to Erdős-Rényi graphs is the emergence of a block-wise structure of the adjacency matrix, from its construction [Leskovec et al., 2005]. We will see how this characteristic affects in the running time of our block-optimization in the next sections.

HADI runs on *M45*, one of the fifty most powerful supercomputers in the world. M45 has 480 hosts (each with 2 quad-core Intel Xeon 1.86 GHz, running RHEL5), with 3.5 Terabytes aggregate RAM, and over 1.5 Petabytes disk size.

Finally, we use the following notations to indicate different optimizations of HADI:

- HADI-BSE: HADI-plain with bit shuffle encoding.
- HADI-BL: HADI-plain with block operation.
- HADI-OPT: HADI-plain with both bit shuffle encoding and block operation.

3.5.2 Running Time and Scale-up

Figure 3.6 gives the wall-clock time of HADI-OPT versus the number of edges in the graph. Each curve corresponds to a different number of machines used (from 10 to 90). HADI has excellent scalability, with its running time being linear on the number of edges. The rest of the HADI versions (HADI-plain, HADI-BL, and HADI-BSE), were slower, but had a similar, linear trend, and they are omitted to avoid clutter.

Figure 3.7 gives the throughput $1/T_M$ of HADI-OPT. We also tried HADI with one machine; however it did not complete, since the machine would take so long that it would often fail in the meanwhile. For this reason, we do not report the typical scale-up score $s = T_1/T_M$ (ratio of time with 1 machine, over time with M machine), and instead we report just the inverse of T_M. HADI scales up near-linearly with the number of machines M, close to the ideal scale-up.

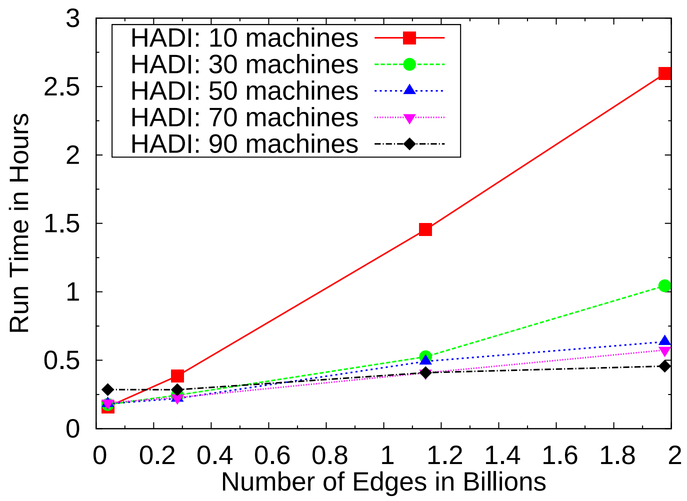


Figure 3.6: Running time versus number of edges with HADI-OPT on Kronecker graphs for three iterations. Notice the excellent scalability: linear on the graph size (number of edges).

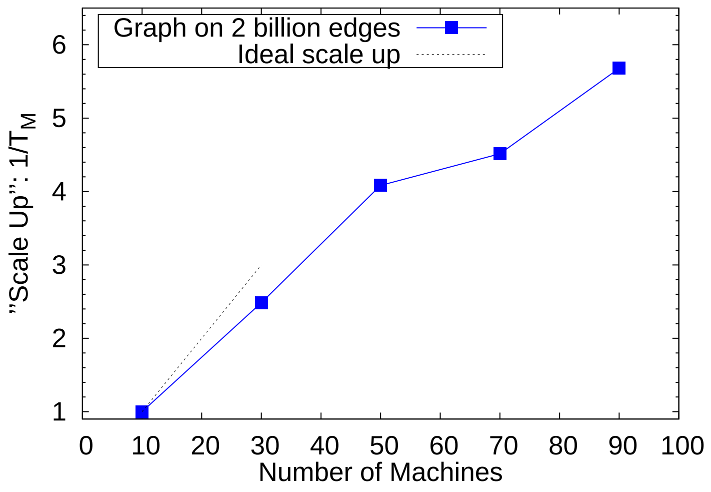


Figure 3.7: "Scale-up" (throughput $1/T_M$) versus number of machines M, for the Kronecker graph (2B edges). Notice the near-linear growth in the beginning, close to the ideal (dotted line).

3.5.3 Effect of Optimizations

Among the optimizations that we mentioned earlier, which one helps the most, and by how much? Figure 3.1 (c) plots the running time of different graphs versus different HADI optimizations. For the Kronecker graphs, we see that block operation is more efficient than bit shuffle encoding. Here, HADI-OPT achieves **7.6**× better performance than HADI-plain. For the Erdős-Rényi graphs, however, we see that block operations do not help more than bit shuffle encoding, because the adjacency matrix has no block structure, while Kronecker graphs do. Also notice that HADI-BLK and HADI-OPT run faster on Kronecker graphs than on Erdős-Rényi graphs of the same sizes. Again, the reason is that Kronecker graphs have fewer nonzero blocks (i.e., "communities") by their construction, and the "block" operation yields more savings.

3.6 HADI At Work

HADI reveals new patterns in massive graphs which we present in this section.

3.6.1 Static Patterns

Diameter

What is the diameter of the Web? Albert et al. [Albert et al., 1999] computed the diameter on a directed Web graph with ≈ 0.3 million nodes, and conjectured that it is around 19 for the 1.4 billion-node Web as shown in the upper line of Figure 3.8. Broder et al. [Broder et al., 2000] used sampling from ≈ 200 million-nodes Web and reported 16.15 and 6.83 as the diameter for the directed and the undirected cases, respectively. What should be the effective diameter, for a significantly larger crawl of the Web, with billions of nodes? Figure 3.1 gives the surprising answer:
Observation 1 (Small Web)**.** *The effective diameter of the YahooWeb graph (year: 2002) is surprisingly small ($\approx 7 \sim 8$).*

The previous results from Albert et al. and Broder et al. are based on the average diameter. For the reason, we also computed the average diameter and show the comparison of diameters of different graphs in Figure 3.8. We first observe that the average diameters of all graphs are relatively small (< 20) for both the directed and the undirected cases. We also observe that the Albert et al.'s conjecture of the diameter of the directed graph is over-pessimistic: both the sampling and HADI reported smaller values for the diameter of the directed graph. For the diameter of the undirected graph, we observe the constant or shrinking diameter pattern [Leskovec et al., 2007].

Shape of Distribution

The next question is, how are the radii distributed in real networks? Is it Poisson? Lognormal? Figure 3.1 gives the surprising answer: multimodal! In other relatively small networks, however, we observed bi-modal structures. As shown in the Radius plots of U.S. Patent and LinkedIn networks in Figure 3.9, they have a peak at zero, a dip at a small radius value (9, and 4, respectively) and another peak very close to the dip. Given the prevalence of the bi-modal shape, our conjecture is that the multi-modal shape of

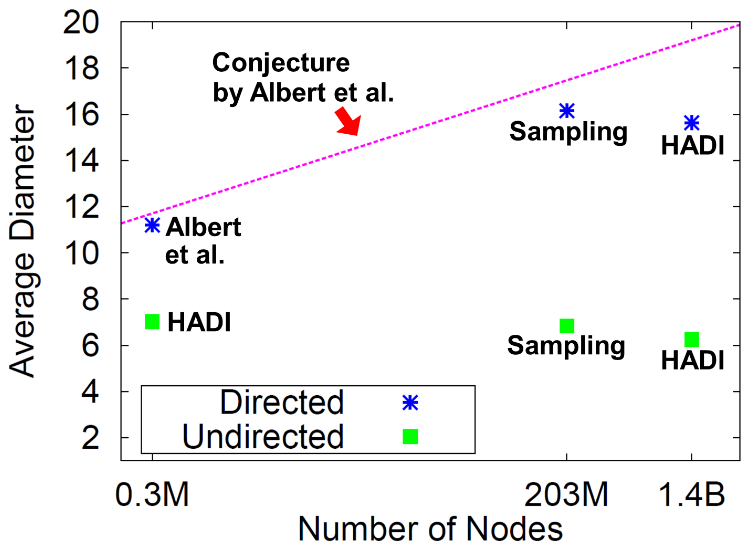


Figure 3.8: Average diameter vs. number of nodes in lin-log scale for the three different Web graphs, where M and B represent millions and billions, respectively. **(0.3M):** Web pages inside nd.edu at 1999, from Albert et al.'s work. **(203M):** Web pages crawled by Altavista at 1999, from Broder et al.'s work **(1.4B):** Web pages crawled by Yahoo at 2002 (YahooWeb in Table 3.3). The annotations (Albert et al., Sampling, and HADI) near the points represent the algorithms for computing the diameter. The Albert et al.'s algorithm seems to be an exact breadth first search, although not clearly specified in their paper. Notice the relatively small diameters for both the directed and the undirected cases. Also notice that the diameters of the undirected Web graphs remain near-constant.

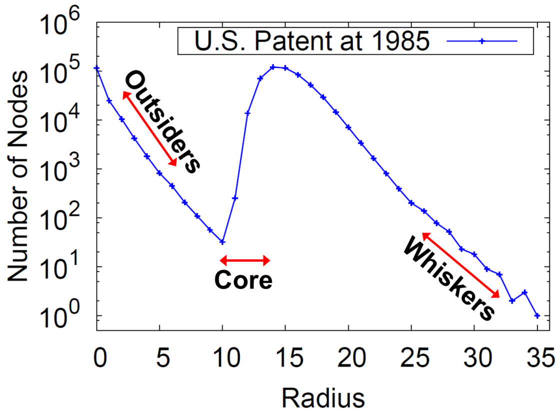
YahooWeb is possibly due to a mixture of relatively smaller sub-graphs, which got loosely connected recently.

Observation 2 (Multi-modal and Bi-modal)**.** *The Radius distribution of the Web graph has a multi-modal structure. Many smaller networks have the bi-modal structure.*

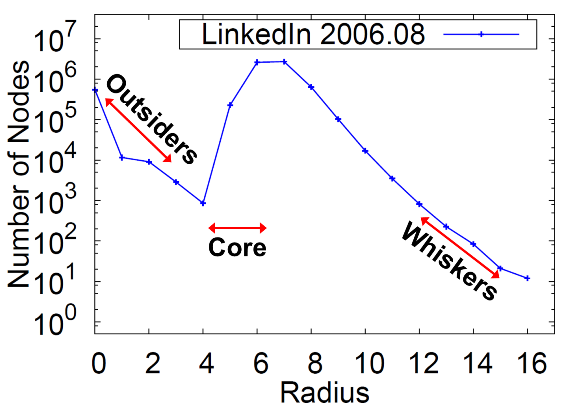
About the bi-modal structure, a natural question to ask is what are the common properties of the nodes that belong to the first peak; similarly, for the nodes in the first dip, and the same for the nodes of the second peak. After investigation, the former are nodes that belong to the disconnected components (DCs); nodes in the dip are usually core nodes in the giant connected component (GCC), and the nodes at the second peak are the vast majority of well connected nodes in the GCC. Figure 3.10 exactly shows the radii distribution for the nodes of the GCC (in blue), and the nodes of the few largest remaining components.

In Figure 3.10, we clearly see that the second peak of the bimodal structure came from the giant connected component. But, where does the first peak around radius 0 come from? We can get the answer from the distribution of connected component of the same graph in Figure 3.11. Since the ranges of radius are limited by the size of connected components, we see the first peak of Radius plot came from the disconnected components whose size follows a power law.

Now we can explain the three important areas of Figure 3.9: '*outsiders*' are the nodes in the disconnected components, and responsible for the first peak and the negative slope to the dip. '*Core*' are the central nodes with the smallest radii from the giant connected component. '*Whiskers*' [Leskovec et al., 2008] are the nodes connected to the GCC with long paths (resembling a whisker), and are the reasons of the second negative slope.



(a) U.S. Patent



(b) LinkedIn

Figure 3.9: Static Radius Plot (Count versus Radius) of U.S. Patent and LinkedIn graphs. Notice the bi-modal structure with 'outsiders' (nodes in the DCs), 'core' (central nodes in the GCC), and 'whiskers' (nodes connected to the GCC with long paths).

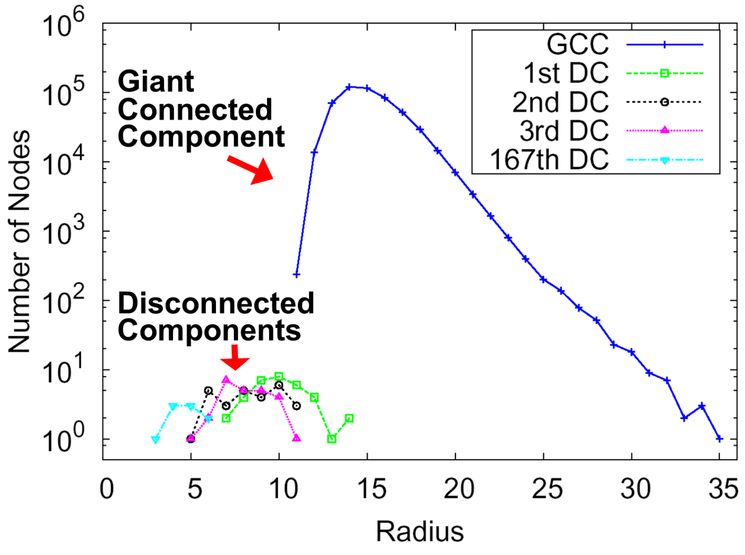
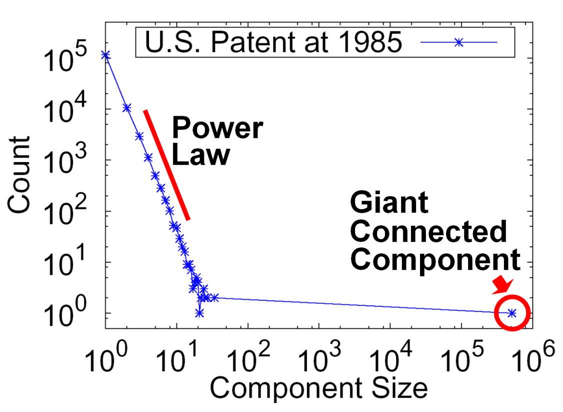
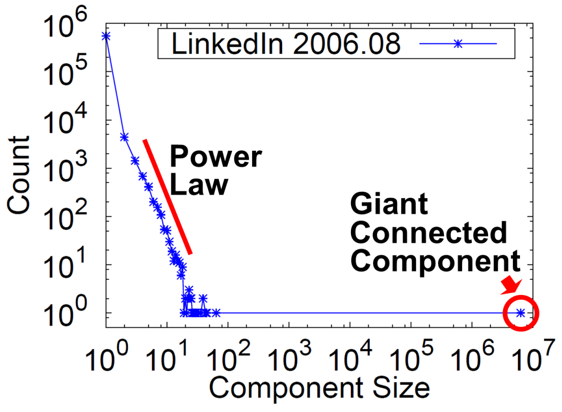


Figure 3.10: Radius plot (Count versus radius) for several connected components of the U.S. Patent data in 1985. In blue: the distribution for the GCC (Giant Connected Component); rest colors: several DC (Disconnected Component)s.



(a) Patent



(b) LinkedIn

Figure 3.11: Size distribution of connected components. Notice the size of the disconnected components (DCs) follows a power-law which explains the first peak around radius 0 of the radius plots in Figure 3.9.

Radius plot of GCC

Figure 3.1 (b) shows a striking pattern: all nodes of the GCC of the YahooWeb graph have radius 6 or more, except for 1 node (only!). Inspection shows that this is `google.com`. We were surprised, because we would expect a few more popular nodes to be in the same situation (eg., Yahoo, eBay, Amazon).

"Core" and "Whisker" nodes

The next question is, what can we say about the connectivity of the *core* nodes, and the *whisker* nodes? For example, is it true that the highest degree nodes are the most central ones (i.e. minimum radius)? The answer is given by the "Radius-Degree" plot in Figure 3.12: this is a scatter-plot, with one dot for every node, plotting the degree of the node versus its radius. We also color-coded the nodes of the GCC (in blue), while the rest are in magenta.

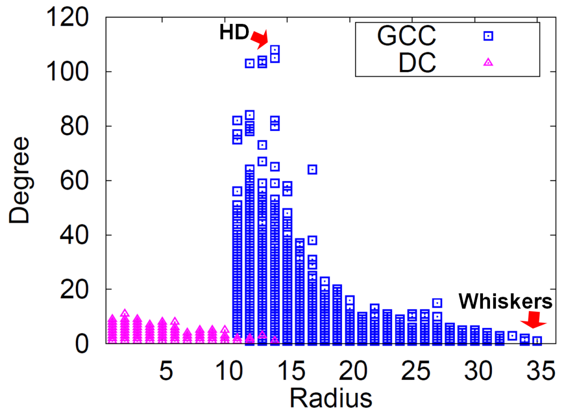
Observation 3 (High degree nodes)**.** *High degree nodes have relatively small radii. The highest degree node (a) belongs to the core nodes inside the GCC but (b) is* not *necessarily the one with the smallest radius.*

Observation 4 (Whisker nodes)**.** Whisker *nodes have small degree, that is, they belong to chains (as opposed to more complicated shapes).*

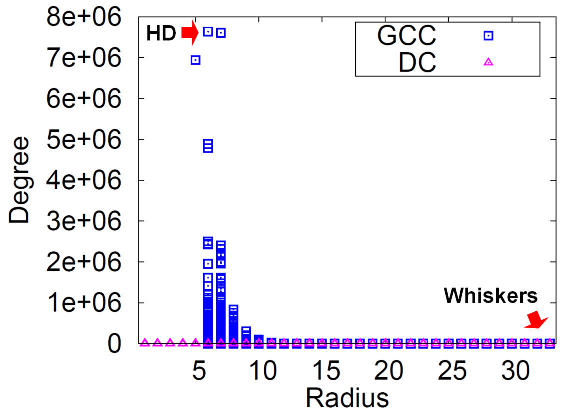
Radius plots of anomalous DCs

Observation 5 (Anomalous Radius Plot)**.** *The radius plots of some of the largest disconnected components of YahooWeb graph show anomalous radius distributions as opposed to the bi-modal distribution.*

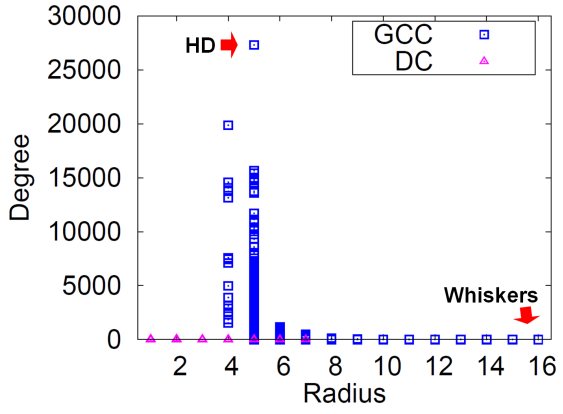
The graph in Figure 3.2 (a) is the largest disconnected component which has a near-bipartite-core structure. The component is very likely to be a link farm since almost all the nodes are connected to the three nodes at the bottom center with the effective radius 1. Similarly, we observed many star-shaped disconnected



(a) Patent

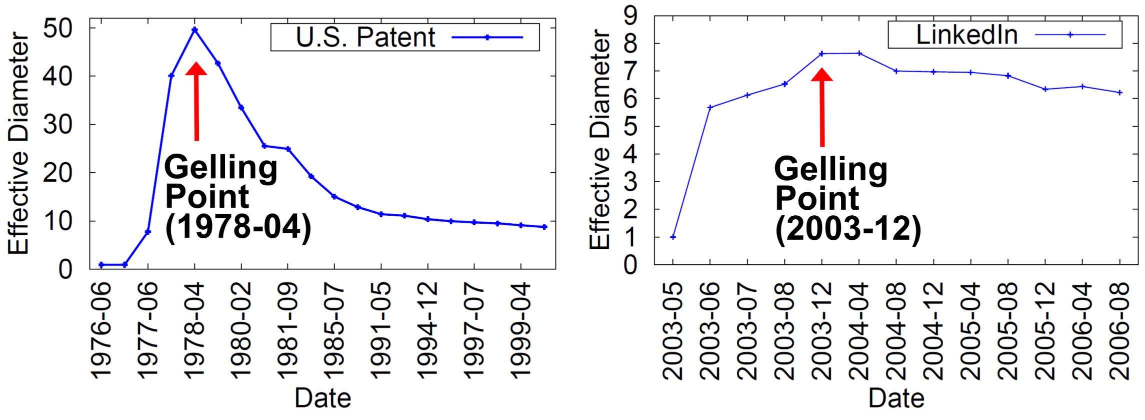


(b) YahooWeb



(c) LinkedIn

Figure 3.12: Radius-Degree plots of real-world graphs. HD represents the node with the highest degree. Notice that HD belongs to core nodes inside the GCC, and whiskers have small degree.



(a) Patent (b) LinkedIn

Figure 3.13: Evolution of the effective diameter of real graphs. The diameter increases until a 'gelling' point, and starts to decrease after the point.

components as shown in Figure 3.2 (c). This is also a strong candidate for a link farm, where a node with the effective radius 1 is connected to all the other nodes with the effective radius 2.

3.6.2 Temporal Patterns

Here we study how the radius distribution changes over time. We know that the diameter of a graph typically grows with time, spikes at the 'gelling point' (the point when a GCC emerges), and then shrinks [Mcglohon et al., 2008, Leskovec et al., 2007]. Indeed, this holds for our datasets as shown in Figure 3.13.

The question is, how does the radius distribution change over time? Does it still have the bi-modal pattern? Do the peaks and slopes change over time? We show the answer in Figure 3.14 and Observation 6.
Observation 6 (Expansion-Contraction)**.** *The radius distribution expands to the right until it reaches the gelling point. Then, it contracts to the left.*

Another striking observation is that the decreasing segments seem to be well fit by a line, in log-lin axis, thus indicating an exponential decay.
Observation 7 (Exponential Decays)**.** *The decreasing segments of several, real radius plots seem to decay exponentially, that is*

$$count(r) \propto \exp(-cr) \tag{3.5}$$

for every time tick after *the gelling point.* $count(r)$ *is the number of nodes with radius* r*, and* c *is a constant.*

For the Patent and LinkedIn graphs, the correlation coefficient was excellent (typically, -0.98 or better).

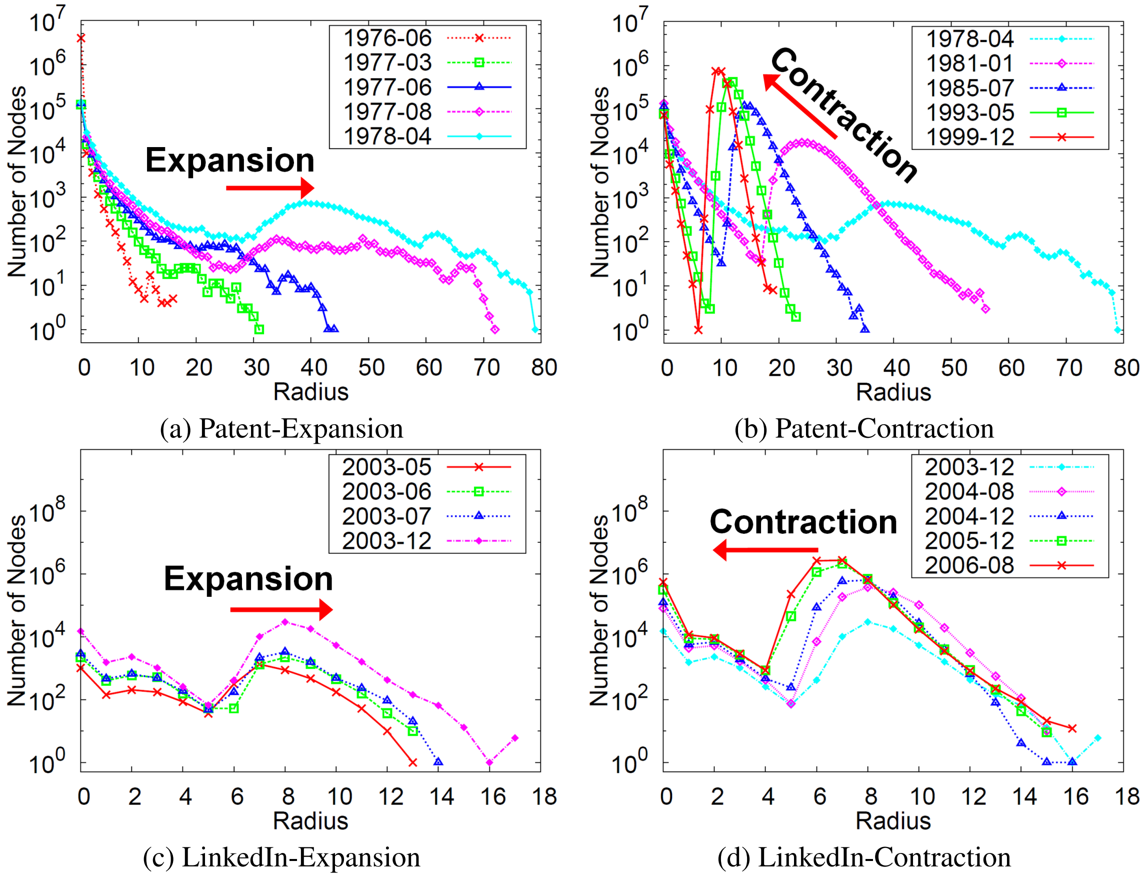


Figure 3.14: Radius distribution over time. "Expansion": the radius distribution moves to the right until the gelling point. "Contraction": the radius distribution moves to the left after the gelling point.

3.7 Background

We briefly review related works on algorithms for radius and diameter computation. The typical algorithms to compute the radius and the diameter of a graph include Breadth First Search (BFS) and Floyd's algorithm [Cormen et al., 1990]. Both approaches are prohibitively slow for large graphs, requiring $O(|V|^2 + |V||E|)$ and $O(|V|^3)$ time. For the same reason, related BFS or all-pair shortest-path based algorithms like [Ferrez et al., 1998, Bader and Madduri, 2008, Ma and Ma, 1993, Sinha et al., 1986] can not handle large graphs. A sampling approach starts BFS from a subset of nodes, typically chosen at random as in [Broder et al., 2000]. Despite its practicality, this approach has no obvious solution for choosing the representative sample for BFS.

3.8 Conclusion

In this chapter, we design HADI, an algorithm for computing radii and diameter of Tera-byte scale graphs, and analyze large networks to observe important patterns. The contributions of this chapter are the following:

- **Design.** We develop HADI, a scalable MAPREDUCE algorithm for diameter and radius estimation, on massive graphs.
- **Optimization.** Careful fine-tunings on HADI, leading to up to *7.6*$\times$ faster computation, linear scalability on the size of the graph (number of edges) and near-linear speed-up on the number of machines. The experiments run on the M45 HADOOP cluster of Yahoo, one of the 50 largest supercomputers in the world.
- **Observations.** Thanks to HADI, we study the diameter and radii distribution of one of the largest public Web graphs ever analyzed (over 6 *billion* edges); we also observe the "Small Web" phenomenon, multi-modal/bi-modal radius distributions, and palindrome motions of radius distributions over time in real networks.

Chapter 4

Generalized Iterative Matrix-Vector Multiplication

In this chapter, we generalize the HADI operation (Chapter 3) to propose a graph mining primitive called Generalized Iterated Matrix-Vector multiplication (GIM-V) which unifies many graph mining operations including PageRank, diameter estimation, connected components, and etc. GIM-V is highly optimized, achieving (a) good scale-up on the number of available machines, (b) linear running time on the number of edges, and (c) more than *5 times* faster performance over the non-optimized version of GIM-V.

We run experiments on M45, and report our findings on several real graphs, including one of the largest publicly available Web graphs, thanks to Yahoo!, with $\approx$ 6,7 billion edges.

4.1 Introduction

How can we analyze the structure (e.g. connected components, PageRank, etc.) of very large graphs with billions of nodes and edges? How do we unify many different graph mining algorithms? In this chapter we introduce a general primitive called Generalized Iterated Matrix-Vector multiplication (GIM-V) which unifies many different graph mining operations, and use it to analyze real world graphs. The contributions are the following:

1. Unification of seemingly different graph mining tasks, via a generalization of matrix-vector multiplication (GIM-V).
2. The careful implementation of GIM-V, with several optimizations, and several graph mining operations (PageRank, Random Walk with Restart (RWR), diameter estimation, and connected components). Moreover, the method is linear on the number of edges, and scales up well with the number of available machines.
3. Performance analysis, pinpointing the most successful combination of optimizations, which lead to up to *5 times* better speed than naive implementation.
4. Analysis of large, real graphs, including one of the largest publicly available graph that was ever analyzed, Yahoo's Web graph.

The rest of the chapter is organized as follows. Section 4.2 describes our framework and explains several graph mining algorithms. Section 4.3 discusses optimizations that allow us to achieve significantly faster

Symbol	Definition
M	a matrix
v	a vector
A	adjacency matrix of a graph
D	diagonal matrix induced from A such that $D_{ii} = \sum_j A_{ij}$

Table 4.1: Table of symbols.

performance in practice. In Section 4.4 we present timing results and Section 4.5 our findings in real world, large scale graphs. After presenting the related work in Section 4.6, we conclude in Section 4.7.

Table 4.1 lists the symbols frequently used in this chapter.

4.2 Proposed Method

How can we quickly find connected components, diameter, PageRank, node proximities of very large graphs? We show that, even if they seem unrelated, eventually we can unify them using the GIM-V primitive, standing for Generalized Iterative Matrix-Vector multiplication, which we describe in the next.

4.2.1 Main Idea

GIM-V, or 'Generalized Iterative Matrix-Vector multiplication' is a generalization of normal matrix-vector multiplication. Suppose we have a n by n matrix M and a vector v of size n. Let $m_{i,j}$ denote the (i, j)-th element of M. Then the usual matrix-vector multiplication is

$M \times v = v'$ where $v'_i = \sum_{j=1}^{n} m_{i,j} v_j$.

There are three operations in the previous formula, which, if customized separately, will give a surprising number of useful graph mining algorithms:

1. `combine2`: multiply $m_{i,j}$ and v_j.
2. `combineAll`$_i$: sum n multiplication results for node i.
3. `assign`: overwrite the previous value of v_i with the new result to make v'_i.

In GIM-V, let us define the operator $\times_G$, where the three operations can be defined arbitrarily. Formally, we have:

$v' = M \times_G v$
where v'_i = `assign`(v_i,`combineAll`$_i$($\{x_j \mid j = 1..n$, and $x_j =$`combine2`$(m_{i,j}, v_j)\}$)).

The functions `combine2`(), `combineAll`$_i$(), and `assign`() have the following signatures (generalizing the product, the sum and the assignment, respectively, that the traditional matrix-vector multiplication requires):

1. `combine2`$(m_{i,j}, v_j)$: combine $m_{i,j}$ and v_j.
2. `combineAll`$_i(x_1, ..., x_n)$: combine all the results from `combine2`() for node i.
3. `assign`(v_i, v_{new}) : decide how to update v_i with v_{new}.

```
SELECT E.sid, combineAll_{E.sid}(combine2(E.val,V.val))
  FROM E, V
  WHERE E.did=V.id
  GROUP BY E.sid
```

Table 4.2: GIM-V in terms of SQL.

The 'Iterative' in the name of GIM-V denotes that we apply the $\times_G$ operation until an algorithm-specific convergence criterion is met. As we will see in a moment, by customizing these operations, we can obtain different, useful algorithms including PageRank, Random Walk with Restart, connected components, and diameter estimation. But first we want to highlight the strong connection of GIM-V with SQL: when `combineAll`$_i$() and `assign`() can be implemented by user defined functions, the operator $\times_G$ can be expressed concisely in terms of SQL. This viewpoint is important when we implement GIM-V in large scale parallel processing platforms, including HADOOP, if they can be customized to support several SQL primitives including JOIN and GROUP BY. Suppose we have an `edge` table `E(sid, did, val)` and a `vector` table `V(id, val)`, corresponding to a matrix and a vector, respectively. Then, $\times_G$ corresponds to the SQL statement in Table 4.2. We assume that we have (built-in or user-defined) functions, `combineAll`$_i$() and `combine2`(), and we also assume that the resulting table/vector will be fed into the `assign`() function (omitted, for clarity).

In the following sections we show how we can customize GIM-V, to handle important graph mining operations including PageRank, Random Walk with Restart, diameter estimation, and connected components.

4.2.2 GIM-V and PageRank

Our first application of GIM-V is PageRank, a famous algorithm used by Google to calculate relative importance of Web pages [Brin and Page, 1998]. The PageRank vector p of n Web pages satisfies the following eigenvector equation:

$$p = (cA^TD^{-1} + (1-c)U)p\,,$$

where c is a damping factor (usually set to 0.85), A^T is the transpose of the n by n adjacency matrix A, D is the diagonal matrix whose ith element D_{ii} is $\sum_j A_{ij}$, and U is a n by n matrix with all elements set to $1/n$.

To calculate the eigenvector p we can use the "power method", which multiplies an initial vector with the matrix, several times. We initialize the current PageRank vector p^{cur} and set all its elements to $1/n$. Then the next PageRank vector p^{next} is calculated by $p^{next} = (cA^TD^{-1} + (1-c)U)p^{cur}$. We continue to do the multiplication until p converges.

PageRank is a direct application of GIM-V. In this view, we first construct a matrix $M = A^TD^{-1}$, which is the column-normalized adjacency matrix. Then the next PageRank is calculated by $p^{next} = M \times_G p^{cur}$ where the three operations are defined as follows:

1. `combine2`$(m_{i,j}, v_j) = c \times m_{i,j} \times v_j$.
2. `combineAll`$_i(x_1, ..., x_n) = \frac{(1-c)}{n} + \sum_{j=1}^{n} x_j$.
3. `assign`$(v_i, v_{new}) = v_{new}$.

4.2.3 GIM-V and Random Walk with Restart

Random Walk with Restart (RWR) is an algorithm to measure the proximity of nodes in graph [Pan et al., 2004]. In RWR, the proximity vector r_k from node k indicates how close other nodes are to node k. It satisfies the equation:

$$r_k = cA^T D^{-1} r_k + (1-c)e_k \ ,$$

where e_k is a n-vector whose k^{th} element is 1, and every other elements are 0. c is a restart probability parameter which is typically set to 0.85 [Pan et al., 2004], A^T is the transpose of the n by n adjacency matrix A, and D is the diagonal matrix whose ith element D_{ii} is $\sum_j A_{ij}$, as in Section 4.2.2. In GIM-V, RWR is formulated by $r_k^{next} = M \times_G r_k^{cur}$, where $M = A^T D^{-1}$ is the column-normalized adjacency matrix. The three operations are defined as follows (δ_{ik} is the *Kronecker delta*, equal to 1 if $i = k$ and 0 otherwise):

1. `combine2`$(m_{i,j}, v_j) = c \times m_{i,j} \times v_j$.
2. `combineAll`$_i(x_1, ..., x_n) = (1-c)\delta_{ik} + \sum_{j=1}^{n} x_j$.
3. `assign`$(v_i, v_{new}) = v_{new}$.

4.2.4 GIM-V and Diameter Estimation

HADI, which is described in Chapter 3, is an algorithm to estimate the diameter and radius of large graphs. The main operation of HADI is updating the number of neighbors as h increases. Specifically, the number of neighbors within hop h reachable from node v_i is encoded in a probabilistic bitstring b_i^h which is updated as follows:

$$b_i^{h+1} = b_i^h \text{ BITWISE-OR } \{b_k^h \mid (i,k) \in E\},$$

where E is the set of edges. In GIM-V, the bitstring update of HADI is represented by

$$b^{h+1} = M \times_G b^h \ ,$$

where M is the adjacency matrix, b^{h+1} is a vector of length n which is updated by
b_i^{h+1} =`assign`$(b_i^h$,`combineAll`$_i(\{x_j \mid j = 1..n$, and x_j =`combine2`$(m_{i,j}, b_j^h)\}))$,
and the three operations are defined as follows:

1. `combine2`$(m_{i,j}, v_j) = m_{i,j} \times v_j$.
2. `combineAll`$_i(x_1, ..., x_n)$ = BITWISE-OR$\{x_j \mid j = 1..n\}$.
3. `assign`(v_i, v_{new}) = BITWISE-OR(v_i, v_{new}).

The $\times_G$ operation is run iteratively until the bitstring for all the nodes do not change.

4.2.5 GIM-V and Connected Components

We propose HCC, a new algorithm for finding connected components in large graphs. Like HADI, HCC is an application of GIM-V with custom functions. The main idea is as follows. For every node v_i in the graph, we maintain a component id c_i^h which is the minimum node id within h hops from v_i. Initially, c_i^h of v_i is set to its own node id: that is, $c_i^0 = i$. For each iteration, each node sends its current c_i^h to its neighbors. Then c_i^{h+1}, component id of v_i at the next step, is set to the minimum value among its current component id and the received component ids from its neighbors. The crucial observation is that this communication between neighbors can be formulated in GIM-V as follows:

$$c^{h+1} = M \times_G c^h \ ,$$

where M is the adjacency matrix, c^{h+1} is a vector of length n which is updated by
c_i^{h+1} =`assign`(c_i^h,`combineAll`$_i$($\{x_j \mid j = 1..n$, and x_j =`combine2`$(m_{i,j}, c_j^h)\}$)),
and the three operations are defined as follows:

1. `combine2`$(m_{i,j}, v_j)$ = $m_{i,j} \times v_j$.
2. `combineAll`$_i(x_1, ..., x_n)$ = MIN$\{x_j \mid j = 1..n\}$.
3. `assign`(v_i, v_{new}) = MIN(v_i, v_{new}).

By repeating this process, component ids of nodes in a component are set to the minimum node id of the component. We iteratively do the multiplication until component ids converge. The upper bound of the number of iterations in HCC are determined by the following theorem.
Theorem 4.1 (Upper bound of iterations in HCC)**.** HCC *requires maximum d iterations where d is the diameter of the graph.*

Proof. The minimum node id is propagated to its neighbors at most d times. □

Since the diameter of real graphs are relatively small, HCC completes after small number of iterations.

4.3 Fast Algorithms for GIM-V

How can we parallelize the algorithm presented in the previous section? In this section, we first describe naive HADOOP algorithms for GIM-V. Then we propose several faster methods for GIM-V.

4.3.1 GIM-V BASE: Naive Multiplication

GIM-V BASE is a two-stage algorithm whose pseudo code is shown in Algorithm 4.1 and 4.2. The inputs are an edge file and a vector file. Each line of the edge file contains one $(id_{src}, id_{dst}, mval)$ which corresponds to a non-zero cell in the adjacency matrix M. Similarly, each line of the vector file contains one $(id, vval)$ which corresponds to an element in the vector V. `Stage1` performs `combine2` operation by combining columns of matrix (id_{dst} of M) with rows of vector (id of V). The output of `Stage1` are (key, value) pairs where key is the source node id of the matrix (id_{src} of M) and the value is the partially combined result (`combine2`$(mval, vval)$). This output of `Stage1` becomes the input of `Stage2`. `Stage2` combines all partial results from `Stage1` and assigns the new vector to the old vector. The `combineAll`$_i$() and `assign`() operations are done in line 15 of `Stage2`, where the "self" and "others" tags in line 15 and line 21 of `Stage1` are used to make v_i and v_{new} of GIM-V, respectively.

This two-stage algorithm is run iteratively until application-specific convergence criterion is met. In Algorithm 4.1 and 4.2, Output(k, v) means to output data with the key k and the value v.

Algorithm 4.1: GIM-V BASE Stage 1.

```
Input: matrix $M = \{(id_{src}, (id_{dst}, mval))\}$, and
    vector $x = \{(id, vval)\}$.
Output: partial vector $x' = \{(id_{src}, \texttt{combine2}(mval, vval)\}$.
Stage1-Map(Key $k$, Value $v$):
if $(k, v)$ is of type $x$ then
    Output($k, v$); // ($k$: $id$, $v$: $vval$)
else if $(k, v)$ is of type M then
    $(id_{dst}, mval) \leftarrow v$;
    Output($id_{dst}, (k, mval)$); // ($k$: $id_{src}$)
end if

Stage1-Reduce(Key $k$, Value $v[1..r]$):
$saved\_kv \leftarrow [\ ]$;
$saved\_v \leftarrow [\ ]$;
for $z \in v[1..r]$ do
    if $(k, z)$ is of type $x$ then
        $saved\_v \leftarrow z$;
        Output($k$, ("$self$", $saved\_v$));
    else if $(k, z)$ is of type M then
        Add $z$ to $saved\_kv$; // ($z$: $(id_{src}, mval)$)
    end if
end for
for $(id'_{src}, mval') \in saved\_kv$ do
    Output($id'_{src}$, ("$others$",combine2($mval', saved\_v$)));
end for
```

4.3.2 GIM-V BL: Block Multiplication

GIM-V BL is a fast algorithm for GIM-V which is based on block multiplication. The main idea is to group elements of the input matrix into blocks or submatrices of size b by b. Also we group elements of input vectors into blocks of length b. Here the grouping means we put all the elements in a group into one line of input file. Each block contains only non-zero elements of the matrix or vector. The format of a matrix block with k nonzero elements is ($row_{block}, col_{block}, row_{elem_1}$, $col_{elem_1}, mval_{elem_1}, ...,$ $row_{elem_k}, col_{elem_k}, mval_{elem_k}$). Similarly, the format of a vector block with k nonzero elements is ($id_{block}, id_{elem_1}, vval_{elem_1}, ..., id_{elem_k}$, $vval_{elem_k}$). Only blocks with at least one nonzero elements are saved to disk. This block encoding forces nearby edges in the adjacency matrix to be closely located; it is different from HADOOP's default behavior which do not guarantee co-locating them. After grouping, GIM-V is performed on blocks, not on individual elements. GIM-V BL is illustrated in Figure 4.1.

In our experiment at Section 4.4, GIM-V BL is more than 5 times faster than GIM-V BASE. There are two main reasons for this speed-up.

- **Sorting Time.** Block encoding decreases the number of items to sort in the shuffling stage of HADOOP. We observed that one of the main bottleneck of programs in HADOOP is its shuffling stage where network transfer, sorting, and disk I/O happens. By encoding to blocks of width b, the

Algorithm 4.2: GIM-V BASE Stage 2.

```
Input: partial vector $x' = \{(id_{src}, vval')\}$.
Output: result vector $x = \{(id_{src}, vval)\}$.
1: Stage2-Map(Key $k$, Value $v$):
2: Output($k, v$);
3:
4: Stage2-Reduce(Key $k$, Value $v[1..r]$):
5: $others\_v \leftarrow$[ ];
6: $self\_v \leftarrow$[ ];
7: for $z \in v[1..r]$ do
8:     $(tag, v') \leftarrow z$;
9:     if $tag$ = "same" then
10:        $self\_v \leftarrow v'$;
11:    else if $tag$ = "others" then
12:        Add $v'$ to $others\_v$;
13:    end if
14: end for
15: Output($k$,assign($self\_v$,combineAll$_k$($others\_v$)));
```

number of lines in the matrix and the vector file decreases to $1/b^2$ and $1/b$ times of their original size, respectively for full matrices and vectors.

- **Compression.** The size of the data decreases significantly by converting edges and vectors to block format. The reason is that in GIM-V BASE we need 4×2 bytes to save each (srcid, dstid) pair since we need 4 bytes to save a node id using an Integer. However in GIM-V BL we can specify each *block* using a block row id and a block column id with two 4-byte Integers, and refer to elements inside the block using $2 \times \log b$ bits. This is possible because we can use $\log b$ bits to refer to a row or column inside a block. By this block method we decreased the edge file size (e.g., more than 50% for YahooWeb graph in Section 4.4).

4.3.3 GIM-V CL: Clustered Edges

When we use block multiplication, another advantage is that we can benefit from clustered edges. As shown in Figure 4.2, we can use smaller number of blocks if input edge files are clustered. Clustered edges can be built if we can use heuristics in data preprocessing stage so that edges are clustered, or by co-clustering (e.g., see [Papadimitriou and Sun, 2008]). The preprocessing for edge clustering need to be done only once; however, they can be used by every iteration of various application of GIM-V. So we have two variants of GIM-V: GIM-V CL, which is GIM-V BASE with clustered edges, and GIM-V BL-CL, which is GIM-V BL with clustered edges. Be aware that clustered edges are useful only when combined with block encoding. If every element is treated separately, then clustered edges do not help anything for the performance of GIM-V.

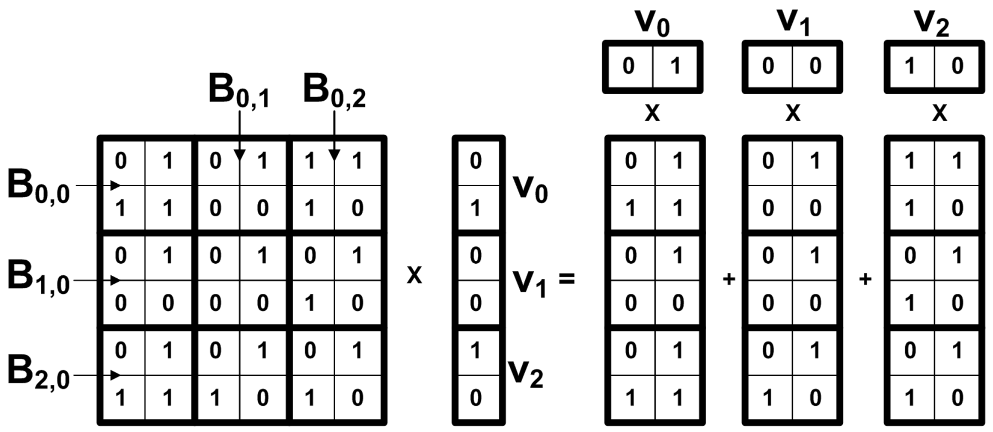


Figure 4.1: GIM-V BL using 2 x 2 blocks. $B_{i,j}$ represents a matrix block, and v_i represents a vector block. The matrix and the vector are joined block-wise, not element-wise.

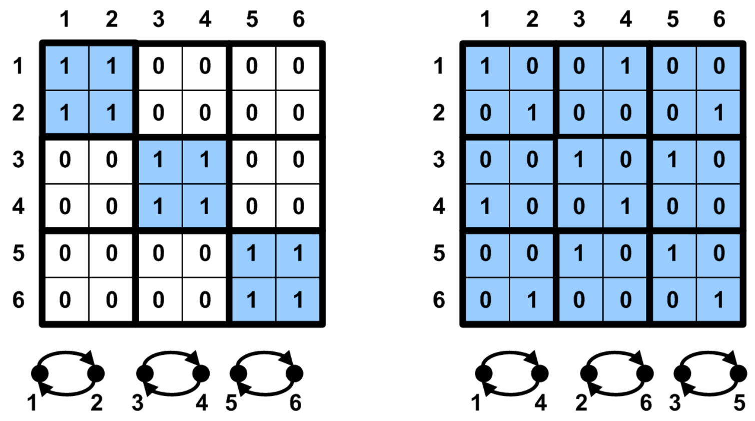


Figure 4.2: Clustered vs. non-clustered adjacency matrices for two isomorphic graphs. The edges are grouped into 2 by 2 blocks. The left graph uses only 3 blocks while the right graph uses 9 blocks.

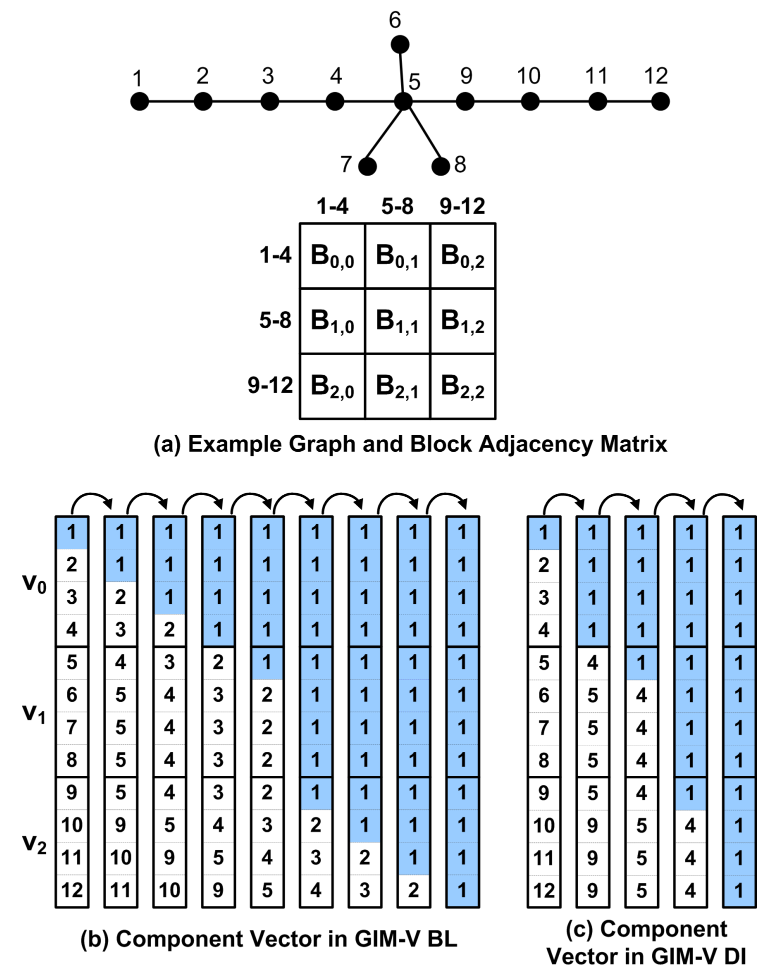

1-4
5-8
9-12
$B_{0,0}$ $B_{0,1}$ $B_{0,2}$
$B_{1,0}$ $B_{1,1}$ $B_{1,2}$
$B_{2,0}$ $B_{2,1}$ $B_{2,2}$
(a) Example Graph and Block Adjacency Matrix
v_0
v_1
v_2
(b) Component Vector in GIM-V BL
(c) Component Vector in GIM-V DI

Figure 4.3: Propagation of component id (=1) when block width is 4. Each element in the adjacency matrix of (a) represents a 4 by 4 block; each column in (b) and (c) represents the vector after each iteration. GIM-V DL finishes in 4 iterations while GIM-V BL requires 8 iterations.

4.3.4 GIM-V DI: Diagonal Block Iteration

Specifically for the connected components algorithm (HCC), we can do one more optimization. As mentioned in Section 4.3.2, the main bottleneck of GIM-V is its shuffling and disk I/O steps. Since GIM-V iteratively runs Algorithm 4.1 and 4.2, and each Stage requires disk IO and shuffling, we could decrease running time if we decrease the number of iterations.

In HCC, it is possible to decrease the number of iterations. The main idea is to multiply diagonal matrix blocks and corresponding vector blocks as many times as possible in one iteration. Remember that multiplying a matrix and a vector corresponds to passing node ids to one step neighbors in HCC. By multiplying diagonal blocks and vectors until the contents of the vectors do not change in one iteration, we can pass node ids to neighbors located more than one step away. This is illustrated in Figure 4.3.

We see that in Figure 4.3 (c) we multiply $B_{i,i}$ with v_i several times until v_i do not change in one iteration. For example in the first iteration v_0 changed from {1,2,3,4} to {1,1,1,1} since it is multiplied to $B_{0,0}$ four times. GIM-V DI is especially useful in graphs with long chains.

The upper bound of the number of iterations in HCC DI with chain graphs is determined by the following theorem.

Theorem 4.2 (Upper bound of iterations in HCC DI)**.** *In a chain graph with length m, it takes maximum $2\lceil m/b \rceil - 1$ iterations in* HCC *DI with block size b.*

Proof. The worst case happens when the minimum node id is in the beginning of the chain. It requires 2 iterations (one for propagating the minimum node id inside the block, another for passing it to the next block) for the minimum node id to move to an adjacent block. Since the farthest block is $\lceil m/b \rceil - 1$ steps away, we need $2(\lceil m/b \rceil - 1)$ iterations. When the minimum node id reached the farthest away block, GIM-V DI requires one more iteration to propagate the minimum node id inside the last block. Therefore, we need $2(\lceil m/b \rceil - 1) + 1 = 2\lceil m/b \rceil - 1$ iterations. □

4.3.5 GIM-V NR: Node Renumbering

In HCC, the minimum node id is propagated to the other parts of the graph within at most d steps, where d is the diameter of the graph. If the node with the minimum id (which we call 'minimum node') is located at the center of the graph, then the number of iterations is small (close to *d*/2). However, if it is located at the boundary of the network, then the number of iteration can be close to d. Therefore, if we preprocess the edges so that the minimum node id is swapped to the center node id, the number of iterations and the total running time of HCC would decrease.

Finding the center node with the minimum radius could be done with the HADI (Chapter 3) algorithm. However, the algorithm is expensive for the pre-processing step of HCC. Therefore, we instead propose the following heuristic for finding the center node: we choose the center node by sampling from the high degree nodes. This heuristic is based on the observation that nodes with high degree have small radii (Observation 3 of Chapter 3). Moreover, computing the degree of very large graphs is trivial in MAPREDUCE and could be performed quickly with one job of MAPREDUCE.

After finding a center node, we need to renumber the edge file to swap the current minimum node id with the center node id. The MAPREDUCE algorithm for this renumbering is shown in Algorithm 4.3. Since the renumbering requires only filtering, it can be done with a Map-only job.

4.3.6 Analysis

We analyze the time and the space complexities of GIM-V. In the lemmas below, s is the number of machines.

Lemma 4.1 (Time Complexity of GIM-V)**.** *One iteration of* G*IM-V takes $O(\frac{|V|+|E|}{s} \log \frac{|V|+|E|}{s})$ time.*

Proof. Assuming uniformity, mappers and reducers of `Stage1` and `Stage2` receives $O(\frac{|V|+|E|}{s})$ records per machine. The running time is dominated by the sorting time for $\frac{|V|+|E|}{s}$ records, which is $O(\frac{|V|+|E|}{s} \log \frac{|V|+|E|}{s})$. □

Lemma 4.2 (Space Complexity of GIM-V)**.** G*IM-V requires $O(|V| + |E|)$ space.*

Algorithm 4.3: Renumbering the minimum node

```
Input: edge $E = \{(id_{src}, id_{dst})\}$,
    current minimum node id $minid_{cur}$, and
    new minimum node id $minid_{new}$.
Output: renumbered edge $E' = \{(id'_{src}, id'_{dst})\}$.
1: Renumber-Map(Key $k$, Value $v$):
2: $src \leftarrow k$;
3: $dst \leftarrow v$;
4: if $src = minid_{cur}$ then
5:     $src \leftarrow minid_{new}$;
6: else if $src = minid_{new}$ then
7:     $src \leftarrow minid_{cur}$;
8: end if
9: if $dst = minid_{cur}$ then
10:     $dst \leftarrow minid_{new}$;
11: else if $dst = minid_{new}$ then
12:     $dst \leftarrow minid_{cur}$;
13: end if
14: Output($src, dst$);
```

Proof. We assume the value of the elements of the input vector v is constant. Then the lemma is proved by noticing that the maximum storage is required at the output of `Stage1` mappers which requires $O(|V| + |E|)$ space up to a constant. □

4.4 Performance and Scalability

We do experiments to answer the following questions:

Q1 How does GIM-V scale up?

Q2 Which of the proposed optimizations (block multiplication, clustered edges, diagonal block iteration, and node renumbering) gives the highest performance gains?

The graphs used in our experiments at Section 4.4 and 4.5 are described in Table 4.3 whose entries are repeated from Table 2.2 for convenience.

We run GIM-V in M45 HADOOP cluster by Yahoo! and our own cluster composed of 9 machines. As we mentioned in Section 3.5, M45 is one of the top 50 supercomputers in the world with the total 1.5 Petabytes storage and 3.5 Terabytes memory. For the performance and scalability experiments, we use synthetic Kronecker graphs [Leskovec et al., 2005] since we can generate them with any size, and they are one of the most realistic graphs among synthetic graphs.

4.4.1 Results

We first show how the performance of our method changes as we add more machines. Figure 4.4 shows the running time and performance of GIM-V for PageRank with Kronecker graph of 282 million edges,

Graph	Nodes	Edges	File Size	Type	Description
YahooWeb	1.4 B	6.6 B	0.12 TB	real	WWW links in 2002
LinkedIn	7.5 M	58 MB	1 GB	real	person-person in 2006
	4.4 M	27 MB	490 MB		person-person in 2005
	1.6 M	6.8 MB	121 MB		person-person in 2004
	85 K	230 KB	4 MB		person-person in 2003
Wikipedia	3.5 M	42 M	605 MB	real	doc-doc in 2007/02
	3 M	35 M	495 MB		doc-doc in 2006/09
	1.6 M	18.5 M	252 MB		doc-doc in 2005/11
DBLP	471 K	112 K	1 MB	real	document-document
WWW-Barabasi	325 K	1.5 M	20 MB	real	WWW links in nd.edu
Flickr	404 K	2.1 M	28 MB	real	person-person
Epinions	75 K	508 K	5 MB	real	who trusts whom
Kronecker	177 K	2 B	25 GB	synthetic	from Kronecker generator [Leskovec et al., 2005]
	121 K	1.1 B	13.9 GB		
	59 K	282 M	3.3 GB		
	20 K	40 M	439 MB		

Table 4.3: Datasets. B: Billion, M: Million, K: Thousand, TB: Terabytes, GB: Gigabytes, MB: Megabytes.

and size 32 blocks if necessary.

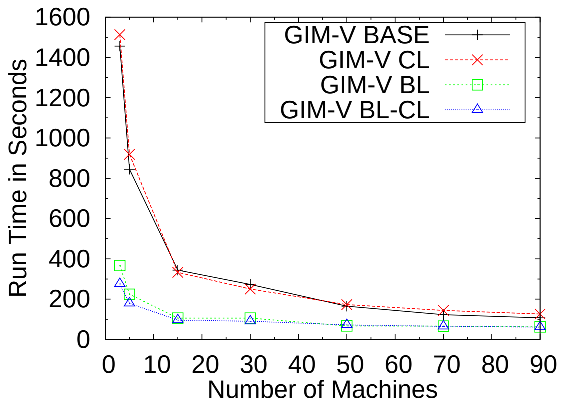
In Figure 4.4 (a), for all of the methods the running time decreases as we add more machines. Note that clustered edges (GIM-V CL) did not help performance unless it is combined with block encoding. When it is combined, however, it showed the best performance (GIM-V BL-CL).
Observation 8. G*IM-V BL-CL, which combines the block encoding with the clustering, gives the best performance.*

In Figure 4.4 (b), we see that the relative performance of each method compared to GIM-V BASE method decreases as number of machines increases. With 3 machines (minimum number of machines which HADOOP 'distributed mode' supports), the fastest method (GIM-V BL-CL) ran 5.27 times faster than GIM-V BASE. With 90 machines, GIM-V BL-CL ran 2.93 times faster than GIM-V BASE. This is expected since there are fixed component (JVM load time, disk I/O, network communication) which can not be optimized even if we add more machines.

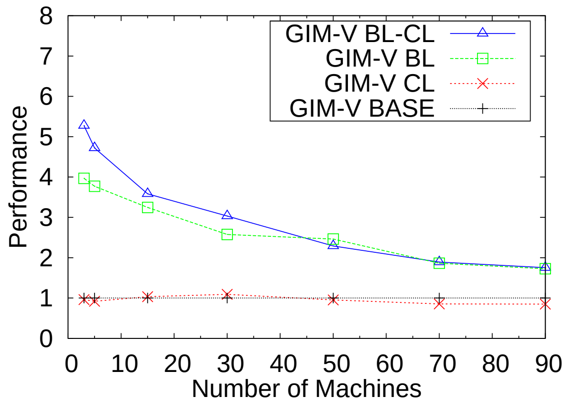
Next we show how the performance of our methods changes as the input size grows. Figure 4.4 (c) shows the running time of GIM-V with different number of edges under 10 machines. As we see, all of the methods scale linearly with the number of edges.
Observation 9. G*IM-V scales linearly with the number of edges.*

Next, we compare the performance of GIM-V DI and GIM-V BL-CL for HCC in graphs with long chains. For this experiment we made a synthetic graph whose diameter is 17, by adding a length 15 chain to the 282 million Kronecker graph which has diameter 2. As we see in Figure 4.5, GIM-V DI finished in 6 iteration while GIM-V BL-CL finished in 18 iteration. The running time of both methods for the first 6 iterations are nearly the same. Therefore, the diagonal block iteration method decreases the number of iterations while not affecting the running time of each iteration much.

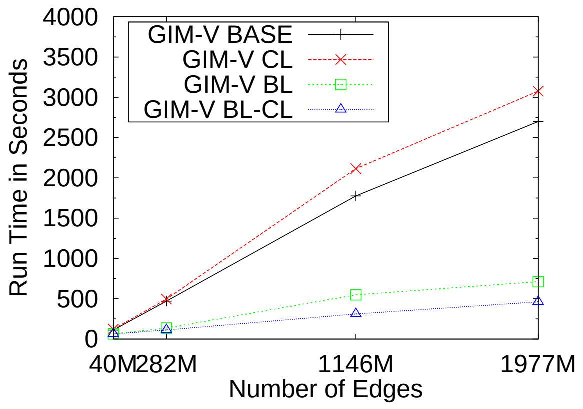
Finally, we compare the number of iterations with/without renumbering. Figure 4.6 shows the degree distribution of LinkedIn. Without renumbering, the minimum node has degree 1, which is not surprising



(a) Running time vs. Machines



(b) Performance vs. Machines



(c) Running time vs. Edges

Figure 4.4: Scalability and Performance of GIM-V. **(a):** Running time decreases quickly as more machines are added. **(b):** The performance (=$1/running\ time$) of 'BL-CL' wins more than 5x (for n=3 machines) over the 'BASE'. **(c):** Every version of GIM-V shows linear scalability.

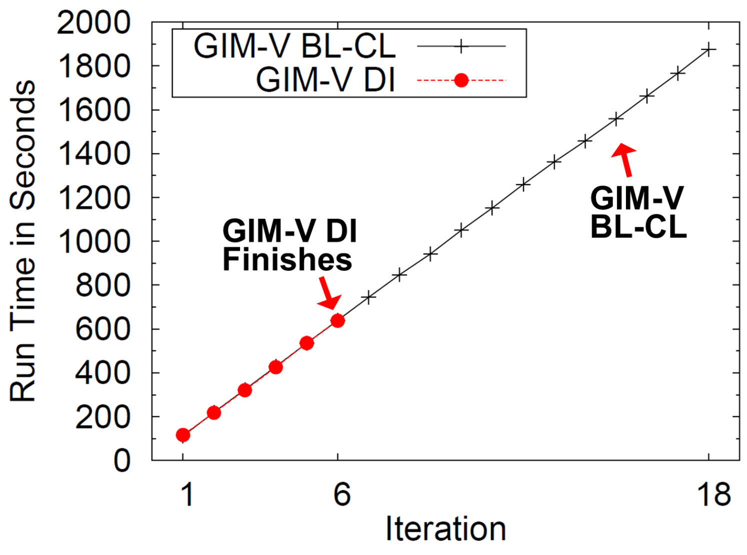


Figure 4.5: Comparison of GIM-V DI and GIM-V BL-CL for HCC. GIM-V DI finishes in 6 iterations while GIM-V BL-CL finishes in 18 iterations due to long chains.

since about 46 % of the nodes have degree 1 due to the power-law behavior of the degree distribution. We show the number of iterations after changing the minimum node to each of the top 5 highest-degree nodes in Figure 4.7. We see that the renumbering decreased the number of iterations to 81 % of the original. Similar results are observed for the Wikipedia graph in Figure 4.8 and 4.9. The original minimum node has degree 1, and the number of iterations decreased to 83 % of the original after renumbering.

4.5 GIM-V At Work

In this section we use GIM-V for mining very large graphs. We analyze connected components, diameter, and PageRank of large real world graphs. We show that GIM-V is useful for finding patterns, outliers, and interesting observations.

4.5.1 Connected Components of Real Networks

We used the LinkedIn social network and Wikipedia page-linking-to-page network, along with the YahooWeb graph for connected component analysis. Figure 4.10 shows the evolution of connected components of LinkedIn and Wikipedia data. Figure 4.11 show the distribution of connected components in the YahooWeb graph. We have the following observations.

Power Laws in Connected Components Distributions. We observed power law relation of count and size of small connected components in Figure 4.10 (a,b) and Figure 4.11. This reflects that the connected components in real networks are formed by processes similar to Chinese Restaurant Process and Yule distribution [Newman, 2005].

Stable Connected Components After Gelling Point. In Figure 4.10 (a), the distribution of connected components remain stable after a ‘gelling’ point [Mcglohon et al., 2008] at year 2003. We see that the

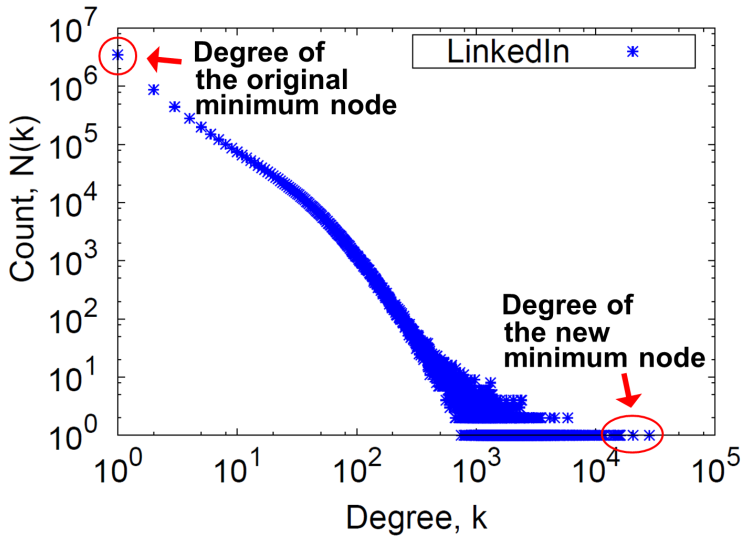


Figure 4.6: Degree distribution of LinkedIn. Notice that the original minimum node has degree 1, which is highly probable given the power-law behavior of the degree distribution. After the renumbering, the minimum node is replaced with a highest-degree node.

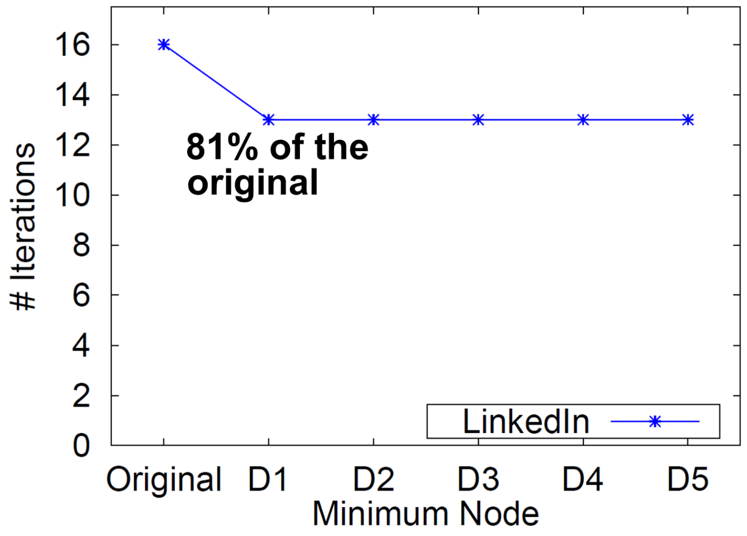


Figure 4.7: Number of iterations vs. the minimum node of LinkedIn, for connected components. Di represents the node with i-th largest degree. Notice that the number of iterations decreased by 19 % after renumbering.

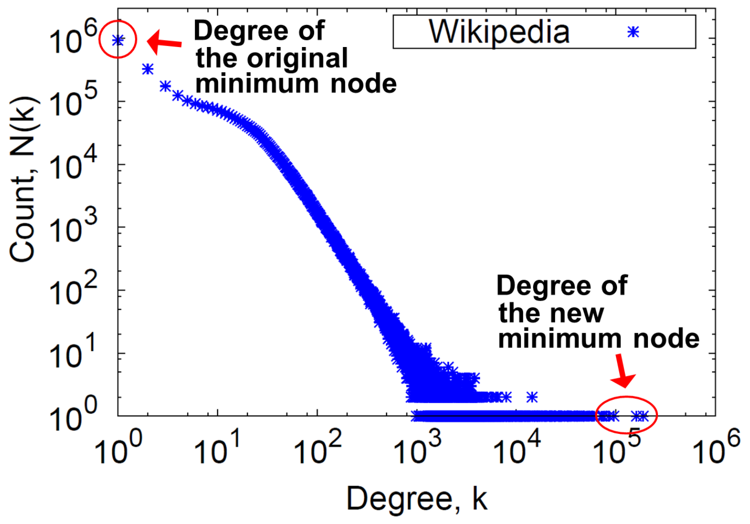


Figure 4.8: Degree distribution of Wikipedia. Notice that the original minimum node has degree 1, as in LinkedIn. After the renumbering, the minimum node is replaced with a highest-degree node.

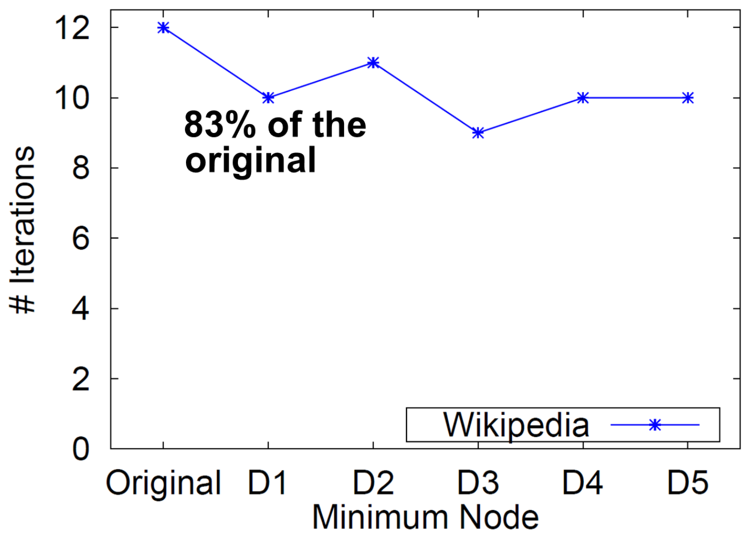
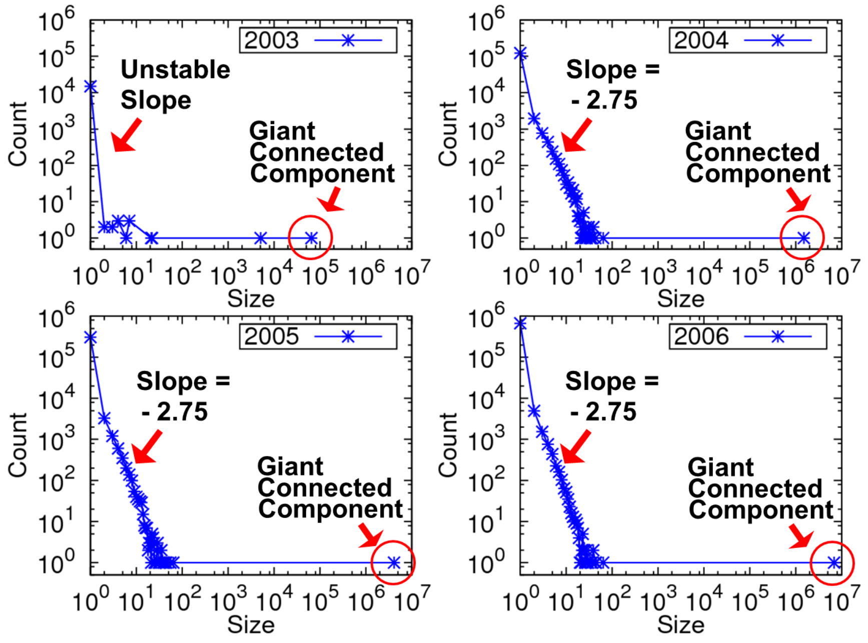
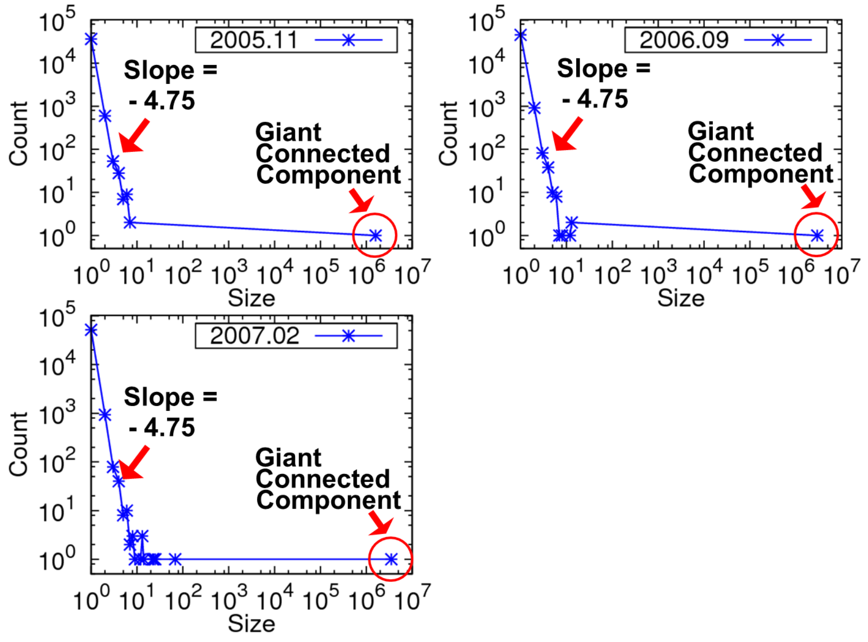


Figure 4.9: Number of iterations vs. the minimum node of Wikipedia, for connected components. Di represents the node with i-th largest degree. Notice that the number of iterations decreased by 17 % after renumbering.



(a) Connected Components of LinkedIn



(b) Connected Components of Wikipedia

Figure 4.10: The evolution of connected components. **(a):** The giant connected component grows for each year. However, the second largest connected component do not grow above Dunbar's number (≈ 150) and the slope of the size distribution remains constant after the gelling point at year 2003. **(b):** As in LinkedIn, notice the growth of giant connected component and the constant slope of the size distribution.

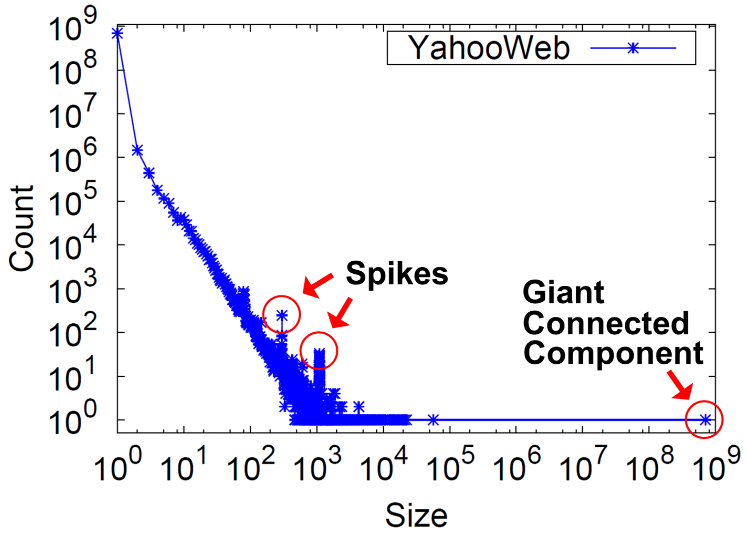


Figure 4.11: Connected Components of YahooWeb. Notice the two anomalous spikes which are far from the constant-slope line. Most of them are domain selling or porn sites which are replicated from templates.

slope of the size distribution do not change after year 2003. We observed the same phenomenon in Wikipedia graph in Figure 4.10 (b). The graph show stable slopes from the beginning, since the network were already mature in year 2005.

Absorbed Connected Components and Dunbar's Number. In Figure 4.10 (a), we find two large connected components in year 2003. However they merged in year 2004. The giant connected component keeps growing, while the second and the third largest connected components do not grow beyond size 100 until they are absorbed to the giant connected component in Figure 4.10 (a) and (b). This agrees with the observation [Mcglohon et al., 2008] that the size of the second/third connected components remains constant or oscillates. Lastly, the maximum connected component size except the giant connected component in the LinkedIn graph agrees well with Dunbar's number [Dunbar, October 1998], which says that the maximum community size in social networks is roughly 150.

Anomalous Connected Components. In Figure 4.11, we found two outstanding spikes. In the first spike at size 300, more than half of the components have exactly the same structure and they were made from a domain selling company where each component represents a domain to be sold. The spike happened because the company *replicated* sites using the same template, and injected the disconnected components into WWW network. In the second spike at size 1101, more than 80 % of the components are adult sites disconnected from the giant connected component. By looking at the distribution plot of connected components, we could find interesting communities with special purposes which are disconnected from the rest of the Internet.

4.5.2 PageRank Scores of Real Networks

We analyzed the PageRank scores of the nodes of real graphs, using GIM-V. Figure 4.12 and 4.13 show the distribution of the PageRank scores for the Web graphs, and Figure 4.14 shows the evolution of PageRank scores of the LinkedIn and Wikipedia graphs. We have the following observations.

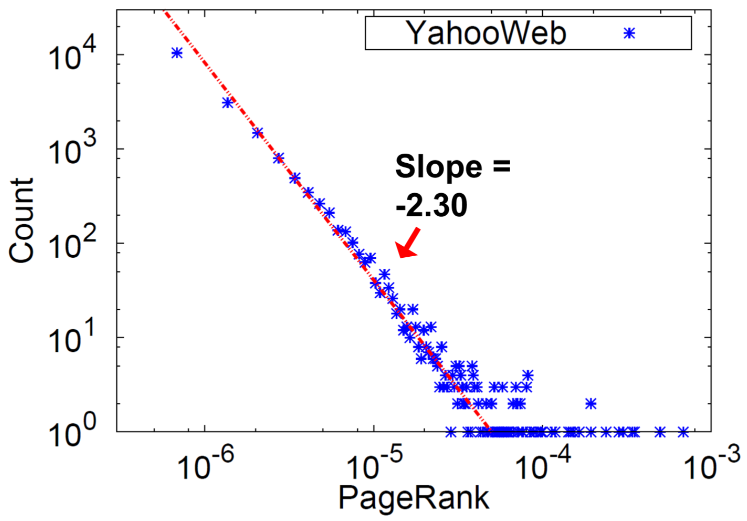


Figure 4.12: PageRank distribution of YahooWeb. The distribution follows a power law with an exponent 2.30.

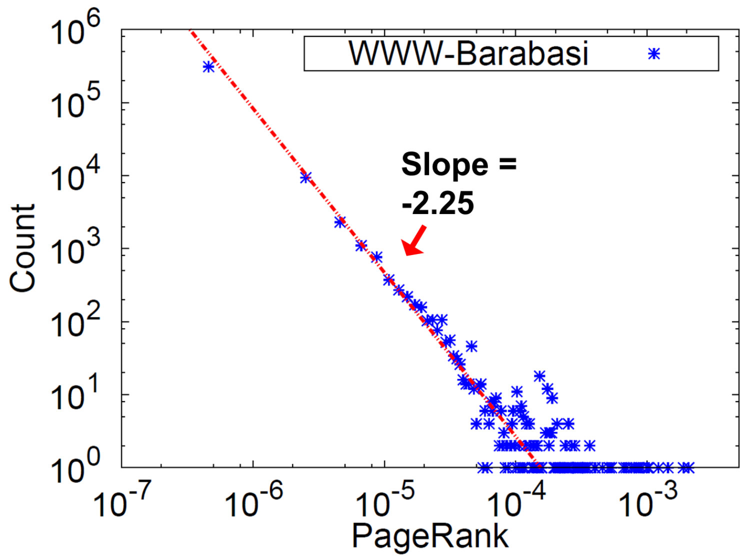
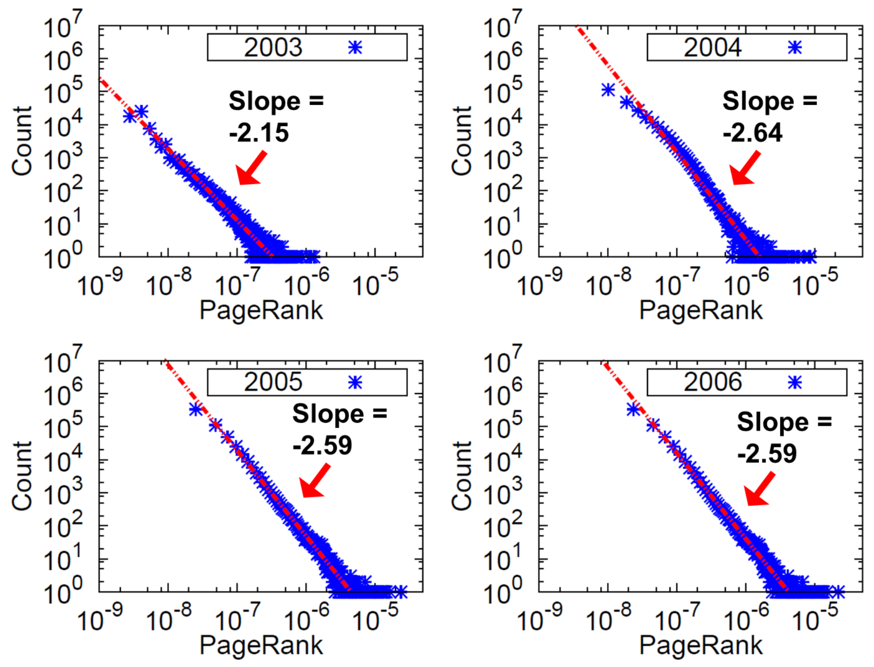
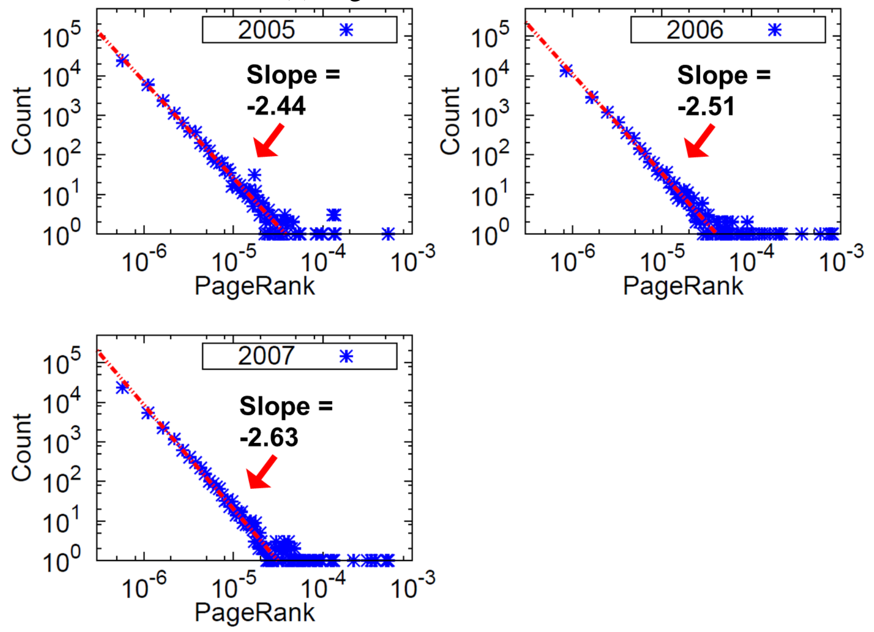


Figure 4.13: PageRank distribution of WWW-Barabasi. The distribution follows a power law with an exponent 2.25.



(a) PageRanks of LinkedIn



(b) PageRanks of Wikipedia

Figure 4.14: The evolution of PageRanks. **(a):** The distributions of PageRanks follows a power-law. However, the exponent at year 2003, which is around the gelling point, is much different from year 2004, which are after the gelling point. The exponent increases after the gelling point and becomes stable. Also notice the maximum PageRank after the gelling point is about 10 times larger than that before the gelling point due to the emergence of the giant connected component. **(b):** Again, the distributions of PageRanks follows a power-law. Since the gelling point is before year 2005, the three plots show similar characteristics: the maximum PageRanks and the slopes are similar.

Power Laws in PageRank Distributions. In Figure 4.12, 4.13, and 4.14, we observe power-law relations between the PageRank score and the number of nodes with such PageRank. Pandurangan et al. [Pandurangan et al., August 2002] observed such a power-law relationship for a 1.69 million network. Our result is that the same observation holds true for about *1,000 times* larger network with 1.4 *billion* pages snapshot of the Internet. The top 3 highest PageRank sites for the year 2002 are `www.careerbank.com`, `access.adobe.com`, and `top100.rambler.ru`. As expected, they have huge in-degrees (from $\approx$70K to $\approx$70M).

PageRank and the Gelling Point. In the LinkedIn network (see Figure 4.14 (a)), we see a discontinuity for the power-law exponent of the PageRank distribution, before and after the gelling point at year 2003. For the year 2003 (up to the gelling point), the exponent is 2.15; from 2004 (after the gelling point), the exponent stabilizes around 2.59. Also, the maximum PageRank value at 2003 is around 10^{-6}, which is $\frac{1}{10}$ of the maximum PageRank from 2004. This behavior is explained by the emergence of the giant connected component at the gelling point: before the gelling point, there are many small connected components where no outstanding node with large PageRank exists. After the gelling point, several nodes with high PageRank appear within the giant connected component. In the Wikipedia network (see Figure 4.14 (b)), we see the same behavior of the network after the gelling point. Since the gelling point is before year 2005, we see that the maximum PageRank-score and the slopes are similar for the three graphs from 2005.

4.5.3 Diameter of Real Network

We analyzed the diameter and radius of real networks with GIM-V. Figure 4.15 shows the radius plot of real networks. We have following observations:

Small Diameter. For all the graphs in Figure 4.15, the average diameter was less than 6.09. This means that the real world graphs are well connected.

Constant Diameter over Time. For LinkedIn graph, the average diameter was in the range of 5.28 and 6.09. For Wikipedia graph, the average diameter was in the range of 4.76 and 4.99. Note that the diameter do not monotonically increase as network grows: they remain constant or shrinks over time.

Bimodal Structure of Radius Plot. For every plot, we observe a bimodal shape which reflects the structure of these real graphs. The graphs have one giant connected component where majority of nodes belong to, and many smaller connected components whose size follows a power law. Therefore, the first mode is at radius zero which comes from one-node components; second mode (e.g., at radius 6 in Epinions) comes from the giant connected component.

4.6 Background

We briefly review the related work on connected components computation. There are several connected component algorithms, using Breadth-First Search, Depth-First-Search, "propagation" [Shiloach and Vishkin, 1982, Awerbuch and Shiloach, 1983, Hirschberg et al., 1979], or "contraction" [Greiner, June 1994] . These works rely on a shared memory model which limits their ability to handle large, disk-resident graphs. In contrast, our GIM-V handles very large graphs which spans multiple disks over multiple machines.

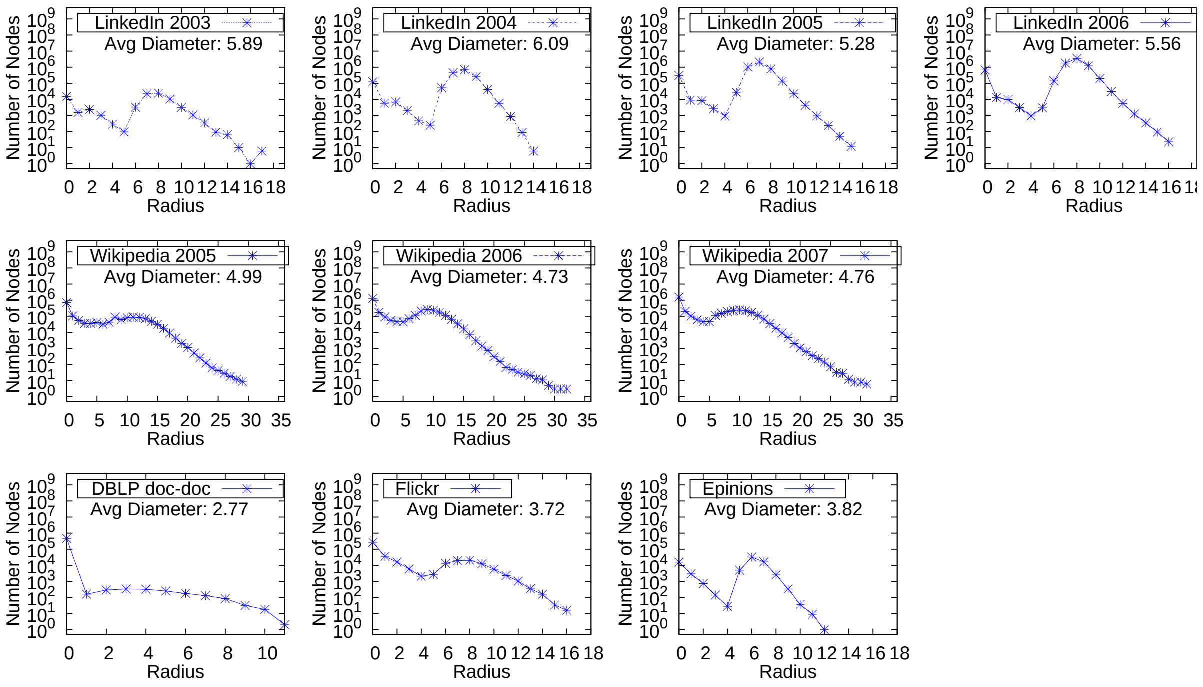


Figure 4.15: Radius of real graphs. X axis: radius in linear scale. Y axis: number of nodes in log scale. **(Row 1):** LinkedIn from 2003 to 2006. **(Row 2):** Wikipedia from 2005 to 2007. **(Row 3):** DBLP, Flickr, and Epinions. Notice that all the radius plots have the bimodal structure due to many smaller connected components (first mode) and the giant connected component (second mode).

4.7 Conclusion

In this chapter we analyze the structure of large graphs using HADOOP. The main contributions are the following:

- We identify the common, underlying primitive of several graph mining operations, and we show that it is a generalized form of a matrix-vector multiplication. We call this operation Generalized Iterative Matrix-Vector multiplication and showed that it includes the diameter estimation, the PageRank estimation, RWR calculation, and finding connected-components, as special cases.
- Given its importance, we propose several optimizations (block-multiplication, diagonal block iteration, node renumbering, and etc.) and reported the winning combination, which is *5 times* faster than the naive implementation.
- We implement GIM-V and run it on M45. We analyze real world graphs to reveal important patterns including power law tails, stability of connected components, and anomalous components. Our largest graph, “YahooWeb”, spanned 120Gb, and is one of the largest publicly available graph that was ever studied.

Part II

Advanced Graph Algorithms

Part II - Advanced Graph Algorithms: Overview

In this part we describe advanced graph algorithms for three important graph mining tasks: inference in graph, spectral graph analysis, and tensor analysis.

First, we tackle the inference task: given a graph and a set of labeled nodes, infer the labels of initially unlabeled nodes. We propose HADOOP LINE GRAPH FIXED POINT (HA-LFP), an efficient distributed algorithm for inference in billion-scale graphs, using HADOOP platform.

Second, we design and implement algorithms for spectral analysis of graphs: i.e., to study the eigenvalues and eigenvectors of graph adjacency matrices. Spectral analysis on graphs leads to many interesting applications including triangle counting, and our proposed HEIGEN algorithm handles $1000\times$ larger matrices than existing algorithms.

Finally, we study tensors, or multi dimensional arrays: e.g., predicates (subject, verb, object) in knowledge bases, and hyperlinks/anchor texts in Web graphs. We generalize the spectral analysis algorithm to multiple dimensions, and propose GIGATENSOR, a large scale tensor decomposition algorithm which solves more than $100\times$ bigger problems than existing methods. We study a large knowledge base tensor, and present interesting findings which include the discovery of potential synonyms among millions of noun-phrases.

Chapter 5

Inference in Graph

In this chapter, we focus on *inference*, which often corresponds, intuitively, to “guilt by association” scenarios. For example, if a person is a drug-abuser, probably its friends are so, too; if a node in a social network is of male gender, his dates are probably females. We show how to do inference on such huge graphs through our proposed HADOOP LINE GRAPH FIXED POINT (HA-LFP), an efficient parallel algorithm for sparse billion-scale graphs, using the HADOOP platform.

Our contributions include (a) the design of HA-LFP, observing that it corresponds to a fixed point on a *line graph* induced from the original graph; (b) scalability analysis, showing that our algorithm scales up well with the number of edges, as well as with the number of machines; and (c) experimental results on two private, as well as two of the largest publicly available graphs — the Web Graphs from Yahoo! (6.6 billion edges and *0.24 Tera bytes*), and the Twitter graph (3.7 billion edges and *0.13 Tera bytes*). We evaluate our algorithm using M45, and we report patterns and anomalies discovered by our algorithm, which would be invisible otherwise.

5.1 Introduction

Given a large graph, with millions or billions of nodes, how can we find patterns and anomalies? One method to do that is through “guilt by association”: if we know that nodes of type “A” (say, males) tend to interact/date nodes of type “B” (females), we can infer the unknown gender of a node, by checking the gender of the majority of its contacts. Similarly, if a node is a telemarketer, most of its contacts will be normal phone users (and *not* telemarketers, or 800 numbers).

We show that the “guilt by association” approach can find useful patterns and anomalies, in large, real graphs. The typical way to handle this is through the so-called *Belief Propagation (BP)* [Pearl, 1982, Yedidia et al., 2003]. BP has been successfully used for social network analysis [Chau et al., 2006], fraud detection [McGlohon et al., 2009], computer vision [Felzenszwalb and Huttenlocher, 2006], error-correcting codes, and many other domains. In this work, we address the research challenge of *scalability* - we show how to run BP on a very large graph with billions of nodes and edges. Our contributions are the following:

1. We observe that the Belief Propagation algorithm is essentially a recursive equation on the *line graph* induced from the original graph. Based on this observation, we formulate the BP problem as

Symbol	Definition
n	number of nodes in a graph
l	number of edges in a graph
S	set of states
$\phi_i(s)$	prior of node i being in state s
$\psi_{ij}(s', s)$	edge potential when nodes i and j being in states s' and s, respectively
$m_{ij}(s)$	message that node i sends to node j expressing node i's belief of node j's being in state s
$b_i(s)$	belief of node i being in state s
MRF	Markov Random Fields
LFP	LINE GRAPH FIXED POINT

Table 5.1: Table of symbols.

finding a fixed point on the line graph. We propose the LINE GRAPH FIXED POINT (LFP) algorithm and show that it is a generalized form of a linear algebra equation.

2. We formulate and devise an efficient algorithm for the LFP that runs on the HADOOP platform, called HADOOP LINE GRAPH FIXED POINT (HA-LFP).
3. We run experiments on a HADOOP cluster and analyze the running time. We analyze the large real-world graphs including YahooWeb and Twitter with HA-LFP, and show patterns and anomalies.

The rest of the chapter is organized as follows. Section 5.2 discusses the related works on the Belief Propagation and HADOOP. Section 5.3 describes our formulation of the Belief Propagation in terms of LINE GRAPH FIXED POINT (LFP), and Section 5.4 provides a fast algorithm in HADOOP. Section 5.5 shows the scalability results, and Section 5.6 gives the results of analyzing the large, real-world graphs. We conclude in Section 5.7.

To enhance readability of this chapter, we have listed the symbols frequently used in this chapter in Table 5.1.

5.2 Related Work

We review the related works on Belief Propagation (BP).

Belief Propagation(BP) [Pearl, 1982] is an efficient inference algorithm for probabilistic graphical models. Since its proposal, it has been widely, and successfully, used in a myriad of domains to solve many important problems (some are seemingly unrelated at the first glance). For example, BP is used in some of the best error-correcting codes, such as the *Turbo code* and *low-density parity-check* code, that approach channel capacity. In computer vision, BP is among the top contenders for stereo shape estimation and image restoration (e.g., denoising) [Felzenszwalb and Huttenlocher, 2006]. BP has also been used for fraud detection, such as for unearthing fraudsters and their accomplices lurking in online auctions [Chau et al., 2006], and pinpointing misstated accounts in general ledger data for the financial domain [McGlohon et al., 2009].

BP is typically used for computing the *marginal distribution* for the unobserved nodes in a graph, conditional on the observed ones; we will only discuss this version in this chapter, though with slight and trivial modifications to our implementation, the *most probable distribution* of node states can also be computed.

BP was first proposed for *trees* [Pearl, 1982] and it could compute the exact marginal distributions; it was later applied on general graphs [Pearl, 1988] as an approximate algorithm. When the graph contains cycles or loops, the BP algorithm applied on it is called *loopy BP*, which is also the focus of this work.

BP is generally applied on graphs whose nodes have finite number of states (treating each node as a *discrete* random variable). Gaussian BP is a variant of BP where its underlying distributions are Gaussian [Weiss and Freeman, 2001]. Generalized BP [Yedidia et al., 2003] allows messages to be passed between subgraphs, which can improve accuracy in the computed beliefs and promote convergence.

BP is computationally-efficient; its running time scales linearly with the number of edges in the graph. However, for graphs with *billions* of nodes and edges — a focus of our work — this cost becomes significant. There are several recent works that investigated parallel BP on multicore shared memory [Gonzalez et al., 2009b] and MPI [Gonzalez et al., 2009a, Mendiburu et al., 2007]. However, all of them assume the graphs would fit in the main memory (of a single computer, or a computer cluster). Our work specifically tackles the important, and increasingly prevalent, situation where the graphs would not fit in main memory.

5.3 Proposed Method

In this section, we describe LINE GRAPH FIXED POINT (LFP), our proposed parallel formulation of the BP on HADOOP. We first describe the standard BP algorithm, and then explains our method in detail.

5.3.1 Belief Propagation

We provide a quick overview of the Belief Propagation (BP) algorithm, which briefly explains the key steps in the algorithm and their formulation; this information will help our readers better understand how our implementation nontrivially captures and optimizes the algorithm in latter sections. For detailed information regarding BP, we refer our readers to the excellent article by Yedidia et al. [Yedidia et al., 2003].

The BP algorithm is an efficient method to solve inference problems for probabilistic graphical models, such as Bayesian networks and pairwise Markov Random Fields (MRF). In this work, we focus on pairwise MRF, which has seen empirical success in many domains (e.g., Gallager codes, image restoration) and is also simpler to explain; the BP algorithms for other types of graphical models are mathematically equivalent [Yedidia et al., 2003].

When we view an undirected simple graph $G = (V, E)$ as a pairwise MRF, each node i in the graph becomes a random variable X_i, which can be in a discrete number of states S. The goal of the inference is to find the marginal distribution $P(x_i)$ for all node i, which is an NP-complete problem.

Fortunately, BP may be used to solve this problem approximately (for MRF; exactly for trees). At a high level, BP infers the "true" (or so-called "hidden") distribution of a node from some prior (or "observed") knowledge about the node, and from the node's neighbors. This is accomplished through iterative message passing between all pairs of nodes v_i and v_j. We use $m_{ij}(x_j)$ to denote the message sent from i to j, which intuitively represents i's opinion about j's likelihood of being in state x_j. The prior knowledge about a node i, or the prior probabilities of the node being in each possible state are expressed through the *node potential function* $\phi(x_i)$. This prior probability may simply be called a *prior*. The message-passing procedure stops if the messages no longer change much from one iteration to the another — or equivalently when the nodes' marginal probabilities are no longer changing much. The estimated marginal probability is called *belief*, or symbolically $b_i(x_i)$ ($\approx P(x_i)$).

In detail, messages are obtained as follows. Each edge e_{ij} is associated with messages $m_{ij}(x_j)$ and $m_{ji}(x_i)$ for each possible state. Provided that all messages are passed in every iteration, the order of passing can be arbitrary. Each message vector m_{ij} is normalized to sum to one. Normalization also prevents numerical underflow (or zeroing-out values). Each outgoing message from a node i to a neighbor j is generated based on the incoming messages from the node's other neighbors. Mathematically, the message-update equation is:

$$m_{ij}(x_j) = \sum_{x_i} \phi_i(x_i)\psi_{ij}(x_i, x_j)\frac{\prod_{k\in N(i)} m_{ki}(x_i)}{m_{ji}(x_i)}, \tag{5.1}$$

where $N(i)$ is the set of neighboring nodes of i, and $\psi_{ij}(x_i, x_j)$ is called the *edge potential*; intuitively, it is a function that *transforms* a node's incoming messages collected into the node's outgoing ones. Formally, $\psi_{ij}(x_i, x_j)$ equals the probability of a node i being in state x_i and that its neighbor j is in state x_j.

The algorithm stops when the beliefs converge (within some threshold, e.g., 10^{-5}), or a maximum number of iterations has finished. Although convergence is not guaranteed theoretically for general graphs, except for those that are trees, the algorithm often converges in practice, where convergence is quick and the beliefs are reasonably accurate. When the algorithm ends, the node beliefs are determined as follows:

$$b_i(x_i) = c\phi_i(x_i)\prod_{k\in N(i)} m_{ki}(x_i), \tag{5.2}$$

where c is a normalizing constant.

5.3.2 Recursive Equation

As seen in the last section, BP is computed by iteratively running equations (5.1) and (5.2), as described in Algorithm 5.1.

In a shared-memory system in which random access to memory is allowed, the implementation of Algorithm 5.1 might be straightforward. However, large scale algorithm for MAPREDUCE requires careful thinking since the random access is not allowed and the data are read sequentially within mappers and reducers. A good news is that the two equations (5.1) and (5.2) involve only local communications between neighboring nodes, and thus it seems hopeful to develop a parallel algorithm for HADOOP. Naturally, one might think of an iterative algorithm in which nodes exchange messages to update its beliefs using an extended form of matrix-vector multiplication (Chapter 4). In such formulation, a current belief vector and the message matrix is combined to compute the next belief vector. Thus, we want a recursive equation to update the belief vector. However, such an equation cannot be derived due to the denominator $m_{ji}(x_i)$

Algorithm 5.1: Belief Propagation

```
Input: edge $E$,
    node prior $\phi^{n\times 1}$, and
    propagation matrix $\psi^{S\times S}$.
Output: belief matrix $b^{n\times S}$.
1: while m does not converge do
2:   for $(i,j) \in E$ do
3:     for $s \in S$ do
4:       $m_{ij}(s) \leftarrow \sum_{s'} \phi_i(s')\psi_{ij}(s',s)\prod_{k\in N(i)\backslash j} m_{ki}(s')$;
5:     end for
6:   end for
7: end while
8: for $i \in V$ do
9:   for $s \in S$ do
10:    $b_i(s) \leftarrow c\phi_i(s)\prod_{k\in N(i)} m_{ki}(s)$;
11:  end for
12: end for
```

in Equation (5.1). If it were not for the denominator, we could get the following modified equation where the superscript t and $t-1$ denote the iteration number:

$$\begin{aligned} m_{ij}(x_j)^{(t)} &= \sum_{x_i} \phi_i(x_i)\psi_{ij}(x_i,x_j) \prod_{k\in N(i)} m_{ki}(x_i)^{(t-1)} \\ &= \sum_{x_i} \psi_{ij}(x_i,x_j)\frac{b_i(x_i)^{(t-1)}}{c}, \end{aligned}$$

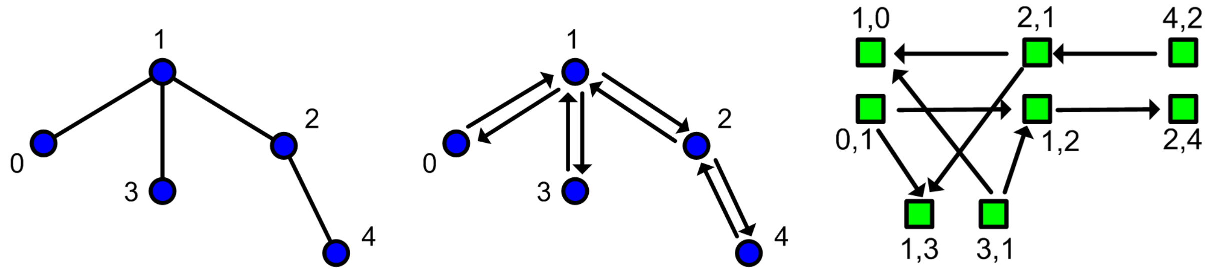
and thus

$$\begin{aligned} b_i(x_i)^{(t)} &= c\phi_i(x_i) \prod_{k\in N(i)} m_{ki}(x_i)^{(t-1)} \\ &= \phi_i(x_i) \prod_{k\in N(i)} \sum_{x_k} \psi_{ki}(x_k,x_i) b_k(x_k)^{(t-2)}. \end{aligned} \tag{5.3}$$

Notice that the recursive equation (5.3) is a fake, imaginary equation derived from the assumption that equation (5.1) has no denominator. Although the recursive equation for the belief vector cannot be acquired by this way, there is a more direct and intuitive way to get a recursive equation. We will describe how to get it in the next section.

5.3.3 Main Idea: Line graph Fixed Point (LFP)

How can we get the recursive equation for the BP? What we need is a tractable recursive equation well-suited for large scale MAPREDUCE framework. In this section, we describe LINE GRAPH FIXED POINT



(a) Original graph (b) Directed graph (c) Directed line graph

Figure 5.1: Converting a undirected graph to a directed line graph. **(a to b):** Replace a undirected edge with two directed edges. **(b to c):** For an edge (i, j) in (b), make a node (i, j) in (c). Make a directed edge from (i, j) to (k, l) in (c) if $j = k$ and $i \neq l$. The rectangular nodes in (c) corresponds to edges in (b).

(LFP), our formulation of BP in terms of finding the fixed point of an induced graph from the original graph. As seen in the last section, a recursive equation to update the beliefs cannot be acquired due to the denominator in the message update equation. Our main idea to solve the problem is to flip the notion of the nodes and edges in the original graph and thus use the equation (5.1), without modification, as the recursive equation for updating the 'nodes' in the new formulation. The 'flipping' means we consider an induced graph, called the *line graph*, whose nodes correspond to edges in the original graph, and the two nodes in the induced graph are connected if the corresponding edges in the original graph are incident. Notice that for each edge (i, j) in the original graph, two messages need to be defined since m_{ij} and m_{ji} are different. Thus, the line graph should be directed, although the original graph is undirected. Formally, we define the 'directed line graph' as follows.

Definition 3 (Directed Line Graph)**.** *Given a directed graph G, its directed line graph L(G) is a graph such that each node of L(G) represents an edge of G, and there is an edge from v_i to v_j of L(G) if the corresponding edges e_i and e_j form a length-two directed path from e_i to e_j in G.*

For example, see Figure 5.1 for a graph and its directed line graph. To convert a undirected line graph G to a directed line graph $L(G)$, we first convert G to a directed graph by converting each undirected edge to two directed edges. Then, a directed edge from v_i to v_j in $L(G)$ is created if their corresponding edges e_i and e_j form a directed path e_i to e_j in G.

Now, we derive the exact recursive equation on the line graph. Let G be the original undirected graph with n nodes and l edges, and $L(G)$ be the directed line graph of G with $2l$ nodes as defined by Definition 3. The (i, j)th element $L(G)_{i,j}$ is defined to be 1 if the edge exist, or 0 otherwise. Let m be a $2l$-vector whose element corresponding to the edge (i, j) in G contains the reverse directional message m_{ji}. The reason of this reverse directional message will be described soon. Let ϕ be a n-vector containing priors of each node. We build a $2l$-vector φ as follows: if the kth element φ_k of φ corresponds to an edge (i, j) in G, then set φ_k to $\phi(i)$. A standard matrix-vector multiplication with vector addition operation on $L(G)$, m, φ is expressed as follows (we abuse notation to let $L(G)$ denote the l by l adjacency matrix of the line graph):

$$m' = L(G) \times m + \varphi,$$

where

$$m'_i = \sum_{j=1}^{n} L(G)_{i,j} \times m_j + \varphi_i.$$

In the above equation, four operations are used to get the result vector:

1. $\texttt{combine2}(L(G)_{i,j}, m_j)$: multiply $L(G)_{i,j}$ and m_j.
2. $\texttt{combineAll}_i(y_1, ..., y_n)$: sum n multiplication results for node i.
3. $\texttt{sumVector}(\varphi_i, v_{aggr})$: add φ_i to the result v_{aggr} of $\texttt{combineAll}_i$.
4. $\texttt{assign}(m_i, oldval_i, newval_i)$: overwrite the previous value $oldval_i$ of m_i with the new value $newval_i$ to make m'_i.

Now, we generalize the operators $\times$ and $+$ to $\times_G$ and $+_G$, respectively, so that the four operations can be any functions of their arguments. In this generalized setting, the matrix-vector multiplication with vector addition operation becomes

$$m' = L(G) \times_G m +_G \varphi,$$

where

$$m'_i = \texttt{assign}(m_i, oldval_i,$$
$$\texttt{sumVector}(\varphi_i, \texttt{combineAll}_i(\{y_j \mid j = 1..n, \text{ and } y_j = \texttt{combine2}(L(G)_{i,j}, m_j)\}))).$$

An important observation is that the BP equation (5.1) can be represented by this generalized form of the matrix-vector multiplication with vector addition. For simplifying the explanation, we omit the edge potential ψ_{ij}, since it is a tiny information (e.g. 2 by 2 or 3 by 3 table), and the summation over x_i, both of which can be accommodated easily. Then, the BP equation (5.1) is expressed by

$$m' = L(G)^T \times_G m +_G \varphi, \tag{5.4}$$
$$m'' = ChangeMessageDirection(m'), \tag{5.5}$$

where

$$m'_i = \texttt{sumVector}(\varphi_i, \texttt{combineAll}_i(\{y_j \mid j = 1..n, \text{ and } y_j = \texttt{combine2}(L(G)^T_{i,j}, m_j)\})),$$

the four operations are defined by

1. $\texttt{combine2}(L(G)_{i,j}, m_j) = L(G)_{i,j} \times m_j$.
2. $\texttt{combineAll}_i(y_1, ..., y_n) = \prod_{j=1}^{n} y_j$.
3. $\texttt{sumVector}(\varphi_i, v_{aggr}) = \varphi_i \times v_{aggr}$.
4. $\texttt{assign}(m_i, oldval_i, newval_i) = newval_i / val_i$.

and the ChangeMessageDirection function is defined by Algorithm 5.2. The computed m'' of equation (5.5) is the updated message which can be used as m in the next iteration. Thus, our LINE GRAPH FIXED POINT (LFP) comprises running the equation (5.4) and (5.5) iteratively until a fixed point, where the message vector converges, is found.

Two details should be addressed for the complete description of our method. First, notice that $L(G)^T$, instead of $L(G)$, is used in the equation (5.4). The reason is that a message should aggregate other messages pointing *to* itself, which is the reverse direction of the line graph construction. Second, what is the use of ChangeMessageDirection function? We mentioned earlier that the BP equation (5.1) contained a denominator m_{ji} which is the reverse directional message. Thus, the input message vector m of equation (5.4) contains the reverse directional message. However, the result message vector m' of equation (5.4) contains the forward directional message. For the m' to be used in the next iteration, it needs to change the direction of the messages, and that is what ChangeMessageDirection does.

Algorithm 5.2: ChangeMessageDirection

```
Input: message vector $m$ of length $2l$.
Output: new message vector $m'$ of length $2l$.
1: for $k \in 1..2l$ do
2:     $(i, j) \leftarrow$ edge in $G$ corresponding to $m_k$;
3:     $k' \leftarrow$ element index of $m$ corresponding to the edge $(j, i)$ in $G$;
4:     $m'_{k'} \leftarrow m_k$;
5: end for
```

Algorithm 5.3: LINE GRAPH FIXED POINT (LFP)

```
Input: edge $E$ of a undirected graph $G = (V, E)$,
    node prior $\phi^{n\times 1}$, and
    propagation matrix $\psi^{S\times S}$.
Output: belief matrix $b^{n\times S}$.
1: $L(G) \leftarrow$ directed line graph from $E$;
2: $\varphi \leftarrow$ line prior vector from $\phi$;
3: while m does not converge do
4:     for $s \in S$ do
5:         $m(s)^{next} = L(G) \times_G m^{cur} +_G \varphi$;
6:     end for
7:     for $i \in V$ do
8:         for $s \in S$ do
9:             $b_i(s) \leftarrow c\phi_i(s) \prod_{j\in N(i)} m_{ji}(s)$;
10:        end for
11:    end for
12: end while
```

In sum, a generalized matrix-vector multiplication with addition is the recursive message update equation which is run until convergence. The resulting algorithm LFP is summarized in Algorithm 5.3.

5.4 Fast Algorithm for Hadoop

In this section, we first describe the naive algorithm for LFP and propose an efficient algorithm.

5.4.1 Naive Algorithm

The formulation of BP in terms of the fixed point in the line graph provides an intuitive way to understand the computation. However, a naive algorithm without careful design is not efficient for the following reason. In a naive algorithm, we first build the matrix for the line graph $L(G)$ and the message vector, and apply the recursive equation on them. The problem is that a node in G with degree d will generate $d(d-1)$ edges in $L(G)$. Since there exists many nodes with a very large degree in real-world graphs due to the well-known power-law degree distribution, the number of nonzero elements will grow too large. For example, the YahooWeb graph in Section 5.5 has several nodes with the several-million degree. As a result, the number of nonzero elements in the corresponding line graph is more than 1 trillion. Thus, we need an efficient algorithm for dealing with the problem.

5.4.2 Lazy Multiplication

The main idea to solve the problem in the previous section is not to build the line graph explicitly: instead, we do the same computation on the original graph, or perform a 'lazy' multiplication. The crucial observation is that the edges in the original graph G contain all the edge information in $L(G)$: each edge $e \in E$ of G is a node in $L(G)$, and $e_1, e_2 \in G$ are adjacent in $L(G)$ if and only if they share the node in G. For each edge (i, j) in G, we associate the reverse message m_{ji}. Then, grouping edges by source node id i enables us to get all the messages pointing *to* the source node. Thus, for each node j of i's neighbors, the updated message m_{ij} is computed by calculating $\frac{\prod_{k \in N(i)} m_{ki}(x_i)}{m_{ji}(x_i)}$ from the messages in the grouped edges (incorporating priors and the propagation matrix is described soon). Since we associate the reverse message for each edge, the output triple (src, dst, reverse message) is (j, i, m_{ij}).

An issue in computing $\frac{\prod_{k \in N(i)} m_{ki}(x_i)}{m_{ji}(x_i)}$ is that a straightforward implementation requires $N(i)(N(i)-1)$ multiplications which are prohibitively large. However, we decrease the number of multiplication to $2N(i)$ by first computing $t = \prod_{k \in N(i)} m_{ki}(s')$, and for each $j \in N(i)$ computing $t/m_{ji}(s')$.

The only remaining pieces of the computation is to incorporate the prior ϕ and the propagation matrix ψ. The propagation matrix ψ is a tiny bit of information, so it can be sent to every reducer by a variable passing functionality of HADOOP. The prior vector ϕ can be large, since the length of the vector can be the number of nodes in the graph. In the HADOOP algorithm, we also group the ϕ by the node id: each node prior is grouped together with the edges (messages) whose source id is the node id. Algorithm 5.4 shows the high-level algorithm of HADOOP LINE GRAPH FIXED POINT (HA-LFP). Algorithm 5.5 shows the BP message initialization algorithm which requires only a Map function. Algorithm 5.6 shows the HADOOP algorithm for the message update which implements the algorithm described above. After the messages converge, the final belief is computed by Algorithm 5.7.

Algorithm 5.4: HADOOP LINE GRAPH FIXED POINT (HA-LFP)

```
Input: edge $E$ of a undirected graph $G = (V, E)$,
    node prior $\phi^{n \times 1}$, and
    propagation matrix $\psi^{S \times S}$.
Output: belief matrix $b^{n \times S}$.
1: Initialization(); // Algorithm 5.5
2: while m does not converge do
3:     MessageUpdate(); // Algorithm 5.6
4: end while
5: BeliefComputation(); // Algorithm 5.7
```

Algorithm 5.5: HA-LFP Initialization

```
Input: edge $E = \{(id_{src}, id_{dst})\}$, and
    set of states $S = \{s_1, ..., s_p\}$.
Output: message matrix $M = \{(id_{src}, id_{dst}, m_{dst,src}(s_1), ..., m_{dst,src}(s_p))\}$.
1: Initialization-Map(Key $k$, Value $v$):
2: Output(($k, v$), ($\frac{1}{|S|}$, ..., $\frac{1}{|S|}$)); // ($k$: $id_{src}$, $v$: $id_{dst}$)
```

5.4.3 Analysis

We analyze the time and the space complexities of HA-LFP. The main result is that one iteration of the message update on the line graph has the same complexity as one matrix-vector multiplication on the original graph. In the lemma below, s is the number of machines.

Lemma 5.1 (Time Complexity of HA-LFP). *One iteration of* HA-LFP *takes* $O(\frac{|V|+|E|}{s} \log \frac{|V|+|E|}{s})$ *time. It could take* $O(\frac{|V|+|E|}{s})$ *time if* HADOOP *uses only hashing, not sorting, on its shuffling stage.*

Proof. Notice that the number of states is usually very small (2 or 3), thus can be considered as a constant. Assuming uniform distribution of data to machines, the time complexity is dominated by the Message-Update job. Thanks to the 'lazy multiplication' described in the previous section, both Map and Reduce takes linear time to the input. Thus, the time complexity is $O(\frac{|V|+|E|}{s} \log \frac{|V|+|E|}{s})$, which is the sorting time for $\frac{|V|+|E|}{s}$ records. It could be $O(\frac{|V|+|E|}{s})$, if HADOOP performs only hashing without sorting on its shuffling stage. □

For space complexity, we have the following result.

Lemma 5.2 (Space Complexity of HA-LFP). HA-LFP *requires* $O(|V| + |E|)$ *space.*

Proof. The prior vector requires $O(|V|)$ space, and the message matrix requires $O(2|E|)$ space. Since the number of edges is greater than the number of nodes, HA-LFP requires $O(|V| + |E|)$ space, in total. □

5.5 Experiments

In this section, we present experimental results to answer the following questions:

Algorithm 5.6: HA-LFP Message Update

```
Input: set of states $S = \{s_1, ..., s_p\}$,
    current message matrix $M^{cur} = \{(sid, did, m_{did,sid}(s_1), ..., m_{did,sid}(s_p))\}$,
    prior matrix $\Phi = \{(id, \phi_{id}(s_1), ..., \phi_{id}(s_p))\}$, and
    propagation matrix $\psi$.
Output: updated message matrix $M^{next} = \{(id_{src}, id_{dst}, m_{dst,src}(s_1), ..., m_{dst,src}(s_p))\}$.
1: MessageUpdate-Map(Key $k$, Value $v$):
2: if $(k, v)$ is of type $M$ then
3:     Output$(k, v)$; // ($k$: $sid$, $v$: $did, m_{did,sid}(s_1), ..., m_{did,sid}(s_p)$)
4: else if $(k, v)$ is of type $\Phi$ then
5:     Output$(k, v)$; // ($k$: $id$, $v$: $\phi_{id}(s_1), ..., \phi_{id}(s_p)$)
6: end if
7:
8: MessageUpdate-Reduce(Key $k$, Value $v[1..r]$):
9: $temps[1..p] \leftarrow [1..1]$;
10: $saved\_prior \leftarrow [\ ]$;
11: HashTable<int, double[1..$p$]> $h$;
12: for $z \in v[1..r]$ do
13:     if $(k, z)$ is of type $\Phi$ then
14:         $saved\_prior[1..p] \leftarrow z$;
15:     else if $(k, z)$ is of type $M$ then
16:         $(did, m_{did,sid}(s_1), ..., m_{did,sid}(s_p)) \leftarrow z$;
17:         $h.add(did, (m_{did,sid}(s_1), ..., m_{did,sid}(s_p)))$;
18:         for $i \in 1..p$ do
19:             $temps[i] = temps[i] \times m_{did,sid}(s_i)$;
20:         end for
21:     end if
22: end for
23: for $(did, (m_{did,sid}(s_1), ..., m_{did,sid}(s_p))) \in h$ do
24:     $outm[1..p] \leftarrow 0$;
25:     for $x \in 1..p$ do
26:         for $y \in 1..p$ do
27:             $outm[x] = outm[x] + saved\_prior[y]\psi(y, x)temps[y]/m_{did,sid}(s_v)$;
28:         end for
29:     end for
30:     Output$(did, (sid, outm[1], ..., outm[p]))$;
31: end for
```

Algorithm 5.7: HA-LFP Belief Computation

```
Input: set of states $S = \{s_1, ..., s_p\}$,
    current message matrix $M^{cur} = \{(sid, did, m_{did,sid}(s_1), ..., m_{did,sid}(s_p))\}$, and
    prior matrix $\Phi = \{(id, \phi_{id}(s_1), ..., \phi_{id}(s_p))\}$.
Output: belief vector $b = \{(id, b_{id}(s_1), ..., b_{id}(s_p))\}$.
1: BeliefComputation-Map(Key $k$, Value $v$):
2: if $(k, v)$ is of type $M$ then
3:     Output($k, v$); // ($k$: $sid$, $v$: $did, m_{did,sid}(s_1), ..., m_{did,sid}(s_p)$)
4: else if $(k, v)$ is of type $\Phi$ then
5:     Output($k, v$); // ($k$: $id$, $v$: $\phi_{id}(s_1), ..., \phi_{id}(s_p)$)
6: end if
7:
8: BeliefComputation-Reduce(Key $k$, Value $v[1..r]$):
9: $b[1..p] \leftarrow [1..1]$;
10: for $z \in v[1..r]$ do
11:     if $(k, z)$ is of type $\Phi$ then
12:         $prior[1..p] \leftarrow z$;
13:         for $i \in 1..p$ do
14:             $b[i] = b[i] \times prior[i]$;
15:         end for
16:     else if $(k, z)$ is of type $M$ then
17:         $(did, m_{did,sid}(s_1), ..., m_{did,sid}(s_p)) \leftarrow z$;
18:         for $i \in 1..p$ do
19:             $b[i] = b[i] \times m_{did,sid}(s_i)$;
20:         end for
21:     end if
22: end for
23: Output($k, (b[1], ..., b[p])$);
```

Graph	Nodes	Edges	File Size	Type	Description
YahooWeb	1.4 B	6.6 B	0.12 TB	real	WWW links in 2002
Twitter'10	104 M	3.7 B	0.13 TB	real	person-person
Twitter'09	63 M	1.8 B	56 GB	real	person-person
VoiceCall	30 M	260 M	8.4 GB	real	who calls whom
SMS	7 M	38 M	629 MB	real	who sends to whom
Kronecker	177 K	2 B	25 GB	synthetic	from Kronecker generator [Leskovec et al., 2005]
	121 K	1.1 B	13.9 GB		
	59 K	282 M	3.3 GB		

Table 5.2: Datasets. B: Billion, M: Million, K: Thousand, TB: Terabytes, GB: Gigabytes, MB: Megabytes.

Q1 How fast is HA-LFP, compared to a single-machine disk-based Belief Propagation algorithm?
Q2 How does HA-LFP scale up on the number of machines?
Q3 How does HA-LFP scale up on the number of edges?

We performed experiments in the M45 HADOOP cluster by Yahoo!. As mentioned earlier, the cluster has total 480 machines with 1.5 Petabytes total storage and 3.5 Terabytes memory. The single-machine experiment was done in a machine with 3 Terabyte of disk and 48 GB memory. The single-machine BP algorithm is a scaled-up version of a memory-based BP which reads all the nodes, not the edges, in the memory. That is, the single-machine BP loads only the node information into a memory, but it reads the edges sequentially from the disk for every message update, instead of loading all the edges into a memory once for all.

The graphs we used in our experiments at Section 5.5 and 5.6 are summarized in Table 5.2 whose entries are repeated from Table 2.2 for convenience..

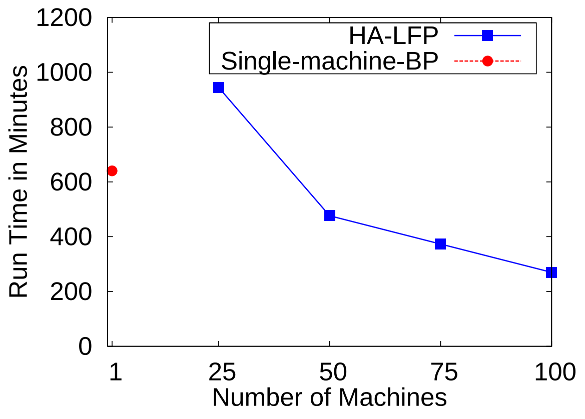
5.5.1 Results

Between HA-LFP and the single-machine BP, which one runs faster? At which point does the HA-LFP outperform the single-machine BP? Figure 5.2 (a) shows the comparison of running time of the HA-LFP and the single-machine BP. Notice that HA-LFP outperforms the single-machine BP when the number of machines exceeds 40. The HA-LFP requires more machines to beat the single-machine BP due to the fixed costs for writing and reading the intermediate results to and from the disk. However, for larger graphs whose nodes do not fit in the memory, HA-LFP is the only solution to the best of our knowledge.

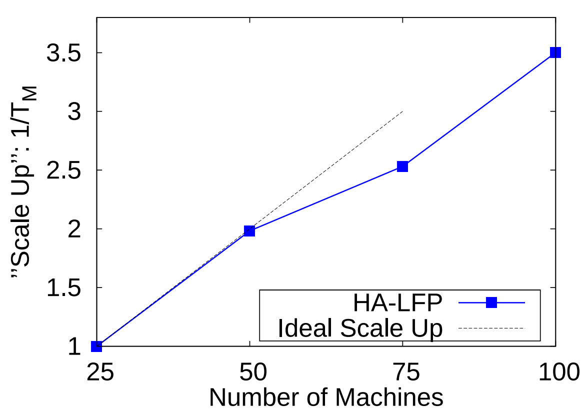
The next question is, how does our HA-LFP scale up on the number of machines and edges? Figure 5.2 (b) shows the scalability of HA-LFP on the number of machines. We see that our HA-LFP scales up linearly close to the ideal scale-up. Figure 5.3 shows the linear scalability of HA-LFP on the number of edges.

5.5.2 Discussion

Based on the experimental results, what are the advantages of HA-LFP? In what situations should it be used? For a small graph whose nodes and edges fit in the memory, the single-machine BP is recommended



(a) Running Time



(b) Scale-Up with Machines

Figure 5.2: Running time of HA-LFP with 10 iterations on the YahooWeb graph with 1.4 billion nodes and 6.7 billion edges. **(a):** Comparison of the running times of HA-LFP and the single-machine BP. Notice that HA-LFP outperforms the single-machine BP when the number of machines exceed ≈40. **(b):** "Scale-up" (throughput $1/T_M$) versus number of machines M, for the YahooWeb graph. Notice the near-linear scale-up close to the ideal (dotted line).

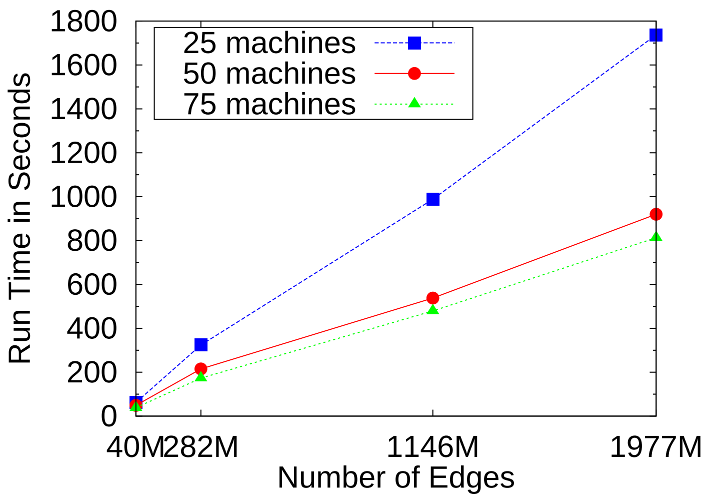


Figure 5.3: Running time of 1 iterations of message update in HA-LFP on Kronecker graphs. Notice that the running time scales-up linearly with the number of edges.

	Good	**Bad**
Good	1-ϵ	ϵ
Bad	0.5	0.5

Table 5.3: Edge potential (ψ matrix) for the YahooWeb. ϵ is set to 0.05 in the experiments. Good pages point to other good pages with high probability. Bad pages point to bad pages, but also good pages with equal chances, to boost their rank in Web search engines.

since it runs faster. For a medium-to-large graph whose nodes fit in the memory but the edges do not fit in the memory, HA-LFP gives the reasonable solution since it runs faster than the single-machine BP. For a very large graph whose nodes do not fit in the memory, HA-LFP is the only solution. We summarize the advantages of the HA-LFP here:

- **Scalability**: HA-LFP is *the only* solution when the nodes information can not fit in the memory. Moreover, HA-LFP scales up near-linearly.
- **Running Time**: even for a graph whose node information fits in the memory, HA-LFP ran 2.4 times faster.
- **Fault Tolerance**: HA-LFP enjoys the fault tolerance that HADOOP provides: data are replicated, and the failed programs due to machine errors are restarted in working machines.

5.6 Analysis of Real Graphs

In this section, we analyze real-world graphs using HA-LFP and show important findings.

5.6.1 HA-LFP on YahooWeb

Given a Web graph, how can we separate the educational ('good') Web pages from the adult ('bad') Web pages? Manually investigating billions of Web pages would take so much time and efforts. In this section, we show how to do it using HA-LFP. We use a simple heuristic to set priors: the Web pages which contain 'edu' have high goodness prior (0.95), and the Web pages which contain either 'sex', 'adult', or 'porno' have low goodness prior (0.05). Among 11.8 million Web pages containing sexually explicit keywords, we keep 10% of the pages as a validation set (goodness prior 0.5), and use the rest 90% as a training set by setting the goodness prior 0.05. Also, among 41.7 million Web pages containing 'edu', we randomly sample 11.8 million Web pages, so that the number equals with that of adult pages given prior, and use 10% as a validation set (goodness prior 0.5), and use the rest 90% as a training set (goodness prior 0.95). The edge potential function is given by Table 5.3. It is given by our observation that good pages tend to point to other good pages, while bad pages might point to good pages, as well as bad pages, to boost their ranking in Web search engines.

Figure 5.4 shows the HA-LFP scores and the number of pages in the test set having such scores. Notice that almost all the pages with LFP score less than 0.9 in our test data contain adult Web sites. Thus, the LFP score 0.9 can be used as a decision boundary for adult Web pages.

Figure 5.5 shows the HA-LFP scores vs. PageRank scores of pages in our test set. We see that the

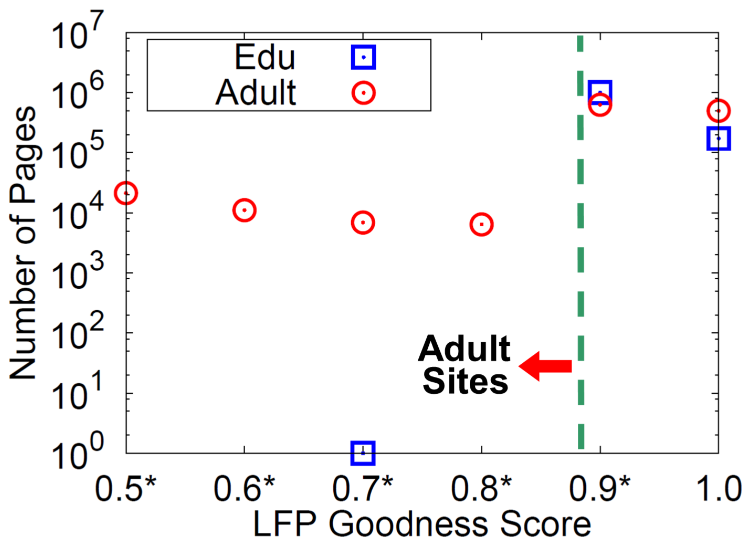


Figure 5.4: HA-LFP scores and the number of pages in the test set having such scores. Note that pages whose goodness scores are less than 0.9 (the left side of the vertical bar) are likely to be adult pages with very high chances.

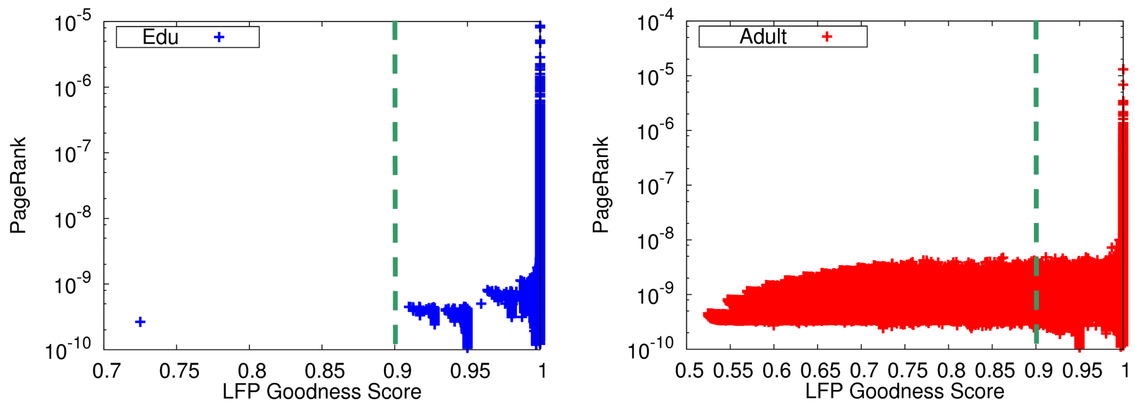


Figure 5.5: HA-LFP scores vs. PageRank scores of pages in our test set. The vertical dashed line is the same decision boundary as in Figure 5.4. Note that in contrast to HA-LFP, PageRank scores cannot be used to differentiating the good from the bad pages.

	Celebrity	**Spammer**	**Normal**
Celebrity	0.1	0.05	0.85
Spammer	0.1	0.45	0.45
Normal	0.35	0.05	0.6

Table 5.4: Edge potential (ψ matrix) for Twitter and VoiceCall.

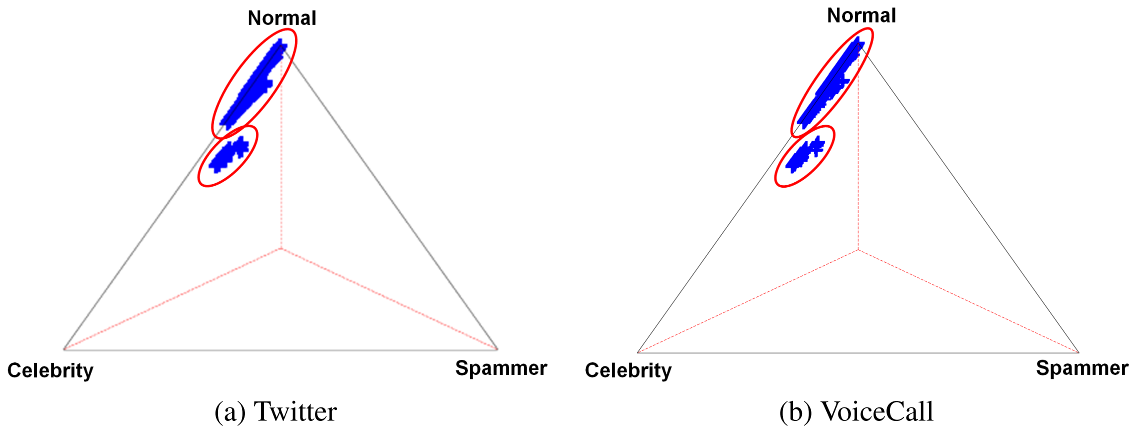


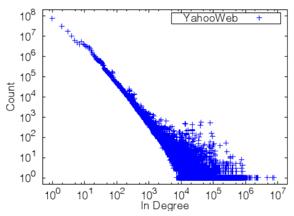
Figure 5.6: HA-LFP scores of people in Twitter and VoiceCall data. The points represent the scores of the final beliefs in each state, forming a simplex in 3-dimensional space whose axes are the red lines that meet at the center (origin). Notice that people seem to form two groups, in *both* datasets, despite the fact that the two datasets are completely of different nature.

PageRank cannot be used for differentiating between educational and adult Web pages. However, HA-LFP can be used to spotting adult Web pages, by using the threshold 0.9.

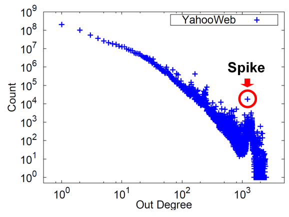
5.6.2 HA-LFP on Twitter and VoiceCall

We run HA-LFP on Twitter and VoiceCall data which are both social networks representing who follows whom or who calls whom. We define three roles: 'celebrity', 'spammer', and normal people. We define a celebrity as a person with a high in-degree ($\geq$1000), and not-too-large out-degree ($< 10 \times indegree$). We define a spammer as a person with a high out-degree ($\geq$1000), but low in-degree ($< 0.1 \times outdegree$). For celebrities, we set (0.1, 0.05, 0.85) for (celebrity, spammer, normal) prior probabilities. For spammers, we set (0.1, 0.45, 0.45) for (celebrity, spammer, normal) prior probabilities. The edge potential function is given by Table 5.4. It encodes our observation that celebrities tend to follow normal persons the most, spammers follow other spammers or normal persons, and normal persons follow other normal persons or celebrities.

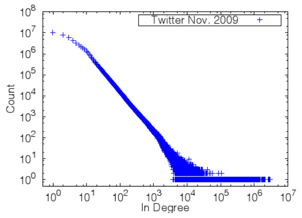
Figure 5.6 shows the HA-LFP scores of people in Twitter and VoiceCall data. There are two clusters in both of the data. The large cluster starting from the 'Normal' vertex contains high degree nodes, and the small cluster below the large cluster contains low degree nodes.



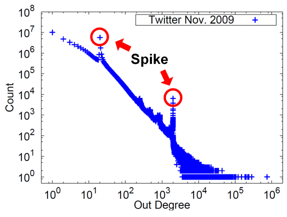
(a) YahooWeb: In Degree



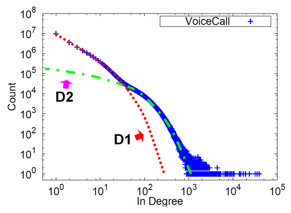
(b) Yahoo Web: Out Degree



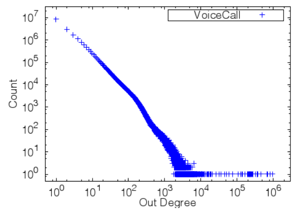
(c) Twitter: In Degree



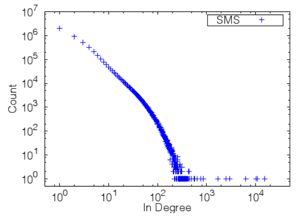
(d) Twitter: Out Degree



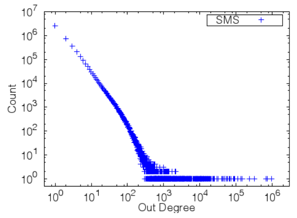
(e) VoiceCall: In Degree



(f) VoiceCall: Out Degree



(g) SMS: In Degree



(h) SMS: Out Degree

Figure 5.7: [(e): Best Viewed In Color] Degree distributions of real world graphs. Notice many high in-degree or out-degree nodes which can be used to determine the classes for HA-LFP. Most distributions follow power-law or lognormal, except (e) which seems to be a mixture of two lognormal distributions. Also notice the several spikes which suggest anomalous nodes, suspicious activities, or software limits on the number of connections.

5.6.3 Finding Roles and Anomalies

In the experiments of previous sections, we used several classes ('bad' Web sites, 'spammers', 'celebrities', etc.) of nodes. The question is, how can we find classes of a given graph? Finding out such classes is important for BP since it helps to set reasonable priors which could lead to quick convergence. In this section, we analyze real world graphs and give observations on the patterns and anomalies, which could potentially help determine the classes. We focus on the structural properties of graphs, including degree, connected component, and radius.

Using Degree Distributions. We first show the degree distributions of real world graphs in Figure 5.7. Notice that there are nodes with very high in or out degrees, which gives valuable information for setting priors.

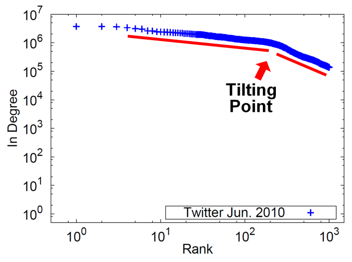
Observation 10 (High In or Out Degree Nodes)**.** *The nodes with high in-degree can have a high prior for 'celebrity', and the nodes with high out-degree but low in-degree can have a high prior for 'spammer'.*

Most of the degree distributions in Figure 5.7 follow power law or log-normal. The VoiceCall in degree distribution (Figure 5.7 (e)) is different from other distributions since it contains mixture of distributions:

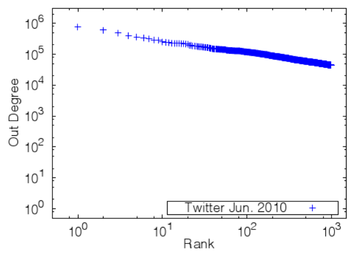
Observation 11 (Mixture of Lognormals in Degree Distribution)**.** *VoiceCall in degree distributions in Figure 5.7 seems to comprise two lognormal distributions shown in D1 (red color) and D2 (green color).*

Another observation is that there are several anomalous spikes in the degree distributions in Figures 5.7 (b) and (d).

Observation 12 (Spikes in Degree Distribution)**.** *There is a huge spike at the out degree 1200 of YahooWeb data in Figure 5.7 (b). They came from online market pages from Germany, where the pages are linked to each other and forming link farms. Two outstanding spikes are also observed at the out degree 20 and*



(a) In degree vs. Rank



(b) Out degree vs. Rank

Figure 5.8: Degree vs. Rank. in Twitter Jun. 2010 data. Notice the change of slope around the tilting point in (a). The point can be used to distinguish super-celebrities (e.g., of international caliber) versus plain celebrities (of national or regional caliber).

2001 of Twitter data in Figure 5.7 (d). The reason seems to be a hard limit in the maximum number of people to follow.

Finally, we study the highest degrees that are beyond the power-law or lognormal cutoff points using the rank plot. Figure 5.8 shows the top 1000 highest in and out degrees and its rank (from 1 to 1000) which we summarize in the following observation.

Observation 13 (Tilt in Rank Plot)**.** *The out degree rank plot of Twitter data in Figure 5.8 (b) follows a power law with a single exponent. The in degree rank plot, however, comprises two fitting lines with a tilting point around rank 240. The tilting point divides the celebrities in two groups: super-celebrities (e.g., possibly, of international caliber) and plain celebrities (possibly, of national or regional caliber).*

Using Connected Component Distributions. The distributions of the sizes of connected components in a graph informs us of the connectivity of the nodes (component size vs. number of components having that size). When these distributions are plotted over time, we may observe when certain nodes participate in various activities — patterns such as periodicity or anomalous deviations from such patterns can generate important insights.

Observation 14 (Periodic Dips and Surges)**.** *Figure 5.9 shows the temporal connected component distribution of the VoiceCall (who-calls-whom) data, where each data point was computed using one day's worth of data (i.e., a one-day snapshot). On every Sunday, we see a dip in the size of the giant connected component (largest component), and an accompanying surge in the number of connected components for the day. This periodicity highlights the typical and rather constant call volume during the work days, and lower volume outside them. Equipped with this information, we may infer that "business" phone numbers (nodes) are those that are regularly active during work days but not weekends; we may in turn characterize these "business" numbers as one class of nodes in our algorithm. The sizes of the second and third largest component oscillate about some small numbers (68 and 50 respectively), echoing previous research findings [Mcglohon et al., 2008].*

Using Radius Distributions. We next analyze the (effective) radius distributions of real graphs. As defined in Definition 1 of Section 3.2.1, effective radius of a node is defined to be the 90%th percentile of all the distances to other nodes from it. Thus, nodes with low radii can reach other nodes in a small number of steps. Figure 5.10 shows the radius distributions of real graphs. In contrast to the VoiceCall and the SMS data, the Twitter data contains several anomalous nodes with long (>10) radii.

Observation 15 (Suspicious Accounts Created By A User)**.** *The Twitter data contain several nodes with long radii. They form chains shown in Figure 5.11. Each chain seems to be created by one user, since the*

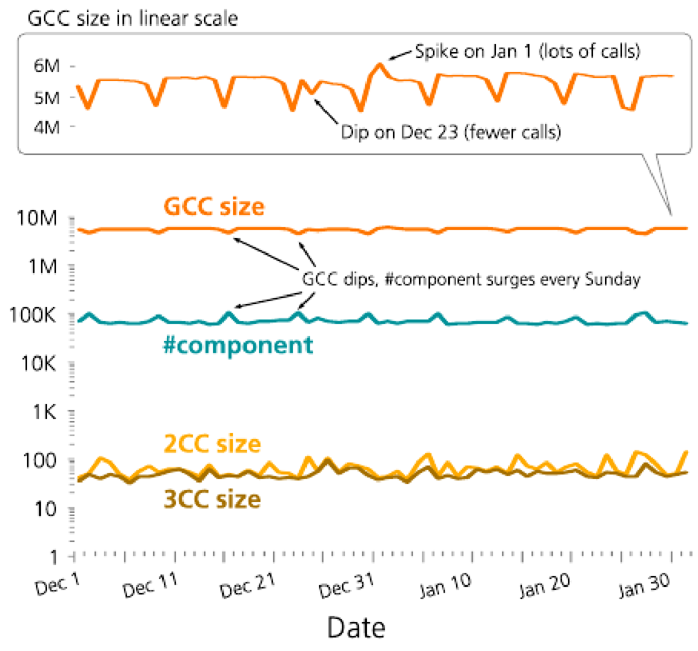
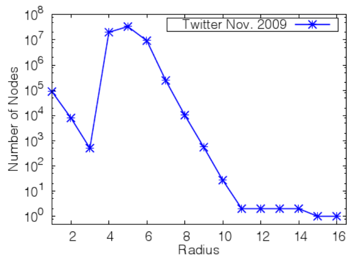
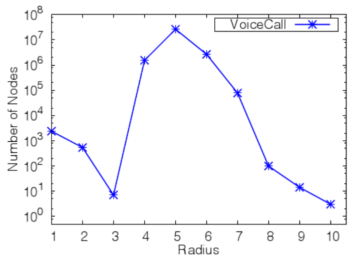


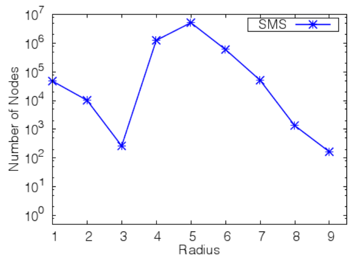
Figure 5.9: **[Best Viewed In Color]** Temporal connected component distributions of the VoiceCall data, from Dec 1, 2007 to Jan 31, 2008, inclusively. Each data point computed using one day's worth of data (i.e., a one-day snapshot.) GCC, 2CC, and 3CC are the first (giant), second, and third largest components respectively. The turquoise line denotes the number of connected components. The temporal trend may be used to set priors for HA-LFP. See the text for details.



(a) Twitter



(b) VoiceCall



(c) SMS

Figure 5.10: Radius distributions of real world graphs. Notice the nodes with long radius in the Twitter data. They are usually suspicious nodes as described in Figure 5.11.

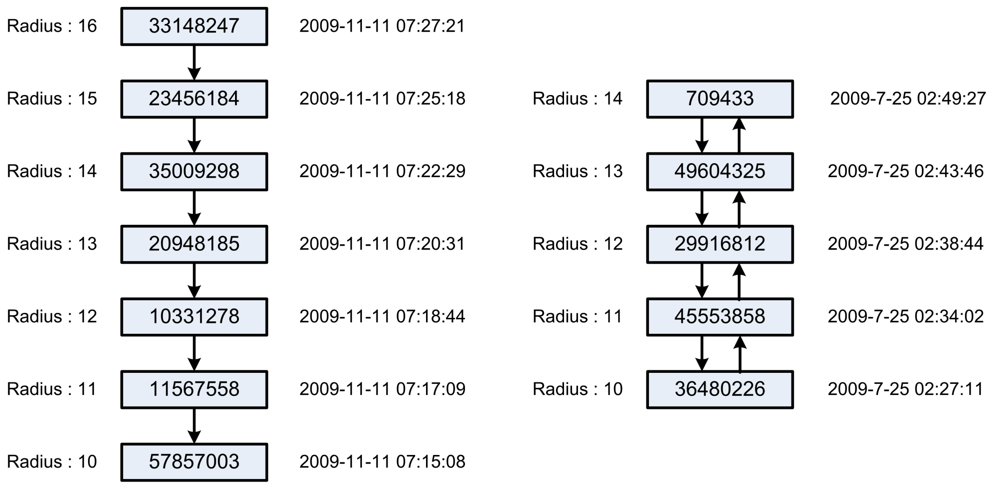


Figure 5.11: Accounts with long radii in the Twitter Nov. 2009 data. Each box represents an account with the corresponding anonymized id. At the right of the boxes, the time that the account was created is shown. All the accounts are suspicious since they form chains with very low degree. They seem to be created from a user, based on the regular timestamps. Especially, all the 7 accounts in the left figure are from Mumbai, India.

times in which accounts are created are regular.

5.7 Conclusion

In this chapter we propose HADOOP LINE GRAPH FIXED POINT (HA-LFP), a HADOOP algorithm for the inferences of graphical models in billion-scale graphs. The main contributions are the following:

- **Efficiency.** We show that the solution of inference problem in graphical models is a fixed point in line graph. We propose LINE GRAPH FIXED POINT (LFP), a formulation of BP on a line graph induced from the original graph, and show that it is a generalized version of a linear algebra operation. We propose HADOOP LINE GRAPH FIXED POINT (HA-LFP), an efficient algorithm carefully designed for LFP in HADOOP.

- **Scalability.** We do the experiments to compare the running time of the HA-LFP and a single-machine BP. We also gives the scalability results and show that HA-LFP has a near-linear scale up.
- **Effectiveness.** We show that our method can find interesting patterns and anomalies, on some of the largest publicly available graphs.

Chapter 6

Spectral Graph Analysis

Given a graph with billions of nodes and edges, are there nodes that participate in too many or too few triangles? Are there close-knit near-cliques? These questions are expensive to answer unless we have the first several eigenvalues and eigenvectors of the graph adjacency matrix. However, eigensolvers suffer from subtle problems (e.g., convergence) for large sparse matrices, let alone for billion-scale ones.

We address this problem with the proposed HEIGEN algorithm, which we carefully design to be accurate, efficient and able to run on the highly scalable MAPREDUCE (HADOOP) environment. This enables HEIGEN to handle matrices more than *1000*$\times$ larger than those which can be analyzed by existing algorithms. We implement HEIGEN and run it on the M45 cluster. We report important discoveries about near-cliques and triangles on several real-world graphs.

6.1 Introduction

Given a billion-scale graph, how can we find near-cliques (a set of tightly connected nodes), the count of triangles, and related graph properties? As we discuss later, triangle counting and related expensive operations can be computed quickly, provided we have the first several eigenvalues and eigenvectors. In general, spectral analysis is a fundamental tool not only for graph mining, but also for other areas of data mining. Eigenvalues and eigenvectors are at the heart of numerous algorithms such as triangle counting [Tsourakakis, 2008], Singular Value Decomposition (SVD) [Kamel, 1984, Berry., 1992], spectral clustering [Shi and Malik, 1997, Ng et al., 2002, Luxburg, 2007], Principal Component Analysis (PCA) [Pearson, 1901], Multi Dimensional Scaling (MDS) [Kruskal and Wish, 1978, Bartell et al., 1992], Latent Semantic Indexing (LSI) [Deerwester et al., 1990], and tensor analysis [Sun et al., 2006b, Kolda and Sun, 2008, Kolda and Bader, 2009, Dunlavy et al., 2011]. Despite their importance, existing eigensolvers do not scale well. As described in Section 6.7, the maximum order and size of input matrices feasible for these solvers are in the order of millions.

In this chapter, we discover patterns on near-cliques and triangles, on several real-world graphs including a Twitter dataset (*56Gb*, 1.8 billion edges) and the "YahooWeb" dataset, one of the largest publicly available graphs (*120Gb*, 1.4 billion nodes, 6.6 billion edges). To enable discoveries, we propose HEIGEN, an eigensolver for billion-scale, sparse symmetric matrices built on the top of HADOOP, an open-source MAPREDUCE framework. Our contributions are the following:

Graph	Nodes	Edges	File Size	Type	Description
YahooWeb	1.4 B	6.6 B	0.12 TB	real	WWW links in 2002
Twitter'09	63 M	1.8 B	56 GB	real	person-person
LinkedIn	7.5 M	58 MB	1 GB	real	person-person in 2006
Wikipedia	3.5 M	42 M	605 MB	real	doc-doc in 2007/02
WWW-Barabasi	325 K	1.5 M	20 MB	real	WWW links in nd.edu
Epinions	75 K	508 K	5 MB	real	who trusts whom
Kronecker	177 K	2 B	25 GB	synthetic	from Kronecker generator [Leskovec et al., 2005]
	121 K	1.1 B	13.9 GB		
	59 K	282 M	3.3 GB		

Table 6.1: Datasets. B: Billion, M: Million, K: Thousand, TB: Terabytes, GB: Gigabytes, MB: Megabytes.

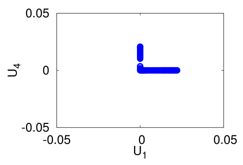
1. **Effectiveness.** With HEIGEN we analyze billion-scale real-world graphs and report discoveries, including a high triangle vs. degree ratio for adult sites and Web pages that participate in billions of triangles.
2. **Careful Design.** We choose among several serial algorithms and selectively parallelize operations for better efficiency.
3. **Scalability.** We use the HADOOP platform for its excellent scalability and implement several optimizations for HEIGEN, such as cache-based multiplications and skewness exploitation. This results in linear scalability in the number of edges, the same accuracy as standard eigensolvers for small matrices, and more than a 76× performance improvement over a naive implementation.

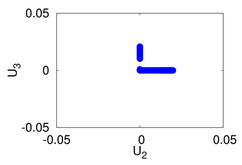
Due to our focus on scalability, HEIGEN can handle sparse symmetric matrices which correspond to graphs with *billions of nodes and edges*, surpassing the capability of previous eigensolvers (e.g. [Wu and Simon, 1999, Song et al., 2008]) by more than *1,000*×. Note that HEIGEN is different from Google's PageRank algorithm [Brin and Page, 1998] since HEIGEN computes the top k eigenvectors while PageRank computes only the *first* eigenvector. Designing top k eigensolver is much more difficult and subtle than designing the first eigensolver, as we will see in Section 6.4. With this powerful tool we are able to study several billion-scale graphs, and we report fascinating patterns on the near-cliques and triangle distributions in Section 6.2.

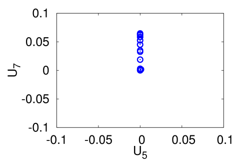
The HEIGEN algorithm (implemented in HADOOP) is available at
`http://www.cs.cmu.edu/~pegasus`. The rest of the chapter is organized as follows. In Section 6.2 we presents the discoveries in real-world, large scale graphs. Section 6.3 explains the design decisions that we considered for selecting the best sequential method. Section 6.4 describes HEIGEN, our proposed eigensolver. Section 6.5 explains additional uses of HEIGEN for interesting eigenvalue based algorithms. Section 6.6 shows the performance results of HEIGEN. After describing previous works in Section 6.7, we conclude in Section 6.8.

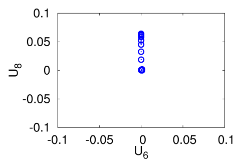
6.2 Discoveries

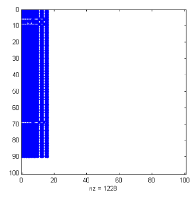
In this section, we show discoveries on billion-scale graphs using HEIGEN. The discoveries include spotting near-cliques, finding triangles, and eigen power-laws. The graphs we used in this and Section 6.6 are described in Table 6.1 whose entries are repeated from Table 2.2 for convenience.


(a) U_4 vs. U_1

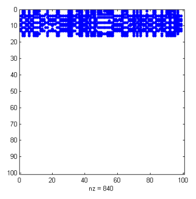

(b) U_3 vs. U_2


(c) U_7 vs. U_5

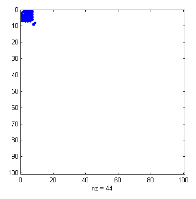

(d) U_8 vs. U_6



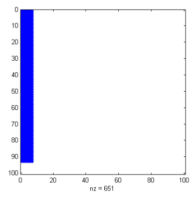
(e) U_1 spoke



(f) U_2 spoke



(g) U_3 spoke



(h) U_4 spoke

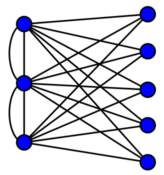

(i) Structure of bi-clique

Figure 6.1: EE-plots and spyplots from YahooWeb. **(a)-(d)**: EE-plots showing the scores of nodes in the ith eigenvector U_i vs. in the jth eigenvector U_j. Notice the clear 'spokes' in top eigenvectors signify the existence of a strongly related group of nodes in near-cliques or bi-cliques as depicted in (i). **(e)-(h)**: spyplots (adjacency matrices of induced subgraphs) of the top 100 largest scoring nodes from each eigenvector. Notice that we see a near clique in U_3, and bi-cliques in U_1, U_2, and U_4. **(i)**: the structure of 'bi-clique' in (e), (f), and (h).

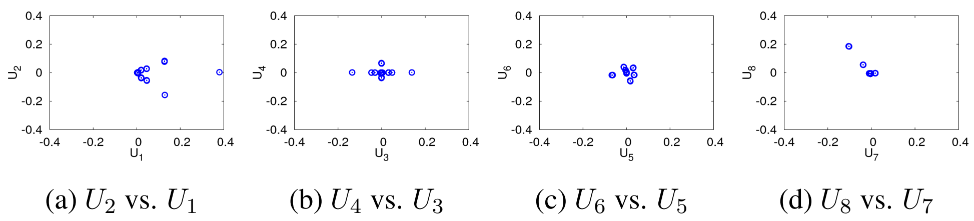
6.2.1 Spotting Near-Cliques

In a large, sparse network, how can we find tightly connected nodes, such as those in near-cliques or bipartite cores? Surprisingly, eigenvectors can be used for this purpose [Prakash et al., 2010a]. Given an adjacency matrix A and its SVD $A = U\Sigma V^T$, an *EE-plot* is defined to be the scatter plot of the vectors U_i and U_j for any i and j. EE-plots of some real-world graphs contain clear separate lines (or 'spokes'), and the nodes with the largest values in each spoke distinguish themselves from the other nodes by forming near-cliques or bipartite cores. Figures 6.1 shows several EE-plots and spyplots (i.e., adjacency matrix of induced subgraph) of the top 100 nodes in top eigenvectors of YahooWeb graph.

In Figure 6.1 (a) - (d), we observe clear 'spokes,' or outstanding nodes, in the top eigenvectors. Moreover, the top 100 nodes with largest values in U_1, U_2, and U_4 form a 'bi-clique', shown in (e), (f), and (h), which is defined to be the combination of a clique and a bipartite core as depicted in Figure 6.1 (i). Another observation is that the top seven nodes shown in Figure 6.1 (g) belong to `indymedia.org` which is the site with the maximum number of triangles as shown in Figure 6.3.

In the WWW-Barabasi graph of Figure 6.2, we also observe spokes in the top eigenvectors. The spokes from the top four eigenvectors form near-cliques, and the union of them (329 nodes) clearly identify three tightly connected communities in Figure 6.2 (i).

Observation 16 (Eigenspokes)**.** *EE-plots of real graphs show clear spokes. Additionally, the extreme nodes in the spokes belong to cliques or bi-cliques.* □


(a) U_2 vs. U_1 (b) U_4 vs. U_3 (c) U_6 vs. U_5 (d) U_8 vs. U_7

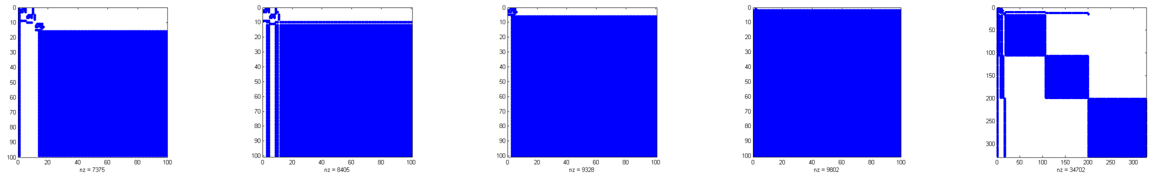
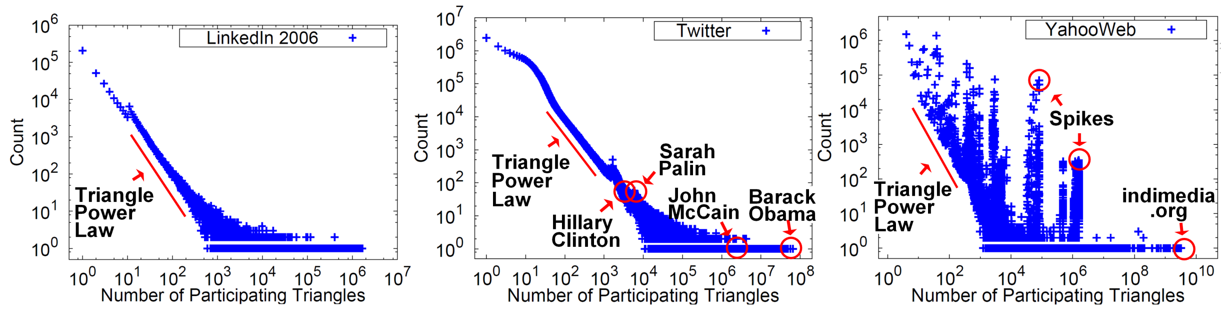

(e) U1 spoke (f) U2 spoke (g) U3 spoke (h) U4 spoke (i) U1-U4 spyplot

Figure 6.2: EE-plots and spyplots of WWW-Barabasi. **(a)-(d)**: EE-plots showing the scores in the ith eigenvector U_i vs. in the jth eigenvector U_j. Notice the 'spokes' in top eigenvectors which signify the cliques shown in the second row. **(e)-(h)**: Spyplots (adjacency matrices of induced subgraphs) from the top 100 largest scoring nodes from each eigenvector. Notice the cliques in all the plots. **(i)**: Spyplots of the union of nodes in the top 4 spokes. Notice the 3 cliques of sizes 90, 100, and 130.



(a) LinkedIn (58M edges) (b) Twitter (2.8B edges) (c) YahooWeb (6.6B edges)

Figure 6.3: The distribution of the number of participating triangles of real graphs. In general, they obey the "triangle power-law." Moreover, well-known U.S. politicians participate in many triangles, demonstrating that their followers are well-connected. In the YahooWeb graph, we observe several anomalous spikes which possibly come from cliques.

6.2.2 Triangle Counting

Given a particular node in a graph, how are its neighbors connected? Do they form stars? Cliques? The above questions about the community structure of networks can be answered by studying triangles (three nodes which are connected to each other). However, directly counting triangles in graphs with billions of nodes and edges is prohibitively expensive [Tsourakakis et al., 2009]. Fortunately, we can approximate triangle counts with high accuracy using HEIGEN by exploiting the connection of triangle counting to eigenvalues [Tsourakakis, 2008]. In a nutshell, the total number of triangles in a graph is related to the sum of cubes of eigenvalues, and the first few eigenvalues provide extremely good approximations. A slightly more elaborate analysis approximates the number of triangles in which a node participates, using the cubes of the first few eigenvalues and the corresponding eigenvectors. Specifically, the total number of triangles $\Delta(G)$ of a graph G is $\Delta(G) = \frac{1}{6}\sum_{i=1}^{n}\lambda_i^3$, and the number of triangles Δ_i that a node i is participating in is $\Delta_i = \frac{1}{2}\sum_{j=1}^{n}\lambda_j^3 u_j[i]^2$ where λ_j is the jth eigenvalue and $u_j[i]$ is the ith element of the jth eigenvector of the adjacency matrix of G. The top k eigenvalues can give highly accurate approximations to the number of triangles since the top eigenvalues dominate the cubic sum given the power-law relation of eigenvalues [Faloutsos et al., 1999], which we also observe in Section 6.2.3.

Using the top k eigenvalues computed with HEIGEN, we analyze the distribution of triangle counts of real graphs including the LinkedIn, the Twitter, and the YahooWeb graphs in Figure 6.3. We first observe that there exist several nodes with extremely large triangle counts. In Figure 6.3 (b), Barack Obama is the person with the fifth largest number of participating triangles, and has many more than other U.S. politicians. In Figure 6.3 (c), the Web page `lists.indymedia.org` contains the largest number of triangles; this page is a list of mailing lists which apparently point to each other.

We also observe regularities in triangle distributions and note that the beginning part of the distributions follows a power-law.

Observation 17 (Triangle power law). *The beginning part of the triangle count distribution of real graphs follows a power-law.* □

In the YahooWeb graph in Figure 6.3 (c), we observe many spikes. One possible explanation for these spikes is that they come from cliques: a k-clique generates k nodes with $\binom{k-1}{2}$ triangles.

Observation 18 (Spikes in triangle distribution). *In the Web graph, there exist several spikes which possibly come from cliques.* □

The rightmost spike in Figure 6.3 (c) contains 125 Web pages each of which has about 1 million triangles in their neighborhoods. They all belong to the news site `ucimc.org`, and are connected to a tightly coupled group of pages.

Triangle counts exhibit even more interesting patterns when combined with the degree information as shown in the triangle-degree plot of Figure 6.4. We see that celebrities have high degree and mildly connected followers, while accounts for adult sites have significantly fewer, but extremely well connected, followers. Degree-triangle plots can be used to spot and eliminate harmful accounts such as those of adult advertisers and spammers.

Observation 19 (Anomalous Triangles vs. Degree Ratio). *In Twitter, anomalous accounts have very high triangles vs. degree ratio compared to other regular accounts.* □

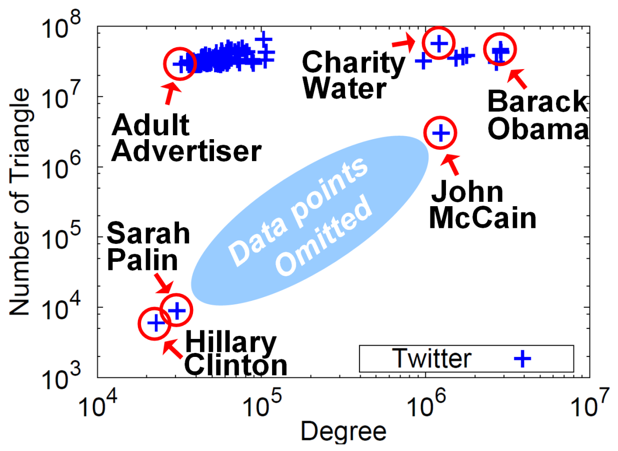


Figure 6.4: The number of participating triangles vs. degree of some 'celebrities' (rest: omitted, for clarity) in Twitter accounts. Also shown are accounts of adult sites which have smaller degree, but belong to an abnormally large number of triangles (= many, well connected followers - probably, 'robots').

6.2.3 Eigen Exponent

The power-law relationship $\lambda_i \propto i^{\varepsilon}$ of eigenvalues λ_i vs. rank i has been observed in Internet topology graphs with up to 4,389 nodes [Faloutsos et al., 1999]. Will the same power-law be observed in up to 300,000$\times$ larger graphs? The scree plots in Figure 6.5 show the answer. Note that all plots have correlation coefficients equal to -0.94 or better, except for the WWW-Barabasi with the correlation coefficient -0.84.

Observation 20 (Power-law scree plots)**.** *For all real graphs of Figure 6.5, the scree plots indicate power laws. Most of the graphs have the correlation coefficients -0.94 or better.* □

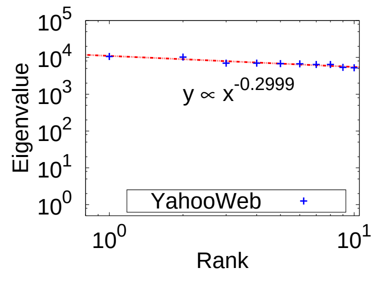
The difference of the slopes between YahooWeb and WWW-Barabasi graphs means that the larger Web graph (YahooWeb) has a more skewed eigenvalue distribution, with the small set of eigenvalues dominating most of the spectra compared to the smaller Web graph (WWW-Barabasi).

All of the above observations need a fast, scalable eigensolver. This is exactly what HEIGEN does, and we describe our proposed design next.

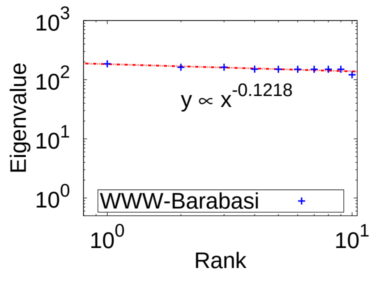
6.3 Background - Sequential Algorithms

Our goal is an eigensolver that finds the top k eigenvalues of a billion-scale matrix. Our natural choice of parallel platform is HADOOP, as described in Chapter 2. We limit our attention to symmetric matrices due to the computational difficulty since even the best method for non-symmetric eigensolver requires significantly heavier computations than the symmetric case [Trefethen and Bau III, 1997].

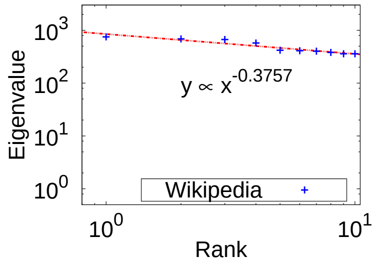
The problem of finding the eigenvalues of a matrix, however, is inherently difficult since it essentially boils down to finding the roots of a high-degree polynomial which may not have the general solution. Designing the parallel eigensolver algorithm is even more complicated since it requires a careful choice of operations that could be performed well in parallel. In this section, we review some of the major



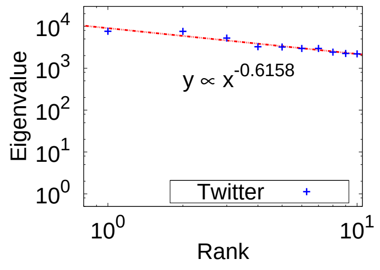
(a) YahooWeb
Corr. Coefficient: -0.943



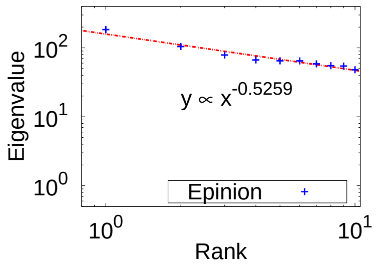
(b) WWW-Barabasi
Corr. Coefficient: -0.842



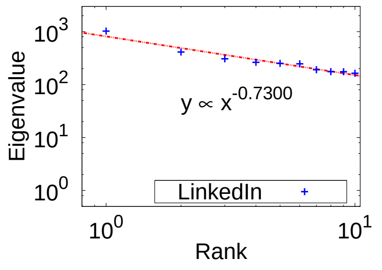
(c) Wikipedia
Corr. Coefficient: -0.946



(d) Twitter
Corr. Coefficient: -0.961



(e) Epinions
Corr. Coefficient: -0.974



(f) LinkedIn
Corr. Coefficient: -0.972

Figure 6.5: The scree plot: absolute eigenvalue vs. rank in log-log scale. Notice that the similar power-laws are observed in various real graphs.

Symbol	**Definition**
n	order of input matrix
m	number of iterations in Lanczos
A, M	n-by-n input matrix
A_s, M_s	n-by-m input matrix, $n \gg m$
y, x	n-vector
x_s	m-vector, $n \gg m$
α, β	a real number
$\|\|y\|\|$	L2 norm of the vector y
$\|\|T\|\|$	induced matrix L2 norm of the matrix T which is the largest singular value of T
e_m	a vector whose mth element is 1, while other elements are 0
$EIG(A)$	outputs QDQ^T by symmetric eigen decomposition
ϵ	machine epsilon: upper bound on the relative computation error

Table 6.2: Table of symbols.

sequential eigensolver algorithms and show the important design decisions that guided our choice of the best sequential method for parallel eigensolvers for very large graphs. Table 6.2 lists the symbols used in this chapter. For indexing elements of a matrix, we use $A[i, j]$ for (i, j)th element of A, $A[i, :]$ for ith row of A, and $A[:, j]$ for jth column of A.

6.3.1 Power Method

The simplest and probably most popular way of finding the *first* eigenvector of a matrix is the Power method. The *first* eigenvector is simply the eigenvector corresponding to the largest eigenvalue of the matrix. In the Power method, the input matrix A is multiplied with the initial random vector b multiple times to compute the sequence of vectors $Ab, A(Ab), A(A^2b), ...$ which converges to the first eigenvector of A. The intuition behind the power method is as follows: consider the eigendecomposition of $A = U\Lambda U^T$. If we take A^2 we essentially get

$$A^2 = (U\Lambda U^T)(U\Lambda U^T) = U\Lambda^2 U^T$$

because $U^TU = I$, by definition. At the k-th iteration of the Power method, we eventually get all the eigenvalues lifted to the power k; as k grows, the largest eigenvalue begins to dominate the rest, smaller eigenvalues, therefore, A^kb, with some appropriate scaling, will be a very good estimate of the first eigenvector of A: actually, $A^kb \approx u_1u_1^Tb\lambda_1^k$, where u_1 is the first column of U, and λ_1 is the largest eigenvalue. This constitutes no proof of the Power method, but merely an intuitive explanation of why it works.

The Power method is attractive since it requires only matrix-vector multiplications, which may be carried out efficiently in many parallel platforms including HADOOP. Furthermore, it is one of the ways of computing the PageRank of a graph [Brin and Page, 1998]. However, the main drawback of the Power method in the present context is that it is very restrictive, since it computes only the first eigenvector. Other variants of the power method, such as *shifted inverse iteration* and *Rayleigh quotient iteration* also have the same limitation. Therefore, we need to find a better method which can find top k eigenvalues and eigenvectors.

Shortcomings: Power method computes only the first eigenvector.

6.3.2 Simultaneous Iteration (or QR algorithm)

The simultaneous iteration (or QR algorithm, which is essentially the same) is an extension of Power method in the sense that it applies the Power method to several vectors at once. It can be shown that the orthogonal bases of the vectors converge to top k eigenvectors of the matrix [Trefethen and Bau III, 1997]. The main problem of the simultaneous iteration is that it requires several large matrix-matrix multiplications which are prohibitively expensive for billion-scale graphs. Therefore, we restrict our attention to algorithms that require only matrix-vector multiplications.

Shortcomings: Simultaneous iteration is too expensive for billion-scale graphs due to large matrix-matrix multiplications.

6.3.3 Lanczos-NO: No Orthogonalization

The next method we consider is the basic Lanczos algorithm [Lanczos, 1950] which we henceforth call Lanczos-NO (No Orthogonalization). Lanczos-NO method is attractive since it can find the top k eigenvalues of sparse, symmetric matrix, with its most costly operation being the matrix-vector multiplication.

Overview - Intuition. The Lanczos-NO algorithm is a clever improvement over the Power method. Like the Power method,

- it requires several (m) matrix-vector multiplications, that can easily be done with HADOOP
- then it generates a dense, but skinny matrix ($n \times m$, with $n \gg m$)
- it computes a small, sparse square $m \times m$ matrix, whose eigenvalues are good approximations to the required eigenvalues
- and then computes the top k eigenvectors ($k < m$), also with HADOOP-friendly operations.

Thus, all the expensive steps can be easily done with HADOOP. Next we provide more details on Lanczos-NO, which can be skipped on first glance.

Details. The Lanczos-NO algorithm is a clever extension of the Power method. In the Power method, the intermediate vectors $A^k b$ are discarded for the final eigenvector computation. In Lanczos-NO, the intermediate vectors are used for constructing orthonormal bases of the so-called *Krylov subspace* K_m, which is defined as

$$K_m = < b, Ab, ..., A^{m-1}b > .$$

The orthonormal bases are constructed by creating a new vector which is orthogonal to all previous bases, as in Gram-Schmidt orthogonalization. Therefore, Lanczos-NO can be summarized as an iterative algorithm which constructs orthonormal bases for successive Krylov subspaces. Specifically, Lanczos-NO with m iterations computes the Lanczos-NO factorization which is defined as follows:

$$AV_m = V_m T_m + f_m e_m^T,$$

where $A^{n\times n}$ is the input matrix, $V_m^{n\times m}$ contains the m orthonormal bases as its columns, $T_m^{m\times m}$ is a tri-diagonal matrix that contains the coefficients for the orthogonalization, f_m is a new n-vector orthogonal to all columns of V_m, and e_m is a vector whose mth element is 1, while other elements are 0. Here, m (the number of matrix-vector multiplication) is much smaller than n (the order of the input matrix): e.g., for billion-scale graphs, $n = 10^9$, and $m = 20$. The Lanczos-NO iteration is shown in Algorithm 6.1.

After m iterations, the V_m matrix and T_m matrices are constructed (T_m is built by $T_m[i,i] \leftarrow \alpha_i$, and $T_m[i,i+1] = T_m[i+1,i] \leftarrow \beta_i$). The eigenvalues of T_m are called the Ritz values, and the columns of $V_m Y$, where Y contains the eigenvector of T_m in its columns, are called the Ritz vectors which are constructed by Algorithm 6.2. The Ritz values and the Ritz vectors are good approximations of the eigenvalues and the eigenvectors of A, respectively [Golub and Van Loan, 1996]. The computation of the eigenvalues of T_m can be done quickly with direct algorithms such as QR since the matrix is very small (e.g., 20 by 20). For details, see [Golub and Van Loan, 1996].

The problem of Lanczos-NO is that some eigenvalues jump up to the next eigenvalues, thereby creating spurious eigenvalues. We will see the solution to this problem in the next section.

Algorithm 6.1: Lanczos-NO (No Orthogonalization)

```
Input: matrix $A^{n\times n}$,
    random $n$-vector $b$, and
    number of steps $m$.
Output: orthogonal matrix $V_m^{n\times m} = [v_1 ... v_m]$, and
    coefficients $\alpha[1..m]$, $\beta[1..m-1]$.
1: $\beta_0 \leftarrow 0$, $v_0 \leftarrow 0$, $v_1 \leftarrow b/||b||$;
2: for $i = 1, ..m$ do
3:   $v \leftarrow Av_i$;
4:   $\alpha_i \leftarrow v_i^T v$;
5:   $v \leftarrow v - \beta_{i-1}v_{i-1} - \alpha_i v_i$; // make a new basis
6:   $\beta_i \leftarrow ||v||$;
7:   if $\beta_i = 0$ then
8:     break for loop;
9:   end if
10:  $v_{i+1} \leftarrow v/\beta_i$;
11: end for
```

Algorithm 6.2: Compute Top k Ritz Vectors

```
Input: orthogonal matrix $V_m^{n\times m}$, and
    coefficients $\alpha[1..m]$, $\beta[1..m-1]$.
Output: Ritz vector $R_k^{n\times k}$.
1: $T_m \leftarrow$ (build a tri-diagonal matrix from $\alpha$ and $\beta$);
2: $QDQ^T \leftarrow EIG(T_m)$;
3: $\lambda_{1..k} \leftarrow$ (top $k$ eigenvalues from $D$);
4: $Q_k \leftarrow$ ($k$ columns of $Q$ corresponding to $\lambda_{1..k}$);
5: $R_k \leftarrow V_m Q_k$;
```

Shortcomings: Lanczos-NO outputs spurious eigenvalues.

6.4 Proposed Method

In this section we describe HEIGEN, a parallel algorithm for computing the top k eigenvalues and eigenvectors of symmetric matrices in MAPREDUCE.

6.4.1 Summary of the Contributions

Efficient top k eigensolvers for billion-scale graphs require careful algorithmic considerations. The main challenge is to carefully design algorithms that work well on distributed systems and exploit the inherent structure of data, including block structure and skewness, in order to be efficient. We summarize the algorithmic contributions here and describe each in detail in later sections.

1. **Careful Algorithm Choice.** We carefully choose a sequential eigensolver algorithm that is efficient for MAPREDUCE and gives accurate results.
2. **Selective Parallelization.** We group operations into expensive and inexpensive ones based on input sizes. Expensive operations are done in parallel for scalability, while inexpensive operations are performed on a single machine to avoid extra overhead of parallel execution.
3. **Blocking.** We reduce the running time by decreasing the input data size and the amount of network traffic among machines.
4. **Exploiting Skewness.** We decrease the running time by exploiting the skewness of data.

6.4.2 Careful Algorithm Choice

In Section 6.3, we considered three algorithms that are not tractable for analyzing billion-scale graphs with MAPREDUCE. Fortunately, there is an algorithm suitable for such a purpose. Lanczos-SO (Selective Orthogonalization) improves on the Lanczos-NO by selectively reorthogonalizing vectors instead of performing full reorthogonalizations.

The main idea of Lanczos-SO (Algorithm 6.3) is as follows: we start with a random initial basis vector b which comprises a rank-1 subspace. For each iteration, a new basis vector is computed by multiplying the input matrix with the previous basis vector. The new basis vector is then orthogonalized against the last two basis vectors and is added to the previous rank-$(m-1)$ subspace, forming a rank-m subspace. Let m be the number of the current iteration, Q_m be the $n \times m$ matrix whose ith column is the ith basis vector, and A be the matrix whose eigenvalues we seek to compute. We also define $T_m = Q_m^T A Q_m$ to be a $m \times m$ matrix. Then, the eigenvalues of T_m are good approximations of the eigenvalues of A . Furthermore, multiplying Q_m by the eigenvectors of T_m gives good approximation of the eigenvectors of A. We refer to [Trefethen and Bau III, 1997] for further details.

If we used the exact arithmetic, the newly computed basis vector would be orthogonal to all previous basis vectors. However, rounding errors from floating-point calculations compound and result in the loss of orthogonality. This is the cause of the spurious eigenvalues in Lanczos-NO. Orthogonality can be recovered once the new basis vector is fully re-orthogonalized to all previous vectors. However, this operation is quite expensive as it requires $O(m^2)$ re-orthogonalizations, where m is the number of iterations. A

Algorithm 6.3: Lanczos-SO (Selective Orthogonalization)

```
Input: matrix $A^{n \times n}$,
    random $n$-vector $b$,
    maximum number of steps $m$,
    error threshold $\epsilon$, and
    number of eigenvalues $k$.
Output: top $k$ eigenvalues $\lambda_{1..k}$, and
    eigenvectors $U^{n \times k}$.
1: $\beta_0 \leftarrow 0$, $v_0 \leftarrow 0$, $v_1 \leftarrow b/||b||$;
2: for $i = 1..m$ do
3:    $v \leftarrow Av_i$; // Find a new basis vector
4:    $\alpha_i \leftarrow v_i^T v$;
5:    $v \leftarrow v - \beta_{i-1}v_{i-1} - \alpha_i v_i$; // Orthogonalize against two previous basis vectors
6:    $\beta_i \leftarrow ||v||$;
7:    $T_i \leftarrow$ (build tri-diagonal matrix from $\alpha$ and $\beta$);
8:    $QDQ^T \leftarrow EIG(T_i)$; // Eigen decomposition of $T_i$
9:    for $j = 1..i$ do
10:      if $\beta_i |Q[i,j]| \leq \sqrt{\epsilon}||T_i||$ then
11:         $r \leftarrow V_i Q[:,j]$;
12:         $v \leftarrow v - (r^T v)r$; // Selectively orthogonalize
13:      end if
14:   end for
15:   if ($v$ was selectively orthogonalized) then
16:      $\beta_i \leftarrow ||v||$; // Recompute normalization constant $\beta_i$
17:   end if
18:   if $\beta_i = 0$ then
19:      break for loop;
20:   end if
21:   $v_{i+1} \leftarrow v/\beta_i$;
22: end for
23: $T \leftarrow$ (build tri-diagonal matrix from $\alpha$ and $\beta$);
24: $QDQ^T \leftarrow EIG(T)$; // Eigen decomposition of $T$
25: $\lambda_{1..k} \leftarrow$ top k diagonal elements of D; // Compute eigenvalues
26: $U \leftarrow V_m Q_k$; // Compute eigenvectors. $Q_k$ is the set of columns of $Q$ corresponding to $\lambda_{1..k}$
```

Operation	**Description**	**Input**	**P?**
$y \leftarrow y + ax$	vector update	Large	Yes
$\gamma \leftarrow x^T x$	vector dot product	Large	Yes
$y \leftarrow \alpha y$	vector scale	Large	Yes
$\|\|y\|\|$	vector L2 norm	Large	Yes
$y \leftarrow M^{n \times n} x$	large matrix-large, dense vector multiplication	Large	Yes
$y \leftarrow M_s^{n \times m} x_s$	large matrix-small vector multiplication ($n \gg m$)	Large	Yes
$A_s \leftarrow M_s^{n \times m} N_s^{m \times k}$	large matrix-small matrix multiplication ($n \gg m > k$)	Large	Yes
$\|\|T\|\|$	matrix L2 norm which is the largest singular value of the matrix	Tiny	No
$EIG(T)$	symmetric eigen decomposition to output QDQ^T	Tiny	No

Table 6.3: Parallelization Choices. The last column (**P?**) indicates whether the operation is parallelized in HEIGEN. Some operations are better to be run in parallel since the input size is very large, while others are better in a single machine since the input size is small and the overhead of parallel execution overshadows its decreased running time.

faster approach uses a quick test (line 10 of Algorithm 6.3) to selectively choose vectors that need to be re-orthogonalized to the new basis [Demmel, 1997]. This selective-reorthogonalization idea is shown in Algorithm 6.3.

The Lanczos-SO has all the properties that we need: it finds the top k largest eigenvalues and eigenvectors, it produces no spurious eigenvalues, and its most expensive operation, a matrix-vector multiplication, is tractable in MAPREDUCE. Therefore, we pick Lanczos-SO as our choice of the sequential algorithm for parallelization.

6.4.3 Selective Parallelization

Among many sub-operations in Algorithm 6.3, which operations should we parallelize? A naive approach is to parallelize all the operations; however, some operations run more quickly on a single machine rather than on multiple machines in parallel. The reason is that the overhead incurred by using MAPREDUCE exceeds gains made by parallelizing the task; simple tasks where the input data is very small are carried out faster on a single machine. Thus, we divide the sub-operations into two groups: those to be parallelized and those to be run in a single machine. Table 6.3 summarizes our choice for each sub-operation. Note that the last two operations in the table can be done with a single-machine standard eigensolver since the input matrices are tiny; they have m rows and columns, where m is the number of iterations.

6.4.4 Blocking

Minimizing the volume of information exchanged between nodes is important to designing efficient distributed algorithms. In HEIGEN, we decrease the amount of network traffic by using the block-based operation which was also used in Section 4.3.2. Normally, one would put each edge "(source, destination)" in one line; HADOOP treats each line as a data element for its mapper functions. Instead, we propose to divide the adjacency matrix into blocks (and, of course, the corresponding vectors also into blocks), and put the edges of each block on a single line, and compress the source- and destination-ids.

Algorithm 6.4: Cache-Based Matrix-Vector Multiplication (CBMV) for HEIGEN

```
Input: matrix $M = \{(id_{src}, (id_{dst}, mval))\}$, and
    vector $x = \{(id, vval)\}$.
Output: result vector $y$.
1: Stage1-Map(Key $k$, Value $v$, Vector $x$) : // Multiply matrix and vector elements
2: $id_{src} \leftarrow k$;
3: $(id_{dst}, mval) \leftarrow v$;
4: Output($id_{src}, (mval \times x[id_{dst}])$); // Multiply and output partial results
5:
6: Stage1-Reduce(Key $k$, Value $v[1..r]$) : // Sum up partial results
7: $sum \leftarrow 0$;
8: for $z \in v[1..r]$ do
9:     $sum \leftarrow sum + z$;
10: end for
11: Output($k, sum$);
```

This makes the mapper functions a bit more complicated to process blocks, but it saves significant transfer time of data over the network. We use these edge-blocks and the vector-blocks for many parallel operations in Table 6.3, including matrix-vector multiplication, vector update, vector dot product, vector scale, and vector L2 norm. Performing operations on blocks is faster than doing so on individual elements since both the input size and the key space decrease. This reduces the network traffic and sorting time in the MAPREDUCE Shuffle stage. As we will see in Section 6.6, the blocking decreases the running time by more than 4×.

6.4.5 Exploiting Skewness: Matrix-Vector Multiplication

HEIGEN uses an adaptive method for sub-operations based on the size of the data. In this section, we describe how HEIGEN implements different matrix-vector multiplication algorithms by exploiting the skewness pattern of the data. There are two matrix-vector multiplication operations in Algorithm 6.3: the one with a large vector (line 3) and the other with a small vector (line 11).

The first matrix-vector operation multiplies a matrix with a large and dense vector, and thus it requires a two-stage standard MAPREDUCE algorithm (see Algorithms 4.1 and 4.2). In the first stage, matrix elements and vector elements are joined and multiplied to produce partial results which are added together to get the result vector in the second stage.

The other matrix-vector operation, however, multiplies with a small vector. HEIGEN uses the fact that the small vector can fit in a machine's main memory, and distributes the small vector to all the mappers using the distributed cache functionality of HADOOP. The advantage of the small vector being available in mappers is that joining edge elements and vector elements can be done inside the mapper, and thus the first stage of the standard two-stage matrix-vector multiplication can be omitted. In this one-stage algorithm the mapper joins matrix elements and vector elements to make partial results, and the reducer adds up the partial results. The pseudo code of this algorithm, which we call CBMV (distributed Cache-Based Matrix-Vector multiplication), is shown in Algorithm 6.4. We want to emphasize that this operation cannot be performed when the vector is large, as is the case in the first matrix-vector multiplication (line 3 of

Algorithm 6.3). The CBMV is faster than the standard method by 57× as described in Section 6.6.

6.4.6 Exploiting Skewness: Matrix-Matrix Multiplication

Skewness can also be exploited to efficiently perform matrix-matrix multiplication (line 26 of Algorithm 6.3). In general, matrix-matrix multiplication is very expensive. A standard, yet naive, way of multiplying two matrices A and B in MAPREDUCE is to multiply $A[:, i]$ and $B[i, :]$ for each column i of A and sum the resulting matrices. This algorithm, which we call direct Matrix-Matrix multiplication (MM), is very inefficient since it generates huge matrices which are summed up many times. Fortunately, when one of the matrices is very small, we may exploit the skewness to come up with an efficient MAPREDUCE algorithm. This is exactly the case in HEIGEN; the first matrix is very large, and the second is very small. The main idea is to distribute the second matrix using the distributed cache functionality in HADOOP, and multiply each element of the first matrix with the corresponding rows of the second matrix. We call the resulting algorithm distributed Cache-Based Matrix-Matrix multiplication, or CBMM. There are other alternatives to matrix-matrix multiplication: one can decompose the second matrix into column vectors and iteratively multiply the first matrix with each of these vectors. We call the algorithms, introduced in Section 6.4.5, Iterative Matrix-Vector multiplications (IMV) and Cache-Based iterative Matrix-Vector multiplications (CBMV). The difference between CBMV and IMV is that CBMV uses distributed cache-based operations while IMV does not. As we will see in Section 6.6, the best method, CBMM, is faster than naive methods by more than 76×.

6.4.7 Analysis

We analyze the time and the space complexities of HEIGEN. In the lemmas below, m is the number of iterations, $|V|$ is the dimension of the matrix, $|E|$ is the number of nonzeros in the matrix, and s is the number of machines.

Lemma 6.1 (Time Complexity)**.** HEIGEN *takes* $O(m\frac{|V|+|E|}{s}\log\frac{|V|+|E|}{s})$ *time.*

Proof. The running time of one iteration of HEIGEN is dominated by the matrix-large vector multiplication whose running time is $O(m\frac{|V|+|E|}{s}\log\frac{|V|+|E|}{s})$. □

Lemma 6.2 (Space Complexity)**.** HEIGEN *requires* $O(|V|+|E|)$ *space.*

Proof. The maximum storage is required at the intermediate output of the two-stage matrix-vector multiplication where $O(|V|+|E|)$ space is needed. □

6.5 Additional Use of HEigen

The HEIGEN algorithm can be readily extended to other eigenvalue-based algorithms. In this secton, we describe large-scale algorithms for Singular Value Decomposition (SVD) and spectral clustering based on HEIGEN.

6.5.1 HEIGEN Gives SVD

Given any matrix A, the Singular Value Decomposition (SVD) gives the factorization

$$A = U\Sigma V^T,$$

where U and V are unitary matrices (i.e. square matrices that satisfy $U^T U = UU^T = I$ with I being the identity matrix), and Σ is a diagonal (if A is square) or a rectangular diagonal matrix (if A is rectangular), whose diagonal entries are real and non-negative, and are called *singular values* of A.

The SVD is a very powerful tool in analyzing graphs as well as matrices [Kamel, 1984, Berry., 1992]; some of its applications include optimal matrix approximation in the least squares sense [Eckart and Young, 1936], Principal Component Analysis [Pearson, 1901], clustering [Zha et al., 2002] (more specifically a relaxed version of the well known k-means clustering problem) and Information Retrieval/Latent Semantic Indexing [Deerwester et al., 1990].

HEIGEN can be extended to SVD of both symmetric and asymmetric matrices.

Symmetric Matrix. For a symmetric matrix A, the singular values of A are the absolute eigenvalues of A, and the singular vectors and the eigenvectors of A are the same up to signs. Thus, given an eigen decomposition $A = U\Lambda U^T$ computed by HEIGEN, we get the SVD

$$\begin{aligned} A &= U\Lambda U^T \\ &= U\Sigma S U^T \\ &= U\Sigma (US)^T, \end{aligned}$$

where $\Lambda = \Sigma S$, Σ is the diagonal matrix whose element $\Sigma(i,i)$ contains the absolute value of $\Lambda(i,i)$, and S is the diagonal matrix whose (i,i)-th element $S(i,i)$ is 1 if $\Lambda(i,i) \geq 0$, and -1 otherwise.

Asymmetric Matrix. For an asymmetric matrix $A^{n\times p}$, the standard method to compute the SVD $A = U\Sigma V^T$ is to build a symmetric $(n+p)\times(n+p)$ matrix

$$\hat{A} = \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix},$$

and apply HEIGEN on $\hat{A}$ [Berry., 1992]. The SVD (up to signs) of $\hat{A}$ is given by

$$\hat{A} = \begin{bmatrix} \frac{1}{\sqrt{2}}U & -\frac{1}{\sqrt{2}}U \\ \frac{1}{\sqrt{2}}V & \frac{1}{\sqrt{2}}V \end{bmatrix} \begin{bmatrix} \Sigma & 0 \\ 0 & -\Sigma \end{bmatrix} \begin{bmatrix} \frac{1}{\sqrt{2}}U^T & \frac{1}{\sqrt{2}}V^T \\ -\frac{1}{\sqrt{2}}U^T & \frac{1}{\sqrt{2}}V^T \end{bmatrix}.$$

Algorithm 6.5 shows the algorithm for standard SVD on asymmetric matrix using HEIGEN.

There are two shortcomings in Algorithm 6.5 which can be improved. First, we need to construct $\hat{A}$ which is $2\times$ larger than the original matrix A. Second, to get k singular values of A, we need to get $2k$ eigenvalues of $\hat{A}$ since there are 2 copies of the same eigenvalues in the eigen decomposition of $\hat{A}$.

Algorithm 6.5: Standard SVD on asymmetric matrix using HEIGEN

```
Input: matrix $A^{n \times p}$, and
    number of singular values $k$.
Output: top $k$ singular values $\sigma[1..k]$,
    left singular vectors $U^{n \times k}$, and
    right singular vectors $V^{p \times k}$ of $A$.
1: $\hat{A} \leftarrow \begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix}$;
2: Apply HEIGEN on $\hat{A}$;
```

Fast SVD for Asymmetric Matrix. We describe a faster SVD method for asymmetric matrices using HEIGEN, with the two main ideas. First, we use the fact that if $A = U\Sigma V^T$ is a SVD of A, then $AA^T = U\Sigma^2 U^T$ is a symmetric, positive definite matrix whose eigenvectors are the same as the left singular vectors of A, and eigenvalues are the square of the singular values of A. Thus, HEIGEN on AA^T gives us U and Σ. Having computed U and Σ, we can solve for V^T by $V^T = \Sigma^{-1}U^T A$. Naively applying HEIGEN on AA^T is not desired, however, since AA^T can be much larger and denser than A. Our second idea solves the problem by never materializing the matrix AA^T in applying HEIGEN on AA^T. Note that the input matrix in HEIGEN is used only for the matrix-vector multiplication on line 3 of Algorithm 6.3. We can efficiently compute the matrix-vector multiplication $(AA^T)v_i$ by $A(A^T v_i)$, which means to first compute $A^T v_i$, and multiply the resulting vector by A, thereby replacing a dense matrix-vector multiplication by two sparse matrix-vector multiplications.

6.5.2 HEIGEN Solves Large Scale Systems of Linear Equations

Solving large scale systems of linear equation is a very important task that pertains to almost every scientific discipline. Consider the following system of linear equations:

$$y = Ax,$$

where A is of size $m \times p$ and x is the vector of unknowns. In general, this system might have one, infinite or no solution, depending on the dimensions of A (if $m > p$ the system is called *overdetermined*, else if $m < p$ it is called *underdetermined*), and on whether y exists in the column space of A or not. In all the above cases, HEIGEN helps us calculate the only solution (when there exists one), find the best (in terms of the ℓ_2 norm) whenever there are infinite ones, or get the best ℓ_2 approximation of x, when there is no exact solution. For all the aforementioned cases, we call $\hat{x}$ the outcome. It can be shown that the solution $\hat{x}$ is given by $\hat{x} = A^\dagger y$, where $A^\dagger$ is the Moore-Penrose pseudoinverse of A [Penrose, 1955] which is defined by $A^\dagger = (A^T A)^{-1} A^T$ for a real matrix A with full column rank. A computationally efficient way to compute the pseudoinverse is to use the SVD. Specifically, given an SVD $A = U\Sigma V^T$ by HEIGEN, the pseudoinverse $A^\dagger$ is given by

$$A^\dagger = V\Sigma^{-1}U^T.$$

Furthermore, $\hat{x} = A^\dagger y = V\Sigma^{-1}U^T y$ can be computed efficiently in HADOOP by three matrix-vector multiplications. Specifically,

$$\hat{x} = Vw,$$

where $w = \Sigma^{-1}z$ and $z = U^T y$.

6.5.3 HEIGEN Gives HITS

HITS [Kleinberg, 1999] is a well-known algorithm to compute the 'hubs' and 'authorities' scores in Web pages. Given an adjacency matrix A of Web pages, the hub and the authority scores are given by the principal eigenvector of AA^T and $A^T A$, respectively. HEIGEN can give them since the left and the right singular vectors of A are the principal eigenvectors of AA^T and $A^T A$, respectively, as described in Section 6.5.1.

6.5.4 HEIGEN Gives Spectral Clustering

Spectral clustering is a popular clustering algorithm on graphs [Luxburg, 2007]. We consider the two spectral clustering algorithms by Shi et al. [Shi and Malik, 1997] and Ng et al. [Ng et al., 2002], and show how they can be easily computed with HEIGEN. Recall that the main idea of the spectral clustering is to first compute the k smallest eigenvectors of $L_{rw} = D^{-1}L$ [Shi and Malik, 1997] or $L_{sym} = D^{-\frac{1}{2}}LD^{-\frac{1}{2}}$ [Ng et al., 2002], respectively, and then run a k-means algorithm. Here, $L = D - A$ is the graph Laplacian matrix where A is a symmetric adjacency matrix of a graph and D is the diagonal matrix computed from A with $D(i, i) = \sum_j A(i, j)$.

Issues. Applying HEIGEN on the spectral clustering is not straightforward for the following two reasons. First, the spectral clustering algorithms require *k smallest* eigenvectors, while HEIGEN computes *k largest* eigenvectors of a matrix. Second, the L_{rw} is *asymmetric*, while HEIGEN works only on a *symmetric* matrix. However, HEIGEN can be used for these algorithms as we show below. We mildly assume that the input graph for the spectral clustering is connected.

Our solution on L_{sym}. Notice that $L_{sym} = D^{-\frac{1}{2}}LD^{-\frac{1}{2}} = I - D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$, where I is an identity matrix, and $L_{rw} = D^{-1}L = I - D^{-1}A$. It can be shown that $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ and $D^{-1}A$ share the same eigenvalues, and the eigenvalues range from -1 to 1. Also note that if λ is an eigenvalue of $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$, then $1 - \lambda$ is an eigenvalue of L_{sym} with the same eigenvector. Thus, the k smallest eigenvalues and eigenvectors of L_{sym} are mapped to the k largest eigenvalues and eigenvectors of $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$, which can be computed by HEIGEN. Algorithm 6.6 shows the spectral clustering with L_{sym} using HEIGEN.

Our solution on L_{rw}. It can be shown that u is an eigenvector of L_{rw} if and only if $D^{\frac{1}{2}}u$ is an eigenvector of L_{sym} with the same eigenvalue [Luxburg, 2007]. Thus, multiplying $D^{-\frac{1}{2}}$ to the k largest eigenvectors of $D^{-\frac{1}{2}}AD^{-\frac{1}{2}}$ leads to top k smallest eigenvectors of L_{rw}, as shown in Algorithm 6.7.

6.6 Performance

In this section, we present experimental results to answer the following questions:

Q1 How well does HEIGEN scale up?

Algorithm 6.6: Spectral Clustering with L_{sym} using HEIGEN

```
Input: matrix $A^{n \times m}$, and
    number of clusters $l$.
Output: $l$ clusters $C_{1..l}$.
1: Construct $D$;
2: $A' \leftarrow D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$;
3: $[U, \lambda_{1..k}] \leftarrow$ HEIGEN($A'$);
4: $C_{1..l} \leftarrow k$-mean on $U$;
```

Algorithm 6.7: Spectral Clustering with L_{rw} using HEIGEN

```
Input: matrix $A^{n \times m}$, and
    number of clusters $l$.
Output: $l$ clusters $C_{1..l}$.
1: Construct $D$;
2: $A' \leftarrow D^{-\frac{1}{2}} A D^{-\frac{1}{2}}$;
3: $[U, \lambda_{1..k}] \leftarrow$ HEIGEN($A'$);
4: $U' \leftarrow D^{-\frac{1}{2}} U$;
5: $C_{1..l} \leftarrow k$-means on $U'$;
```

Q2 Which of our proposed methods give the best performance?

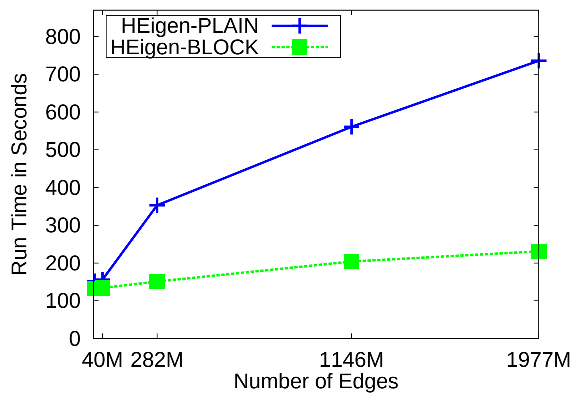
We perform experiments in the Yahoo! M45 HADOOP cluster. The scalability experiments are performed using synthetic Kronecker graphs [Leskovec et al., 2005] since realistic graphs of any size can be easily generated.

6.6.1 Scalability

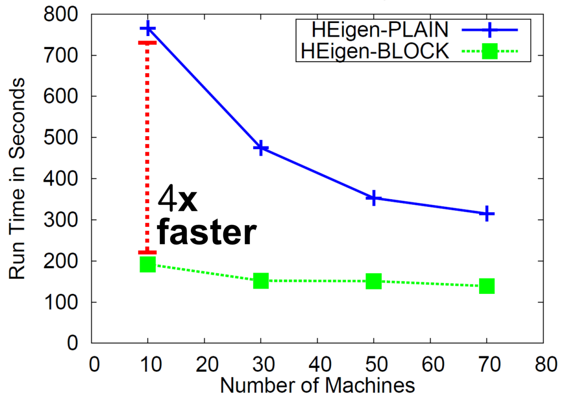
Figure 6.6 (a,b) shows the scalability of HEIGEN-BLOCK, an implementation of HEIGEN that uses blocking, and HEIGEN-PLAIN, an implementation which does not, on Kronecker graphs. Notice that the running time is near-linear in the number of edges and machines. We also note that HEIGEN-BLOCK performs up to 4× faster when compared to HEIGEN-PLAIN.

6.6.2 Optimizations

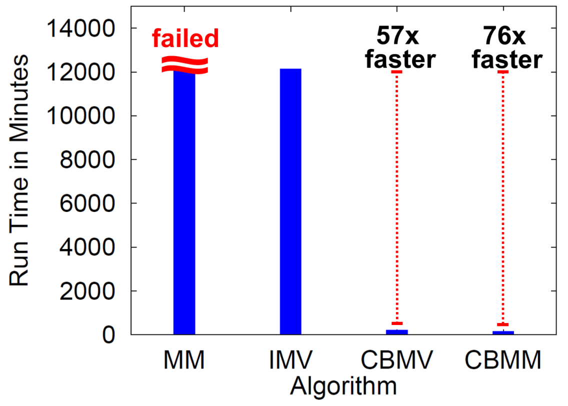
Figure 6.6 (c) shows the comparison of running time of the skewed matrix-matrix multiplication and the matrix-vector multiplication algorithms. We used 100 machines for YahooWeb data. For matrix-matrix multiplications, the best method is our proposed CBMM which is 76× faster than repeated naive matrix-vector multiplications (IMV). The slowest matrix-matrix multiplication algorithm did not even finish, and failed due to heavy amounts of intermediate data. For matrix-vector multiplications, our proposed CBMV is faster than the naive method (IMV) by 48×.



(a) Time vs. # of edges



(b) Time vs. # of machines



(c) Time vs. algorithms

Figure 6.6: **(a):** Running time vs. number of edges in 1 iteration of HEIGEN with 50 machines. Notice the near-linear running time proportional to the edges size. **(b):** Running time vs. number of machines in 1 iteration of HEIGEN. The running time decreases as number of machines increase. **(c):** Comparison of running time between different skewed matrix-matrix and matrix-vector multiplications. For matrix-matrix multiplication, our proposed CBMM outperforms naive methods by at least 76×. The slowest matrix-matrix multiplication algorithm did not even finish and the job failed due to excessive intermediate data. For matrix-vector multiplication, our proposed CBMV is faster than the naive method by 57×.

6.7 Related Work

We review the related works on large-scale eigensolvers. There are several eigensolvers for a single machine, including [Jam, Div], but their scalability is limited by the memory of the machine. On the side of parallel eigensolvers, there are several works including the work by Zhao et al. [Zhao et al., 2007b], HPEC [Guarracino et al., 2006], PLANO [Wu and Simon, 1999], PREPACK [Lehoucq et al., 1998], SCALABLE [Blackford et al., 1997], and PLAYBACK [Alpatov et al., 1997]. However, all of them are based on Message Passing Interface (MPI) which has difficulty in dealing with billion-scale graphs. The maximum order of matrices analyzed with these tools is less than 1 million [Wu and Simon, 1999, Song et al., 2008], which is far from the Web-scale data. Recently (March 2010), the Mahout project [Mah] provides SVD on top of HADOOP. Mahout suffers from two major issues: (a) it assumes that the vector (b, with n=O(billion) entries) fits in the memory of a single machine, and (b) it implements the full re-orthogonalization which is inefficient.

6.8 Conclusion

In this chapter we discover spectral patterns in real-world, billion-scale graphs. This is possible by using HEIGEN, our proposed eigensolver for the spectral analysis of very large-scale graphs. The main contributions are the following:

- **Effectiveness.** We analyze the spectral properties of real world graphs, including Twitter and one of the largest public Web graphs. We report patterns that can be used for anomaly detection and finding tightly-knit communities.
- **Careful Design.** We carefully design HEIGEN to selectively parallelize operations based on how they are most effectively performed.
- **Scalability.** We implement and evaluate a billion-scale eigensolver. Experiments show that HEIGEN scales linearly with the number of edges.

Chapter 7

Tensor Analysis

Many data are modeled as tensors, or multi dimensional arrays. Examples include the predicates (subject, verb, object) in knowledge bases, hyperlinks and anchor texts in Web graphs, sensor streams (time, location, and type), social networks over time, and DBLP conference-author-keyword relations. Tensor decomposition is an important data mining tool with various applications including clustering, trend detection, and anomaly detection. However, current tensor decomposition algorithms are not scalable for large tensors with billions of sizes and hundreds millions of nonzeros: the largest tensor in literatures remains thousands of sizes and hundreds thousands of nonzeros.

Consider a knowledge base tensor consisting of about 26 million noun-phrases. The intermediate data explosion problem, associated with naive implementations of tensor decomposition algorithms, would require the materialization and the storage of a matrix whose largest dimension would be $\approx 7 \cdot 10^{14}$; this amounts to $\sim$ 10 Petabytes, or equivalently a few data centers worth of storage, thereby rendering the tensor analysis of this knowledge base, in the naive way, practically impossible. In this chapter, we propose GIGATENSOR, a scalable distributed algorithm for large scale tensor decomposition. GIGATENSOR exploits the sparseness of the real world tensors, and avoids the intermediate data explosion problem by carefully redesigning the tensor decomposition algorithm.

Extensive experiments show that our proposed GIGATENSOR solves 100× bigger problems than existing methods. Furthermore, we employ GIGATENSOR in order to analyze a very large real world, knowledge base tensor and present our findings which include discovery of potential synonyms among millions of noun-phrases (e.g. the noun 'pollutant' and the noun-phrase 'greenhouse gases').

7.1 Introduction

Tensors, or multi-dimensional arrays appear in numerous applications: predicates (subject, verb, object) in knowledge bases [Carlson et al., 2010], hyperlinks and anchor texts in Web graphs [Kolda and Bader, 2006], sensor streams (time, location, and type) [Sun et al., 2006a], and DBLP conference-author-keyword relations [Kolda and Sun, 2008], to name a few. Analysis of multi-dimensional arrays by tensor decompositions, as shown in Figure 7.2, is a basis for many interesting applications including clustering, trend detection, anomaly detection [Kolda and Sun, 2008], correlation analysis [Sun et al., 2006a], network forensics [Maruhashi et al., 2011], and latent concept discovery [Kolda and Bader, 2006].

Work	Tensor Size	Nonzeros
Kolda et al. [Kolda and Bader, 2006]	$787 \times 787 \times 533$	3583
Acar et al. [Acar et al., 2007]	3.4 K $\times$ 100×18	(dense)
Maruhashi et al. [Maruhashi et al., 2011]	2 K $\times$ 2 K $\times$ 6 K $\times$ 4 K	281 K
GIGATENSOR (**This work**)	26 M $\times$ 26 M $\times$ 48 M	144 M

Table 7.1: Indicative sizes of tensors analyzed in the data mining literature. M: Million, K: Thousand. Our proposed GIGATENSOR analyzes tensors with $\approx 1000\times$ larger sizes and $\approx 500\times$ larger nonzero elements.

There exist two, widely used, toolboxes that handle tensors and tensor decompositions: The Tensor Toolbox for Matlab [Bader and Kolda, 2007a], and the N-way Toolbox for Matlab [Andersson and Bro, 2000]. Both toolboxes are considered the state of the art; especially, the Tensor Toolbox is probably the fastest existing implementation of tensor decompositions for sparse tensors (having attracted best paper awards, e.g. see [Kolda and Sun, 2008]). However, the toolboxes have critical restrictions: 1) they operate strictly on data that can fit in the main memory, and 2) their scalability is limited by the scalability of Matlab. In [Bader and Kolda, 2007b, Kolda and Sun, 2008], efficient ways of computing tensor decompositions, when the tensor is very sparse, are introduced and are implemented in the Tensor Toolbox. However, these methods still operate in main memory and therefore cannot scale to Gigabytes or Terabytes of tensor data. The need for large scale tensor computations is ever increasing, and there is a huge gap that needs to be filled. In Table 7.1, we present an indicative sample of tensor sizes that have been analyzed so far; we can see that these sizes are nowhere near as adequate as needed, in order to satisfy current real data needs, which call for tensors with billions of sizes and hundreds of millions of nonzero elements.

In this chapter, we propose GIGATENSOR, a scalable distributed algorithm for large scale tensor decomposition. GIGATENSOR can handle Tera-scale tensors using the MAPREDUCE [Dean and Ghemawat, 2004] framework, and more specifically its open source implementation, HADOOP [Had]. To the best of our knowledge, this work is the first approach of deploying tensor decompositions in the MAPREDUCE framework. The main contributions of this chapter are the following:

- **Algorithm.** We propose GIGATENSOR, a large scale tensor decomposition algorithm on MAPREDUCE. GIGATENSOR is carefully designed to minimize the intermediate data size and the number of floating point operations.
- **Scalability.** GIGATENSOR decomposes $100\times$ larger tensors compared to existing methods, as shown in Figure 7.1. Furthermore, GIGATENSOR enjoys linear scalability on the number of machines.
- **Discovery.** We discover patterns in a very large knowledge-base tensor dataset from the 'Read the Web' project [Carlson et al., 2010], which until now, was unable to be analyzed using tensor tools. Our findings include potential synonyms of noun-phrases, which were discovered after decomposing the knowledge base tensor; these findings are shown in Table 7.2 and a detailed description of the discovery procedure is covered on Section 7.4.

The rest of this chapter is organized as follows. Section 7.2 presents the preliminaries of the tensor decomposition. Sections 7.3 describes our proposed algorithm for large scale tensor analysis. Section 7.4 presents the experimental results. After reviewing related works in Section 7.5, we conclude in Section 7.6.

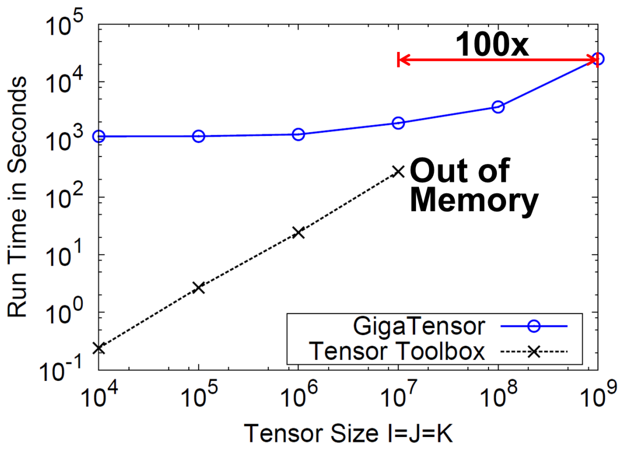


Figure 7.1: The scalability of GIGATENSOR compared to the Tensor Toolbox with regard to the tensor sizes. We fix the number of nonzero elements to 10^4 on the synthetic data. Notice that GIGATENSOR solves $100\times$ larger problem than the Tensor Toolbox which runs out of memory on tensors of sizes beyond 10^7.

(Given) Noun Phrase	**(Discovered) Potential Synonyms**
pollutants	dioxin, sulfur dioxide, greenhouse gases, particulates, nitrogen oxide, air pollutants, cholesterol
disabilities	infections, dizziness, injuries, diseases, drowsiness, stiffness, injuries
vodafone	verizon, comcast
Christian history	European history, American history, Islamic history, history
disbelief	dismay, disgust, astonishment
cyberpunk	online-gaming
soul	body

Table 7.2: Given noun-phrases and potential synonyms, discovered using tensor decomposition of the NELL-1 knowledge base dataset (more details in Section 7.4).

Symbol	Definition
$\mathcal{X}$	a tensor
$\mathbf{X_{(n)}}$	mode-n matricization of a tensor
m	number of nonzero elements in a tensor
a	a scalar (lowercase, italic letter)
$\mathbf{a}$	a column vector (lowercase, bold letter)
$\mathbf{A}$	a matrix (uppercase, bold letter)
R	number of components
$\circ$	outer product
$\odot$	Khatri-Rao product
$\otimes$	Kronecker product
$*$	Hadamard product
$\cdot$	standard product
$\mathbf{A}^T$	transpose of $\mathbf{A}$
$\mathbf{M}^\dagger$	pseudoinverse of $\mathbf{M}$
$\|\mathbf{M}\|_F$	Frobenius norm of $\mathbf{M}$
$bin(\mathbf{M})$	function that converts non-zero elements of $\mathbf{M}$ to 1

Table 7.3: Table of symbols.

7.2 Preliminaries; Tensor Decomposition

In this section, we describe the preliminaries on the tensor decomposition whose fast algorithm will be proposed in Section 7.3. Table 7.3 lists the symbols used in this chapter. For vector/matrix/tensor indexing, we use the Matlab-like notation, i.e. $\mathbf{A}(i, j)$ denotes the (i, j)-th element of matrix $\mathbf{A}$, whereas $\mathbf{A}(:, j)$ spans the j-th column of that matrix.

Matrices and the bilinear decomposition. Let $\mathbf{X}$ be an $I \times J$ matrix. There is a very intuitive way to define the rank of $\mathbf{X}$, namely, the minimum number of rank one matrices that are required to compose $\mathbf{X}$. A rank one matrix is simply an *outer product* of two vectors, say $\mathbf{a}\mathbf{b}^T$. If rank($\mathbf{X}$) = R, then we can write

$$\mathbf{X} = \mathbf{a}_1\mathbf{b}_1^T + \mathbf{a}_2\mathbf{b}_2^T + \cdots + \mathbf{a}_R\mathbf{b}_R^T,$$

which is called the *bilinear decomposition* of $\mathbf{X}$. The bilinear decomposition is compactly written as $\mathbf{X} = \mathbf{A}\mathbf{B}^T$, where the columns of $\mathbf{A}$ and $\mathbf{B}$ are $\mathbf{a}_r$ and $\mathbf{b}_r$, respectively, for $1 \leq r \leq R$. Usually, one may truncate this decomposition for $R \ll$ rank($\mathbf{X}$), in which case we have a low rank approximation of $\mathbf{X}$.

There is a very natural decomposition of $\mathbf{X}$, its Singular Value Decomposition (SVD). As mentioned earlier in Section 6.5.1, the SVD of $\mathbf{X}$ is

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T,$$

where $\mathbf{U}, \mathbf{V}$ are unitary $I \times I$ and $J \times J$ matrices, respectively, and $\mathbf{\Sigma}$ is a rectangular diagonal matrix, containing the (non-negative) singular values of $\mathbf{X}$. If we pick $\mathbf{A} = \mathbf{U}\mathbf{\Sigma}$ and $\mathbf{B} = \mathbf{V}$, and pick $R <$ rank($\mathbf{X}$), this is the optimal low rank approximation of $\mathbf{X}$ in the least squares sense [Eckart and Young, 1936]. The SVD is also a very powerful tool used in computing the so called Moore-Penrose pseudoinverse [Penrose,

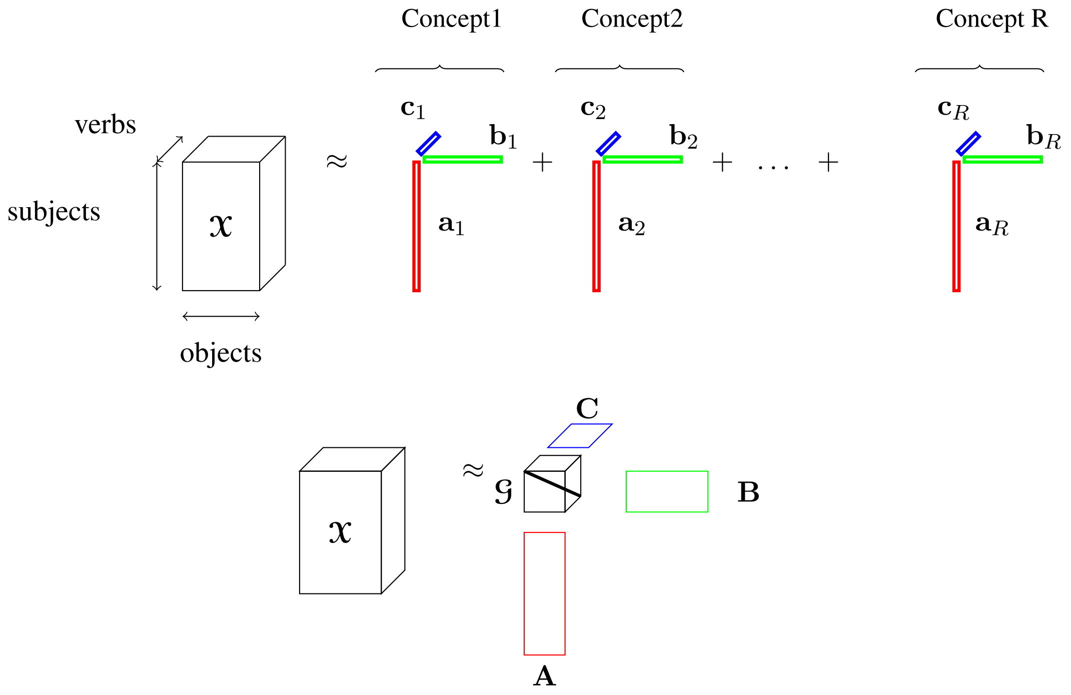


Figure 7.2: PARAFAC decomposition of three-way tensor as sum of R outer products (rank-one tensors), reminiscing of the rank-R SVD of a matrix (top), and as product of matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$, and a super-diagonal core tensor $\mathcal{G}$ (bottom).

1955], which lies in the heart of the PARAFAC tensor decomposition which we will describe soon. The Moore-Penrose pseudoinverse of $\mathbf{X}$ is simply:

$$\mathbf{A}^{\dagger} = \mathbf{V}\mathbf{\Sigma}^{-\mathbf{1}}\mathbf{U}^{T}.$$

We now provide a brief introduction to tensors and the PARAFAC decomposition. For a more detailed treatment of the subject, we refer the interested reader to [Kolda and Bader, 2009].

Introduction to PARAFAC. Consider a three way tensor $\mathcal{X}$ of dimensions $I \times J \times K$; for the purposes of this initial analysis, we restrict ourselves to the study of three way tensors. The generalization to higher ways is trivial, provided that a robust implementation for three way decompositions exists.
Definition 4 (Three way outer product)**.** *The three way outer product of vectors* $\mathbf{a}, \mathbf{b}, \mathbf{c}$ *is defined as*

$$[\mathbf{a} \circ \mathbf{b} \circ \mathbf{c}](i, j, k) = \mathbf{a}(i)\mathbf{b}(j)\mathbf{c}(k).$$

Definition 5 (PARAFAC decomposition)**.** *The PARAFAC [Harshman, 1970, Bro, 1997] (also known as CP or trilinear) tensor decomposition of* $\mathcal{X}$ *in* R *components is*

$$\mathcal{X} \approx \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r.$$

The PARAFAC decomposition, illustrated in Figure 7.2, is a generalization of the matrix bilinear decomposition in three and higher ways. More compactly, we can write the PARAFAC decomposition as a triplet of matrices $\mathbf{A}, \mathbf{B}$, and $\mathbf{C}$, i.e. the r-th column of which contains $\mathbf{a}_r$, $\mathbf{b}_r$ and $\mathbf{c}_r$, respectively.

Furthermore, one may normalize each column of the three factor matrices, and introduce a scalar term λ_r, one for each rank-one factor of the decomposition (comprising a $R \times 1$ vector $\boldsymbol{\lambda}$), which forces the factor vectors to be of unit norm. Hence, the PARAFAC model we are computing is:

$$\mathcal{X} \approx \sum_{r=1}^{R} \lambda_r \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r.$$

Definition 6 (Tensor Unfolding/Matricization). *We may unfold/matricize the tensor $\mathcal{X}$ in the following three ways: $\mathbf{X}_{(1)}$ of size $(I \times JK)$, $\mathbf{X}_{(2)}$ of size $(J \times IK)$ and $\mathbf{X}_{(3)}$ of size $(K \times IJ)$. The tensor $\mathcal{X}$ and the matricizations are mapped in the following way.*

$$\mathcal{X}(i,j,k) \rightarrow \mathbf{X}_{(1)}(i, j + (k-1)J). \tag{7.1}$$

$$\mathcal{X}(i,j,k) \rightarrow \mathbf{X}_{(2)}(j, i + (k-1)I). \tag{7.2}$$

$$\mathcal{X}(i,j,k) \rightarrow \mathbf{X}_{(3)}(k, i + (j-1)I). \tag{7.3}$$

We now set off to introduce some notions that play a key role in the computation of the PARAFAC decomposition.

Definition 7 (Kronecker product). *The Kronecker product of $\mathbf{A}$ and $\mathbf{B}$ is:*

$$\mathbf{A} \otimes \mathbf{B} := \begin{bmatrix} \mathbf{B}\mathbf{A}(1,1) & \cdots & \mathbf{B}\mathbf{A}(1,J_1) \\ \vdots & \ddots & \vdots \\ \mathbf{B}\mathbf{A}(I_1,1) & \cdots & \mathbf{B}\mathbf{A}(I_1,J_1) \end{bmatrix}$$

If $\mathbf{A}$ is of size $I_1 \times J_1$ and $\mathbf{B}$ of size $I_2 \times J_2$, then $\mathbf{A} \otimes \mathbf{B}$ is of size $I_1 I_2 \times J_1 J_2$.

Definition 8 (Khatri-Rao product). *The Khatri-Rao product (or column-wise Kronecker product) $(\mathbf{A} \odot \mathbf{B})$, where $\mathbf{A}, \mathbf{B}$ have the same number of columns, say R, is defined as:*

$$\mathbf{A} \odot \mathbf{B} = \left[\mathbf{A}(:,1) \otimes \mathbf{B}(:,1) \cdots \mathbf{A}(:,R) \otimes \mathbf{B}(:,R)\right]$$

If $\mathbf{A}$ is of size $I \times R$ and $\mathbf{B}$ is of size $J \times R$ then $(\mathbf{A} \odot \mathbf{B})$ is of size $IJ \times R$.

The Alternating Least Squares Algorithm for PARAFAC. The most popular algorithm for fitting the PARAFAC decomposition is the Alternating Least Squares (ALS). The ALS algorithm consists of three steps, each one being a conditional update of one of the three factor matrices, given the other two. The version of the algorithm we are using is the one outlined in Algorithm 7.1; for a detailed overview of the ALS algorithm, see [Kolda and Bader, 2009, Harshman, 1970, Bro, 1997].

The stopping criterion for Algorithm 7.1 is either one of the following: 1) the maximum number of iterations is reached, or 2) the cost of the model for two consecutive iterations stops changing significantly (i.e. the difference between the two costs is within a small number ϵ, usually in the order of 10^{-6}). The cost of the model is simply the least squares cost.

The most important issue pertaining to the scalability of Algorithm 7.1 is the "intermediate data explosion" problem. During the life of Algorithm 7.1, a naive implementation have to materialize matrices $(\mathbf{C} \odot \mathbf{B})$, $(\mathbf{C} \odot \mathbf{A})$, and $(\mathbf{B} \odot \mathbf{A})$, which are very large in sizes.

Problem 1 (intermediate data explosion). *The problem of having to materialize $(\mathbf{C} \odot \mathbf{B})$, $(\mathbf{C} \odot \mathbf{A})$, $(\mathbf{B} \odot \mathbf{A})$ is defined as the intermediate data explosion.*

Algorithm 7.1: Alternating Least Squares for the PARAFAC decomposition.

```
Input: tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$,
    rank $R$, and
    maximum iterations $T$.
Output: PARAFAC decomposition $\boldsymbol{\lambda} \in \mathbb{R}^{R \times 1}$,
    $\mathbf{A} \in \mathbb{R}^{I \times R}$,
    $\mathbf{B} \in \mathbb{R}^{J \times R}$, and
    $\mathbf{C} \in \mathbb{R}^{K \times R}$.
1: Initialize $\mathbf{A}, \mathbf{B}, \mathbf{C}$;
2: for $t = 1, ..., T$ do
3:   $\mathbf{A} \leftarrow \mathbf{X_{(1)}} (\mathbf{C} \odot \mathbf{B}) (\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^{\dagger}$;
4:   Normalize columns of $\mathbf{A}$ (storing norms in vector $\boldsymbol{\lambda}$);
5:   $\mathbf{B} \leftarrow \mathbf{X_{(2)}} (\mathbf{C} \odot \mathbf{A}) (\mathbf{C}^T\mathbf{C} * \mathbf{A}^T\mathbf{A})^{\dagger}$;
6:   Normalize columns of $\mathbf{B}$ (storing norms in vector $\boldsymbol{\lambda}$);
7:   $\mathbf{C} \leftarrow \mathbf{X_{(3)}} (\mathbf{B} \odot \mathbf{A}) (\mathbf{B}^T\mathbf{B} * \mathbf{A}^T\mathbf{A})^{\dagger}$;
8:   Normalize columns of $\mathbf{C}$ (storing norms in vector $\boldsymbol{\lambda}$);
9:   if convergence criterion is met then
10:    break for loop;
11:  end if
12: end for
13: return $\boldsymbol{\lambda}, \mathbf{A}, \mathbf{B}, \mathbf{C}$;
```

In order to give an idea of how devastating this intermediate data explosion problem is, consider the NELL-1 knowledge base dataset, described in Section 7.4, that we are using in this work; this dataset consists of about $26 \cdot 10^6$ noun-phrases (and for a moment, ignore the number of the "context" phrases, which account for the third mode). Then, one of the intermediate matrices will have an explosive dimension of $\approx 7 \cdot 10^{14}$, or equivalently a few data centers worth of storage, rendering any practical way of materializing and storing it, virtually impossible.

In [Bader and Kolda, 2007b], Bader et al. introduce a way to alleviate the above problem, when the tensor is represented in a sparse form, in Matlab. This approach is however, as we mentioned earlier, bound by the memory limitations of Matlab. In Section 7.3, we describe our proposed method which effectively tackles intermediate data explosion, especially for sparse tensors, and is able to scale to very large tensors, because it operates on a distributed system.

7.3 Proposed Method

In this section, we describe GIGATENSOR, our proposed MAPREDUCE algorithm for large scale tensor analysis.

7.3.1 Overview

GIGATENSOR provides an efficient distributed algorithm for the PARAFAC tensor decomposition on MAPREDUCE. The major challenge is to design an efficient algorithm for updating factors (line 3, 5, and 7 of Algorithm 7.1). Since the update rules are similar, we focus on updating the $\mathbf{A}$ matrix. As shown in the line 3 of Algorithm 7.1, the update rule for A is

$$\hat{\mathbf{A}} \leftarrow \mathbf{X}_{(\mathbf{1})}(\mathbf{C} \odot \mathbf{B})(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger, \tag{7.4}$$

where $\mathbf{X}_{(\mathbf{1})} \in \mathbb{R}^{I \times JK}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, $\mathbf{C} \in \mathbb{R}^{K \times R}$, $(\mathbf{C} \odot \mathbf{B}) \in \mathbb{R}^{JK \times R}$, and $(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger \in \mathbb{R}^{R \times R}$. $\mathbf{X}_{(\mathbf{1})}$ is very sparse, especially in real world tensors, while $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$ are dense.

There are several challenges in designing an efficient MAPREDUCE algorithm for Equation (7.4) in GIGATENSOR:

1. **Minimize flops.** How to minimize the number of floating point operations (flops) for computing Equation (7.4)?
2. **Minimize intermediate data.** How to minimize the intermediate data, i.e. the amount of network traffic in the shuffling stage of MAPREDUCE?
3. **Exploit data characteristics.** How to exploit the data characteristics including the sparsity of the real world tensor and the skewness in matrix multiplications to design an efficient MAPREDUCE algorithm?

We have the following main ideas to address the above challenges which we describe in detail in later subsections.

1. **Careful choice of order of computations** in order to minimize flops (Section 7.3.2).
2. **Avoiding intermediate data explosion** by exploiting the sparsity of real world tensors (Section 7.3.3 and 7.3.4).
3. **Parallel outer products** to minimize intermediate data (Section 7.3.4).
4. **Distributed cache multiplication** to minimize intermediate data by exploiting the skewness in matrix multiplications (Section 7.3.4).

7.3.2 Ordering of Computations

Equation (7.4) entails three matrix-matrix multiplications, assuming that we have already computed $(\mathbf{C} \odot \mathbf{B})$ and $(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger$. Since matrix multiplication is commutative, Equation (7.4) can be computed by either multiplying the first two matrices, and multiplying the result with the third matrix:

$$[\mathbf{X}_{(\mathbf{1})}(\mathbf{C} \odot \mathbf{B})](\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger, \tag{7.5}$$

or multiplying the last two matrices, and multiplying the first matrix with the result:

$$\mathbf{X}_{(\mathbf{1})}[(\mathbf{C} \odot \mathbf{B})(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger]. \tag{7.6}$$

The question is, which equation is better between (7.5) and (7.6)? From a standard result of numerical linear algebra (e.g. [Boyd and Vandenberghe, 2004]), the equation (7.5) requires $2mR + 2IR^2$ flops, where m is the number of nonzeros in the tensor $\mathcal{X}$, while the equation (7.6) requires $2mR + 2JKR^2$ flops. Given that the product of the two dimension sizes (JK) is larger than the other dimension size (I) in most practical cases, Equation (7.5) results in smaller flops. For example, referring to the NELL-1 dataset

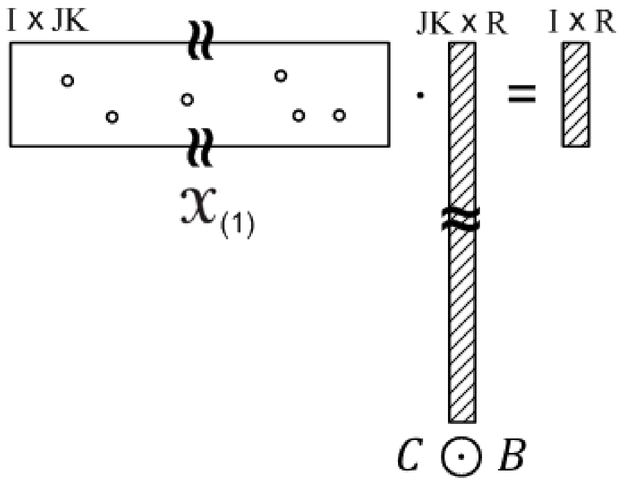


Figure 7.3: The "intermediate data explosion " problem in computing $\mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B})$. Although $\mathbf{X_{(1)}}$ is sparse, the matrix $\mathbf{C} \odot \mathbf{B}$ is very dense and long. Materializing $\mathbf{C} \odot \mathbf{B}$ requires too much storage: e.g., for $J = K \approx 26$ million as in the NELL-1 data of Table 7.7, the number of rows of $\mathbf{C} \odot \mathbf{B}$ explodes to *676 trillion*.

of Table 7.7, Equation (7.5) requires $\approx 8 \cdot 10^9$ flops while Equation (7.6) requires $\approx 2.5 \cdot 10^{17}$ flops. For the reason, we choose the Equation (7.5) ordering for updating factor matrices. That is, we perform the following three matrix-matrix multiplications for Equation (7.4):

$$\texttt{Step 1 :} \quad \mathbf{M}_1 \leftarrow \mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B}) \tag{7.7}$$

$$\texttt{Step 2 :} \quad \mathbf{M}_2 \leftarrow (\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger \tag{7.8}$$

$$\texttt{Step 3 :} \quad \mathbf{M}_3 \leftarrow \mathbf{M}_1\mathbf{M}_2 \tag{7.9}$$

7.3.3 Avoiding the Intermediate Data Explosion Problem

As introduced at the end of Section 7.2, one of the most important issue for scaling up the tensor decomposition is the intermediate data explosion problem. In this subsection we describe the problem in detail, and propose our solution.

Problem. A naive algorithm to compute $\mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B})$ is to first construct $\mathbf{C} \odot \mathbf{B}$, and multiply $\mathbf{X_{(1)}}$ with $\mathbf{C} \odot \mathbf{B}$, as illustrated in Figure 7.3. The problem ("intermediate data explosion ") of this algorithm is that although the matricized tensor $\mathbf{X_{(1)}}$ is sparse, the matrix $\mathbf{C} \odot \mathbf{B}$ is very large and dense; thus, $\mathbf{C} \odot \mathbf{B}$ cannot be stored even in multiple disks in a typical HADOOP cluster.

Our Solution. Our crucial observation is that $\mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B})$ can be computed without explicitly constructing $\mathbf{C} \odot \mathbf{B}$. [1] Our main idea is to decouple the two terms in the Khatri-Rao product, and perform algebraic operations involving $\mathbf{X_{(1)}}$ and $\mathbf{C}$, then $\mathbf{X_{(1)}}$ and $\mathbf{B}$, and then combine the result. Our main idea

[1]Bader et al. [Bader and Kolda, 2007b] has an alternative way to avoid the intermediate data explosion, but it is implemented in MATLAB and thus does not scale to a very large tensor.

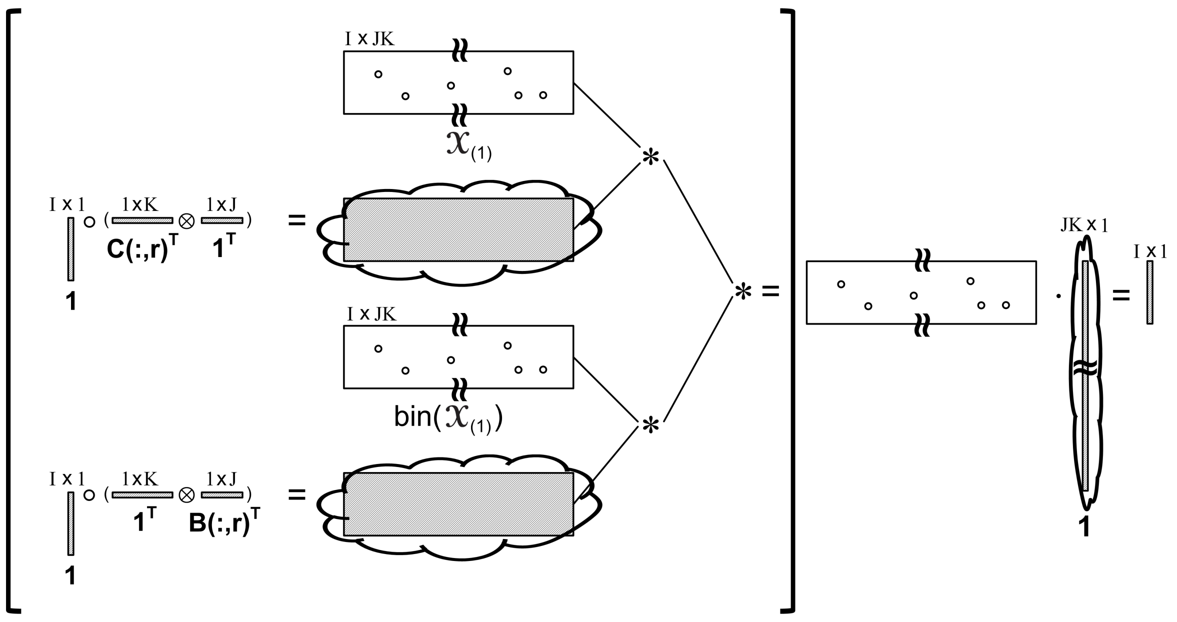


Figure 7.4: Our solution to avoid the intermediate data explosion, for $R = 1$. The main idea is to decouple the two terms in the Khatri-Rao product, and perform algebraic operations using $\mathbf{X_{(1)}}$ and $\mathbf{C}$, and then $\mathbf{X_{(1)}}$ with $\mathbf{B}$, and combine the result. The symbols $\circ, \otimes, *$, and $\cdot$ represents the outer, Kronecker, Hadamard, and the standard product, respectively. Shaded matrices are dense, and empty matrices with several circles are sparse. The clouds surrounding matrices represent that the matrices are *not* materialized. Note that the matrix $\mathbf{C} \odot \mathbf{B}$ is never constructed, and the largest dense matrix is either the $\mathbf{B}$ or the $\mathbf{C}$ matrix.

is described in Algorithm 7.2 as well as in Figure 7.4. In line 7 of Algorithm 7.2, the Hadamard product of $\mathbf{X_{(1)}}$ and a matrix derived from $\mathbf{C}$ is performed. In line 8, the Hadamard product of $\mathbf{X_{(1)}}$ and a matrix derived from $\mathbf{B}$ is performed, where the $bin()$ function converts any nonzero value into 1, preserving sparsity. In line 9, the Hadamard product of the two result matrices from lines 7 and 8 is performed, and the elements of each row of the resulting matrix are summed up to get the final result vector $\mathbf{M}_1(:, r)$ in line 10. The following theorem demonstrates the correctness of Algorithm 7.2.

Theorem 7.1. *Computing* $\mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B})$ *is equivalent to computing* $(\mathbf{N_1} * \mathbf{N_2}) \cdot \mathbf{1}_{JK}$*, where* $\mathbf{N}_1 = \mathbf{X_{(1)}} * (\mathbf{1}_I \circ (\mathbf{C}(:, r)^T \otimes \mathbf{1}_J^T))$*,* $\mathbf{N}_2 = bin(\mathbf{X_{(1)}}) * (\mathbf{1}_I \circ (\mathbf{1}_K^T \otimes \mathbf{B}(:, r)^T))$*, and* $\mathbf{1_{JK}}$ *is an all-1 vector of size* JK*.*

Proof. The (i, y)-th element of $\mathbf{N}_1$ is given by

$$\mathbf{N}_1(i, y) = \mathbf{X_{(1)}}(i, y)\mathbf{C}(\lceil \frac{y}{J} \rceil, r).$$

The (i, y)-th element of $\mathbf{N}_2$ is given by

$$\mathbf{N}_2(i, y) = \mathbf{B}(1 + (y - 1)\%J, r).$$

The (i, y)-th element of $\mathbf{N}_3 = \mathbf{N_1} * \mathbf{N_2}$ is

Algorithm 7.2: Multiplying $\mathbf{X_{(1)}}$ and $\mathbf{C} \odot \mathbf{B}$ in GIGATENSOR.

```
Input: tensor $\mathbf{X_{(1)}} \in \mathbb{R}^{I \times JK}$,
       $\mathbf{C} \in \mathbb{R}^{K \times R}$, and
       $\mathbf{B} \in \mathbb{R}^{J \times R}$.
Output: $\mathbf{M}_1 \leftarrow \mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B})$.
1: $\mathbf{M_1} \leftarrow 0$;
2: $\mathbf{1_I} \leftarrow$ all 1 vector of size $I$;
3: $\mathbf{1_J} \leftarrow$ all 1 vector of size $J$;
4: $\mathbf{1_K} \leftarrow$ all 1 vector of size $K$;
5: $\mathbf{1_{JK}} \leftarrow$ all 1 vector of size $JK$;
6: for $r = 1, ..., R$ do
7:     $\mathbf{N}_1 \leftarrow \mathbf{X_{(1)}} * (\mathbf{1}_I \circ (\mathbf{C}(:, r)^T \otimes \mathbf{1}_J^T))$;
8:     $\mathbf{N}_2 \leftarrow bin(\mathbf{X_{(1)}}) * (\mathbf{1}_I \circ (\mathbf{1}_K^T \otimes \mathbf{B}(:, r)^T))$;
9:     $\mathbf{N}_3 \leftarrow \mathbf{N}_1 * \mathbf{N}_2$;
10:    $\mathbf{M}_1(:, r) \leftarrow \mathbf{N}_3 \cdot \mathbf{1}_{JK}$;
11: end for
12: return $\mathbf{M}_1$;
```

$$\mathbf{N}_3(i, y) = \mathbf{X_{(1)}}(i, y)\mathbf{C}(\lceil \frac{y}{J} \rceil, r)\mathbf{B}(1 + (y-1)\%J, r).$$

Multiplying $\mathbf{N}_3$ with $\mathbf{1}_{JK}$, which essentially sums up each row of $\mathbf{N}_3$, sets the i-th element $\mathbf{M}_1(i, r)$ of the $\mathbf{M}_1(:, r)$ vector equal to the following:

$$\mathbf{M}_1(i, r) = \sum_{y=1}^{JK} \mathbf{X_{(1)}}(i, y)\mathbf{C}(\lceil \frac{y}{J} \rceil, r)\mathbf{B}(1 + (y-1)\%J, r),$$

which is exactly the equation that we want from the definition of $\mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B})$. □

Notice that in Algorithm 7.2, the largest dense matrix required is either $\mathbf{B}$ or $\mathbf{C}$ (not $\mathbf{C} \odot \mathbf{B}$ as in the naive case), and therefore we have effectively avoided the intermediate data explosion problem.

Discussion. Table 7.4 compares the cost of the naive algorithm and GIGATENSOR for computing $\mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B})$. The naive algorithm requires total $JKR + 2mR$ flops (JKR for constructing $(\mathbf{C} \odot \mathbf{B})$, and $2mR$ for multiplying $\mathbf{X_{(1)}}$ and $(\mathbf{C} \odot \mathbf{B})$), and $JKR + m$ intermediate data size (JKR for $(\mathbf{C} \odot \mathbf{B})$, and m for $\mathbf{X_{(1)}}$). On the other hand, GIGATENSOR requires only $5mR$ flops ($3mR$ for three Hadamard products, and $2mR$ for the final multiplication), and $max(J + m, K + m)$ intermediate data size. The dependence on the term JK of the naive method makes it inappropriate for real world tensors which are sparse and the sizes of dimensions are much larger compared to the number m of nonzeros ($JK \gg m$). On the other hand, GIGATENSOR depends on $max(J + m, K + m)$ which is $O(m)$ for most practical cases, and thus fully exploits the sparsity of real world tensors for computing the efficient tensor decomposition.

Algorithm	Flops	Intermediate Data	Example
Naive	$JKR + 2mR$	$JKR + m$	$1.25 \cdot 10^{16}$ Flops, 100 PB
GIGATENSOR	$5mR$	$max(J + m, K + m)$	$5.8 \cdot 10^{9}$ Flops, 1.2 GB

Table 7.4: Cost comparison of the naive and GIGATENSOR for computing $\mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B})$. J and K are the sizes of the second and the third dimensions, respectively, m is the number of nonzeros in the tensor, and R is the desired rank for the tensor decomposition (typically, $R \sim 10$). GIGATENSOR does not suffer from the intermediate data explosion problem, and is much more efficient than the naive algorithm. The example refers to the intermediate data size for NELL-1 dataset of Table 7.7, for 8 bytes per value and $R = 10$.

7.3.4 Our Optimizations for MapReduce

In this subsection, we describe MAPREDUCE algorithms for computing the three steps in Equations (7.7), (7.8), and (7.9).

Avoiding the Intermediate Data Explosion

The first step is to compute $\mathbf{M}_1 \leftarrow \mathbf{X_{(1)}}(\mathbf{C} \odot \mathbf{B})$ (Equation (7.7)). The factors $\mathbf{C}$ and $\mathbf{B}$ are given in the form of $< j, r, \mathbf{C}(j,r) >$ and $< j, r, \mathbf{B}(j,r) >$, respectively. The tensor $\mathcal{X}$ is stored in the format of $< i, j, k, \mathcal{X}(i,j,k) >$, but we assume the tensor data is given in the form of mode-1 matricization ($< i, j, \mathbf{X_{(1)}}(i,j) >$) by using the mapping in Equation (7.1). We use Q_i and Q^j to denote the set of nonzero indices in $\mathbf{X_{(1)}}(i,:)$ and $\mathbf{X_{(1)}}(:,j)$, respectively: i.e., $Q_i = \{j | \mathbf{X_{(1)}}(i,j) > 0\}$ and $Q^j = \{i | \mathbf{X_{(1)}}(i,j) > 0\}$.

We first describe the MAPREDUCE algorithm for line 7 of Algorithm 7.2. The reducer joins the tensor data and the factor data, and performs the Hadamard product. Notice that only the tensor $\mathcal{X}$ and the factor $\mathbf{C}$ are transferred in the shuffling stage.

- MAP-1: map $< i, j, \mathbf{X_{(1)}}(i,j) >$ on $\lceil \frac{j}{J} \rceil$, and $< j, r, \mathbf{C}(j,r) >$ on j such that tuples with the same key are shuffled to the same reducer in the form of $< j, (\mathbf{C}(j,r), \{(i, \mathbf{X_{(1)}}(i,j)) | \forall i \in Q^j\}) >$.
- REDUCE-1: take $< j, (\mathbf{C}(j,r), \{(i, X_{(1)}(i,j)) | \forall i \in Q^j\}) >$ and emit $< i, j, \mathbf{X_{(1)}}(i,j)\mathbf{C}(j,r) >$ for each $i \in Q^j$.

In the second MAPREDUCE algorithm for line 8 of Algorithm 7.2, we perform the similar task as the first MAPREDUCE job but we do not multiply the value of the tensor, since line 8 uses the binary function. Again, only the tensor $\mathcal{X}$ and the factor $\mathbf{C}$ are transferred in the shuffling stage.

- MAP-2: map $< i, j, \mathbf{X_{(1)}}(i,j) >$ on $\lceil \frac{j}{J} \rceil$, and $< j, r, \mathbf{B}(j,r) >$ on j such that tuples with the same key are shuffled to the same reducer in the form of $< j, (\mathbf{B}(j,r), \{i | \forall i \in Q^j\}) >$.
- REDUCE-2: take $< j, (\mathbf{B}(j,r), \{i | \forall i \in Q^j\}) >$ and emit $< i, j, \mathbf{B}(j,r) >$ for each $i \in Q^j$.

Finally, in the third MAPREDUCE algorithm for lines 9 and 10, we combine the results from the first and the second jobs using Hadamard product, and sums up each row to get the final result.

- MAP-3: map $< i, j, \mathbf{X_{(1)}}(i,j)\mathbf{C}(j,r) >$ and $< i, j, \mathbf{B}(j,r) >$ on i such that tuples with the same i are shuffled to the same reducer in the form of $< i, \{(j, \mathbf{X_{(1)}}(i,j)\mathbf{C}(j,r), \mathbf{B}(j,r))\} | \forall j \in Q_i >$.

Algorithm	Flops	Intermediate Data	Example
Naive	KR^2	$K(R^2+R)$	40 GB
GIGATENSOR	KR^2	$d \cdot R^2$	40 KB

Table 7.5: Cost comparison of the naive (column-wise partition) method and GIGATENSOR for computing $\mathbf{C}^T\mathbf{C}$. K is the size of the third dimension, d is the number of mappers used, and R is the desired rank for the tensor decomposition (typically, $R \sim 10$). Notice that although the flops are the same for both methods, GIGATENSOR has much smaller intermediate data size compared to the naive method, considering $K \gg d$. The example refers to the intermediate data size for NELL-1 dataset of Table 7.7, for 8 bytes per value, $R = 10$ and $d = 50$.

- REDUCE-3: take $< i, \{(j, \mathbf{X}_{(1)}(i,j)\mathbf{C}(j,r), \mathbf{B}(j,r))\} | \forall j \in Q_i >$ and emit $< i, \sum_j \mathbf{X}_{(1)}(i,j)\mathbf{C}(j,r)\mathbf{B}(j,r) >$.

Note that the amount of data traffic in the shuffling stage is small (2 times the nonzeros of the tensor $\mathcal{X}$), considering that $\mathcal{X}$ is sparse.

Parallel Outer Products

The next step is to compute $(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^\dagger$ (Equation (7.8)). Here, the challenge is to compute $\mathbf{C^TC} * \mathbf{B^TB}$ efficiently, since once the $\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B}$ is computed, the pseudo-inverse is trivial to compute because matrix $\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B}$ is very small ($R \times R$ where R is very small; e.g. $R \sim 10$). The question is, how to compute $\mathbf{C^TC} * \mathbf{B^TB}$ efficiently? Our idea is to first compute $\mathbf{C}^T\mathbf{C}$, then $\mathbf{B}^T\mathbf{B}$, and perform the Hadamard product of the two $R \times R$ matrices. To compute $\mathbf{C}^T\mathbf{C}$, we express $\mathbf{C}^T\mathbf{C}$ as the sum of outer products of the rows:

$$\mathbf{C}^T\mathbf{C} = \sum_{k=1}^{K} \mathbf{C}(k,:)^T \circ \mathbf{C}(k,:), \tag{7.10}$$

where $\mathbf{C}(k,:)$ is the kth row of the $\mathbf{C}$ matrix. To implement the Equation (7.10) efficiently in MAPREDUCE, we partition the factor matrices row-wise [Liu et al., 2010]: we store each row of $\mathbf{C}$ into a line in the HADOOP File System (HDFS). The advantage of this approach compared to the column-wise partition is that each unit of data is self-joined with itself, and thus can be independently processed; column-wise partition would require each column to be joined with other columns which is prohibitively expensive.

The MAPREDUCE algorithm for Equation (7.10) is as follows.

- Map: map $< j, \mathbf{C}(j,:) >$ on 0 so that all the output is shuffled to the only reducer in the form of $< 0, \{\mathbf{C}(j,:)^T \circ \mathbf{C}(j,:)\} \forall j >$.
- Combine, Reduce: take $< 0, \{\mathbf{C}(j,:)^T \circ \mathbf{C}(j,:)\} \forall j >$ and emit $< 0, \sum_j \mathbf{C}(j,:)^T \circ \mathbf{C}(j,:) >$.

Since we use the combiner as well as the reducer, each mapper computes the local sum of the outer product. The result is that the size of the intermediate data, i.e. the number of input tuples to the reducer, is very small ($d \cdot R^2$ where d is the number of mappers) in GIGATENSOR. On the other hand, the naive column-wise partition method requires KR (the size of $\mathbf{C}^T$) + K (the size of a column of $\mathbf{C}$) intermediate

Algorithm	Flops	Intermediate Data	Example
Naive	IR^2	$I(R+R^2)+R^2$	23 GB
GIGATENSOR	IR^2	IR^2	20 GB

Table 7.6: Cost comparison of the naive (column-wise partition) method and GIGATENSOR for multiplying $\mathbf{X_{(1)}}(\mathbf{C}\odot\mathbf{B})$ and $(\mathbf{C}^T\mathbf{C}*\mathbf{B}^T\mathbf{B})^\dagger$. I is the size of the first dimension, and R is the desired rank for the tensor decomposition (typically, $R\sim 10$). Notice that although the flops are the same for both methods, GIGATENSOR has smaller intermediate data size compared to the naive method. The example refers to the intermediate data size for NELL-1 dataset of Table 7.7, for 8 bytes per value and $R=10$.

data for 1 iteration, and thereby requires $K(R^2+R)$ intermediate data for R iterations, as summarized in Table 7.5.

Distributed Cache Multiplication

The final step is to multiply $\mathbf{X_{(1)}}(\mathbf{C}\odot\mathbf{B})\in\mathbb{R}^{I\times R}$ and $(\mathbf{C}^T\mathbf{C}*\mathbf{B}^T\mathbf{B})^\dagger\in\mathbb{R}^{R\times R}$ (Equation (7.9)). We note that the first matrix $\mathbf{X_{(1)}}(\mathbf{C}\odot\mathbf{B})$ is large and does not fit in the memory of a single machine, while the second matrix $(\mathbf{C}^T\mathbf{C}*\mathbf{B}^T\mathbf{B})^\dagger$ is very small to fit in the memory. Thus, we use the distributed cache multiplication (Section 6.4.6) to broadcast the second matrix to all the mappers that process the first matrix, and perform join in the first matrix. The result is that our method requires only one MAPREDUCE job with smaller intermediate data size (IR^2). On the other hand, the standard naive matrix-matrix multiplication requires two MAPREDUCE jobs (the first job for grouping the data by the column id of $\mathbf{X_{(1)}}(\mathbf{C}\odot\mathbf{B})$ and the row id of $(\mathbf{C}^T\mathbf{C}*\mathbf{B}^T\mathbf{B})^\dagger$, and the second job for aggregation) with larger intermediate data size ($IR+R^2$ for the first job, and IR^2 for the second job), as summarized in Table 7.6.

7.4 Experiments

To evaluate our system, we perform experiments to answer the following questions:

Q1 What is the scalability of GIGATENSOR compared to other methods with regard to the sizes of tensors?
Q2 What is the scalability of GIGATENSOR compared to other methods with regard to the number of nonzero elements?
Q3 How does GIGATENSOR scale with regard to the number of machines?
Q4 What are the discoveries on real world tensors?

The tensor data in our experiments are summarized in Table 7.7, with the following details.

- NELL: real world knowledge base data containing (noun phrase 1, context, noun phrase 2) triples (e.g. 'George Harrison' 'plays' 'guitars') from the 'Read the Web' project [Carlson et al., 2010]. NELL-1 data is the full data, and NELL-2 data is the filtered data from NELL-1 by removing entries whose values are below a threshold.
- Random: synthetic random tensor of size $I\times I\times I$. The size I varies from 10^4 to 10^9, and the number of nonzeros varies from 10^2 to 10^8.

Data	I	J	K	Nonzeros
NELL-1	26 M	26 M	48 M	144 M
NELL-2	15 K	15 K	29 K	77 M
Random	10 K $\sim$ 1 B	10 K $\sim$ 1 B	10 K $\sim$ 1 B	100 $\sim$ 20 M

Table 7.7: Summary of the tensor data used. B: Billion, M: Million, K: Thousand.

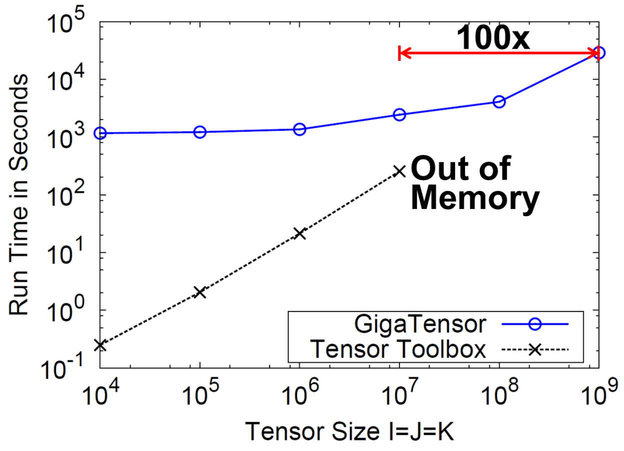


Figure 7.5: The scalability of GIGATENSOR compared to the Tensor Toolbox with regard to the number of nonzeros and tensor sizes on the synthetic data. For a tensor of size $I \times I \times I$, we set the number of nonzero elements to be $I/50$. Notice that GIGATENSOR decomposes tensors of sizes up to 10^9 and beyond, while the Tensor Toolbox 'dies', running out of memory on tensors of sizes beyond 10^7.

7.4.1 Scalability

Scalability on the Size of Tensors. Figures 7.1 shows the scalability of GIGATENSOR with regard to the sizes of tensors. We choose the Tensor Toolbox for Matlab [Bader and Kolda, 2007a] as our baseline for comparison, since it is the current state of the art. We fix the number of nonzero elements to 10^4 on the synthetic data while increasing the tensor sizes $I = J = K$. We use 35 reducers for running GIGATENSOR. Notice that GIGATENSOR solves $100\times$ larger problem than the Tensor Toolbox which runs out of memory on tensors of sizes beyond 10^7. We performed the same experiment while fixing the nonzero elements to 10^7, and we get the similar results.

Scalability on the Number of Nonzero Elements. Figure 7.5 shows the scalability of GIGATENSOR compared to the Tensor Toolbox with regard to the number of nonzeros and tensor sizes on synthetic data. We set the tensor size to be $I \times I \times I$, and the number of nonzero elements to be either (a) $I/50$ or (b) $I/100$. We use 35 reducers for running GIGATENSOR. Notice that GIGATENSOR decomposes tensors of sizes up to 10^9, while the Tensor Toolbox implementation runs out of memory on tensors of sizes beyond 10^7. We choose to display only the case where the number of nonzeros is $I/50$ since the $I/100$ case was almost identical.

Scalability on the Number of Machines. Figure 7.6 shows the scalability of GIGATENSOR with regard to the number of machines. Notice that the running time scales up near linearly.

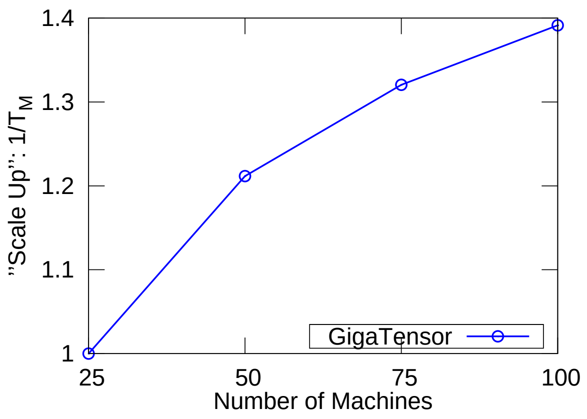


Figure 7.6: The scalability of GIGATENSOR with regard to the number of machines on the NELL-1 data. Notice that the running time scales up near linearly.

7.4.2 Discovery

In this section, we present discoveries on the NELL dataset that was previously introduced; we are mostly interested in demonstrating the power of our approach, as opposed to the current state of the art which was unable to handle a dataset of this magnitude. We perform two tasks: concept discovery, and the synonym detection in the knowledge base tensor.

Concept Discovery. With GIGATENSOR, we decompose the NELL-2 dataset in $R = 10$ components, and obtained $\lambda_i, \mathbf{A}, \mathbf{B}, \mathbf{C}$ (see Figure 7.2). Each one of the R columns of $\mathbf{A}, \mathbf{B}, \mathbf{C}$ represents a grouping of similar (noun-phrase np1, noun-phrase np2, context words) triplets. The r-th column of $\mathbf{A}$ encodes with high values the noun-phrases in position np1, for the r-th group of triplets, the r-th column of $\mathbf{B}$ does so for the noun-phrases in position np2 and the r-th column of $\mathbf{C}$ contains the corresponding context words. In order to select the most representative noun-phrases and contexts for each group, we choose the k highest valued coefficients for each column. Table 7.8 shows 4 notable groups out of 10, and within each group the 3 most outstanding noun-phrases and contexts. Notice that each concept group contains relevant noun phrases and contexts.

Synonym Detection. The lower dimensional embedding of the noun phrases also permits a scalable and robust strategy for synonym detection. We are interested in discovering noun-phrases that occur in similar contexts, i.e. *contextual synonyms*. Using a similarity metric, such as Cosine Similarity, between the lower dimensional embeddings of the Noun-phrases (such as in the factor matrix $\mathbf{A}$), we can identify similar noun-phrases that can be used alternatively in sentence templates such as *np1 context np2*. Using the embeddings in the factor matrix $\mathbf{A}$ (appropriately column-weighted by λ), we get the synonyms that might be used in position *np1*, using $\mathbf{B}$ leads to synonyms for position *np2*, and using $\mathbf{C}$ leads to contexts that accept similar *np1* and *np2* arguments. In Table 7.2, which is located in Section 7.1, we show some exemplary synonyms for position *np1* that were discovered by this approach on NELL-1 dataset. Note that these are not synonyms in the traditional definition, they are phrases that may occur in similar semantic roles in a sentence.

Noun Phrase 1	Noun Phrase 2	Context
Concept 1:“Web Protocol”		
internet	protocol	‘np1’ ‘stream’ ‘np2’
file	software	‘np1’ ‘marketing’ ‘np2’
data	suite	‘np1’ ‘dating’ ‘np2’
Concept 2:“Credit Cards”		
credit	information	‘np1’ ‘card’ ‘np2’
Credit	debt	‘np1’ ‘report’ ‘np2’
library	number	‘np1’ ‘cards’ ‘np2’
Concept 3: “Health System”		
health	provider	‘np1’ ‘care’ ‘np2’
child	providers	‘np’ ‘insurance’ ‘np2’
home	system	‘np1’ ‘service’ ‘np2’
Concept 4: “Family Life”		
life	rest	‘np2’ ‘of’ ‘my’ ‘np1’
family	part	‘np2’ ‘of’ ‘his’ ‘np1’
body	years	‘np2’ ‘of’ ‘her’ ‘np1’

Table 7.8: Four notable groups that emerge from analyzing the NELL dataset.

7.5 Related Work

In this section, we review related works on tensor analysis, emphasizing on data mining applications. Tensors have a very long list of applications, in addition to data mining. For instance, tensors have been used extensively in Chemometrics [Bro, 1997] and Signal Processing [Sidiropoulos et al., 2000]. Some of the data mining applications that employ tensors are the following: in [Kolda and Bader, 2006], Kolda et al. extend the famous HITS algorithm [Kleinberg, 1999] in order to incorporate topical information in the links between Web pages. In [Acar et al., 2007], Acar et al. analyze epilepsy data using tensor decompositions. In [Bader et al., 2006], Bader et al. employ tensors in order perform social network analysis, using the Enron dataset for evaluation. In [Sun et al., 2005b], Sun et al. formulate click data on Web pages as a tensor, in order to improve Web search by incorporating user interests in the results. In [Chew et al., 2007], Chew et al. extend the Latent Semantic Indexing [Deerwester et al., 1990] paradigm for cross-language information retrieval, using tensors. In [Tao et al., 2008], Tao et al. employ tensors for 3D face modeling and in [Tao et al., 2007], a supervised learning framework, based on tensors is proposed. In [Maruhashi et al., 2011], Maruhashi et al. present a framework for discovering bipartite graph like patterns in heterogeneous networks using tensors.

7.6 Conclusion

In this chapter, we propose GIGATENSOR, a tensor decomposition algorithm which scales to billion size tensors, and present interesting discoveries from real world tensors. Our major contributions include:

- **Algorithm.** We propose GIGATENSOR, a carefully designed large scale tensor decomposition algorithm on MAPREDUCE.
- **Scalability.** GIGATENSOR decomposes $100\times$ larger tensors compared to previous methods, and GIGATENSOR scales linearly to the number of machines.
- **Discovery.** We discover patterns of synonyms and concept groups in a very large knowledge base tensor which could not be analyzed before.

Part III

Graph Management

Part III - Graph Management: Overview

Given a very large graph, how do we store and index it in distributed systems so that graph mining queries can be answered quickly? How do we lay out edges of a graph so that the adjacency matrix can be compressed well? In this part we describe algorithms for large graph management in MAPREDUCE/HADOOP.

First, we describe GBASE, a scalable and general graph management system. GBASE provides a parallel indexing mechanism for graph mining operations that both saves storage space, as well as accelerates queries.

Second, we study edge layout problem: given a graph, reorder nodes so that the nonzeros (edges) of the adjacency matrix are well-clustered. Better layout of edges of graphs leads to better compression; however, existing methods based on the assumptions of the existence of clear-cut communities do not work nicely on real world graphs. We propose SLASHBURN, an edge layout algorithm which utilizes the characteristic of real world graphs to compress graphs well.

Chapter 8

Graph Storage and Indexing

How to store large graphs efficiently? What are the core operations/queries on those graph? How to answer the graph queries quickly? We propose GBASE, a scalable and general graph management system. The key novelties lie in 1) our storage and compression scheme for a parallel setting and 2) the carefully chosen graph operations and their efficient implementation. We design and implement an instance of GBASE using MAPREDUCE/HADOOP. GBASE provides a parallel indexing mechanism for graph mining operations that both saves storage space, as well as accelerates queries. We run numerous experiments on real graphs, spanning *billions* of nodes and edges, and we show that our proposed GBASE is indeed fast, scalable and nimble, with significant savings in space and time.

8.1 Introduction

Our goal is to build a general graph management system in parallel, distributed settings to support billion-scale graphs for various applications. For the goal, we address the following problems:

1. **Storage**. How can we efficiently store and manage large graphs in parallel, distributed settings to answer graph queries efficiently? How should we split the edges into smaller units? How should we group the units into files?
2. **Algorithms**. How can we define common, core algorithms to satisfy various graph applications?
3. **Query Optimization**. How can we exploit the efficient storage and general algorithms to execute queries efficiently?

For all the problems, *scalability* is a major challenge. The size of graphs has been experiencing an unprecedented growth. For example, one of the graphs we use here, the *Yahoo Web graph* from 2002, has more than 1 *billion* nodes and almost *7 billion* edges. Similar size, or even larger graphs, exist: the Twitter graph spans several Terabytes; click-streams are reported to reach Petabyte scale [Liu et al., 2009]. Such large graphs violate the assumption that the graph can be fit in main memory or at least the disk of a single workstation, on which most of existing graph algorithms have been built. Thus, we need to re-think those algorithms, and to develop scalable, parallel ones, to manage graphs that span Tera-bytes and beyond.

Our Contributions. We propose GBASE, a scalable and general graph management system, to address the above challenges. The main contributions are the following:

1. **Storage**. We propose a novel graph storage method called 'block compression' to efficiently store homogeneous regions of graphs. We also propose a grid based method to efficiently place blocks into files. We run our algorithm on billion-scale graphs and show that the block compression method leads up to $50\times$ less storage and faster running time. Our block compression method is agnostic to the underlying storage mechanism, which can be applied to distributed file systems as well as relational databases.
2. **Algorithms**. We identify a core graph operation, and use it to formulate *seven* different types of graph queries including neighborhood, induced subgraph, egonet, K-core, and cross-edges. The novelty is in formulating edge-based queries (induced subgraph) as well as node-based queries (neighborhoods) using a unified framework.
3. **Query Optimization**. We propose a grid selection strategy to minimize disk accesses and answer queries quickly. We also propose a MAPREDUCE algorithm to support incidence matrix based queries using the original adjacency matrix, without explicitly building the incidence matrix.

The rest of this chapter is organized as follows. We first present the overall framework in Section 8.2. We describe the storage and indexing method in Section 8.3, and then the query execution in Section 8.4. We provide experimental evaluations and comparisons in Section 8.5. After reviewing the related work in Section 8.6, we conclude in Section 8.7.

8.2 Overall Framework

The overall framework of our GBASE is summarized in Figure 8.1. The design objective is to balance storage efficiency and query performance on large graphs. It comprises of two components: the indexing stage and the query stage. In this section, we give a high level overview of each stage; and we will give more details in Sections 8.3 and 8.4, respectively.

In the indexing stage, given the original raw graph which is stored as a big edge file, GBASE first partitions it into several homogeneous blocks. Second, according to the partition results, we reshuffle the nodes so that the nodes belonging to the same partition are put nearby. Third, we compress all non-empty block through standard compression such as GZip. Finally, we store the compressed blocks, together with some meta information (e.g., the block row id and column id, and all the encoded node ids), into the graph databases. For many real graphs, such homogeneous blocks, community-like structure, do exist. Therefore, after partition and reshuffling, the resulting blocks are either relatively dense (e.g., the diagonal blocks in Figure 8.1) or very sparse (e.g., the off-diagonal blocks in Figure 8.1). Both cases are space efficient for compression (i.e., the compression ratio is high). In the extreme case that a given block is empty, we do not store it at all. Our experiments (See Section 8.5) show that in some cases, we only need less than 2% storage space of the original after the indexing stage.

In the query stage, our goal is to provide a set of core operations that will be sufficient to support a diverse set of graph applications, e.g., ranking, community detection, anomaly detection, and etc. The key of the on-line query stage is the query execution engine, which unifies the different types of inputs as query vectors. It also unifies the (seemingly) different types of operations on the graph by a unified matrix-vector multiplication which we will introduce in Section 8.4. By doing so, GBASE is able to support multiple different types of queries simultaneously. Table 8.1 summarizes the queries (the first column) that are

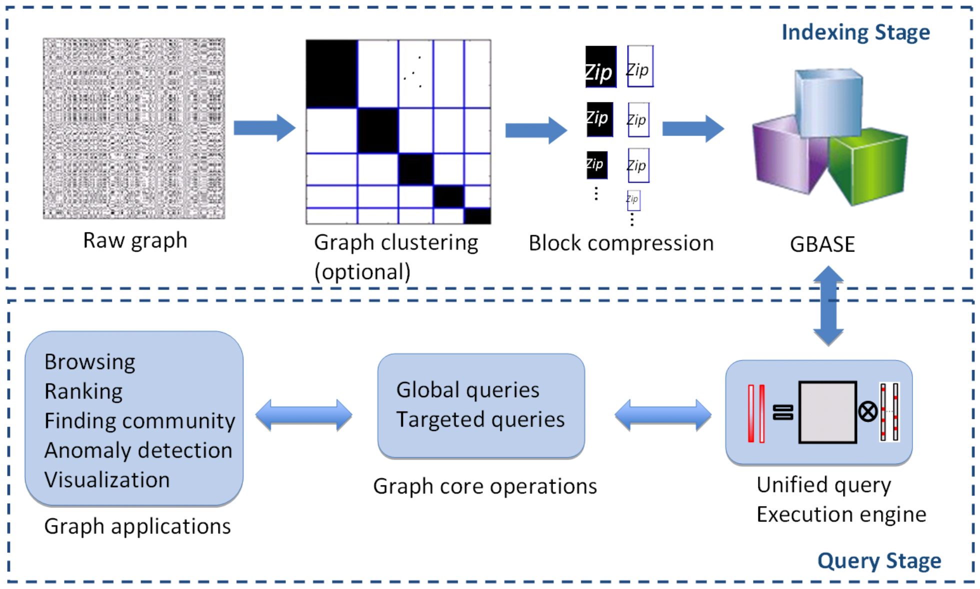


Figure 8.1: Overall framework of GBASE. 1. Indexing Stage: raw graph is clustered and divided into compressed blocks. 2. Query Stage: global and targeted queries from various graph applications are handled by a unified query execution engine.

Query \ **Applications**	**Browsing**	**Ranking**	**Finding Community**	**Anomaly Detection**	**Visualization**
Connected Comp.			✓	✓	
Radius				✓	✓
PageRank, RWR	✓	✓		✓	
Induced Subgraph	✓		✓		✓
(K)-Neighborhood	✓		✓		✓
(K)-Egonet	✓		✓	✓	✓
K**-core**			✓		✓
Cross-edges				✓	✓

Table 8.1: Applications of GBASE. Notice that GBASE answers wide range of both global (top 3 rows) and targeted queries (bottom 5 rows with bold fonts) with applications in browsing [Page et al., 1998, Tong et al., 2006, Lin et al., 2009], ranking [Page et al., 1998, Tong et al., 2006], finding communities [Kang et al., 2009, Lin et al., 2009], anomaly detection [Sun et al., 2005a, Kang et al., 2009, 2010, Akoglu et al., 2010], and visualization [Lin et al., 2009, Alvarez-Hamelin et al.].

Symbol	Definition
A	adjacency matrix of the graph G
B	incidence matrix of the graph G
n	number of nodes
m	number of edges
k	number of partitions
p, q	partition indices, $1 \leq p, q \leq k$
$I^{(p)}$	set of nodes belonging to the p-th partition
$l^{(p)}$	partition size, $l^{(p)} \equiv \lvert I^{(p)} \rvert$, $1 \leq p \leq k$
$\mathcal{G}^{(p,q)}$	subgraphs induced by p-th and q-th partitions
$m^{(p,q)}$	number of edges in $\mathcal{G}^{(p,q)}$

Table 8.2: Table of symbols.

supported by GBASE. These queries construct the main building blocks for a variety of important graph applications (Table 8.1). For example, the diversity of Random Walk with Restart (RWR) [Tong et al., 2006] scores among the neighborhood of a given edge/node is a strong indicator of abnormality of that node/edge [Sun et al., 2005a]. The ratio between the number of edges (or the summation of edge weights) and number of nodes within the egonet can help find abnormal nodes on weighted graphs [Akoglu et al., 2010]. The K-cores and cross-edges can be used for visualization and finding communities in large graphs.

8.3 Graph Storage and Indexing

In this section, we describe in details the indexing and storage stage of GBASE. We use the symbols in Table 8.2.

8.3.1 Baseline Storage Scheme

A typical way to store the raw graph is to use the adjacency list format: for each node, it saves all the out-neighbors adjacent from the node. The adjacency list format is simple and might be good for answering out-neighbor queries. However, it is not efficient format for answering general queries including in-neighbor queries and ego-net queries as we will see in Section 8.4. For the reason, we instead use the sparse adjacency matrix format, where we save each edge by a (source,destination) pair. The advantage of the sparse adjacency matrix format is its generality and flexibility to enable efficient storage and indexing techniques as we will see later in this and the next sections.

The storage system should be designed to be efficient in both storage cost and on-line query response. To this end, we propose to index and store the graph on the homogeneous block, community-like structure, levels. Next, we will describe how to *form*, *compress* and *store/place* such blocks.

8.3.2 Block Formulation

The first step is to partition the graph, i.e., re-order the rows and columns, and make homogeneous regions into blocks. Partitioning algorithms form an active research area, and finding optimal partitions is orthogonal to our work. Any partition algorithms, e.g., METIS [Karypis and Kumar, 1999a], Disco [Papadimitriou and Sun, 2008], and SLASHBURN (Section 9), can be naturally plugged into GBASE.

Graph partitioning can be formally defined as follows. The input is the original raw graph denoted by G. Given a graph G, we partition the nodes into k groups. The set of nodes that are assigned into the p-th partition for $1 \leq p \leq k$ is denoted by $I^{(p)}$. The subgraph or block induced by p-th source partition and q-th destination partition is denoted as $\mathcal{G}^{(p,q)}$. The sets $I^{(p)}$ partition the nodes, in the sense that $I^{(p)} \cap I^{(p')} = \emptyset$ for $p \neq p'$, while $\bigcup_p I^{(p)} = \{1, \ldots, n\}$. In terms of storage, the objective is to find the optimal k partitions which lead to smallest total storage cost of all blocks/subgraphs $\mathcal{G}^{(p,q)}$ where $1 \leq p, q \leq k$. Intuitively, we want the induced subgraphs to be homogeneous (meaning the subgraphs are either very dense or very sparse), which captures not only community structure but also leads to small storage cost.

For many real graphs, the community/clustering structure can be naturally identified. For instance, in Web graphs, the lexicographic ordering of the URL can be used as an indicator of community [Boldi and Vigna, 2004] since there are usually more intra-domain links compared with the inter-domain links. For authorship network, the research interest is often a good indicator to find communities since authors with the same or similar research interest tend to have more collaborations. For patient-doctor graph, the patient information (e.g., geography, disease type, etc) can be used to find the communities (patients with similar disease and living in the same neighborhood have higher chance to visit the same doctor).

8.3.3 Block Compression

The homogeneous block representation provides a more compact representation of the original graph. It enables us to encode the graph in a more efficient way. The encoding of a block $\mathcal{G}^{(p,q)}$ consists of the following information:

- source and destination partition ID p and q;
- the set of sources $I^{(p)}$and the set of destinations $I^{(q)}$.
- the payload, the bit string of subgraph $\mathcal{G}^{(p,q)}$.

A naive way of encoding a block is *raw block encoding* which only stores the coordinates of the non-zero entries in the block, as described in Section 4.3.2. Although this method saves the storage space since the nonzero elements within the block can be encoded with a smaller number of bits ($\log(\max(l^{(p)}, l^{(q)})$) than the original, the savings are not great.

To achieve better compression, we propose *zip block encoding* which converts the adjacency matrix of the subgraph into a binary string and stores the compressed string as the payload. Compared to the raw block encoding, the zip block encoding requires more cpu time to zip and unzip blocks. However, the storage savings and the reduced data transfer size help to improve performance of GBASE as we will see in Section 8.5.

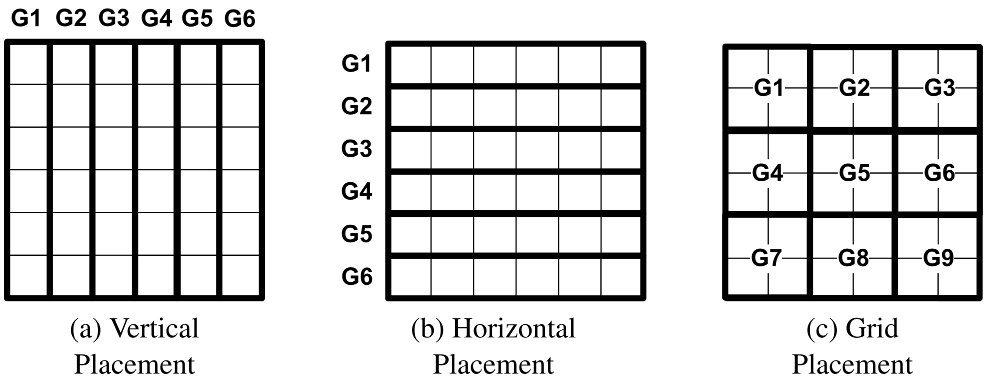


Figure 8.2: Adjacency matrices showing possible placement of blocks into files in HADOOP. The smallest rectangle represents a block in the adjacency matrix. The placement strategy determines which of the blocks are grouped into files G1 to G6 or G9. Vertical placement in (a) is good for in-neighbor queries, but inefficient for out-neighbor or egonet queries. Horizontal placement in (b) is good for out-neighbor queries, but inefficient for in-neighbor or egonet queries. GBASE uses the grid placement, shown in (c), which is efficient for all types of queries.

For example, we have the following adjacency matrix of a graph:

$$\mathcal{G} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix}. \tag{8.1}$$

Raw block encoding will just store the non-zero coordinates $(0,0)$, $(1,0)$, $(2,1)$, and $(2,2)$ as the payload. Zip block encoding will converts the matrix into a binary string $110,001,001$ (in the column major order) and then use the compression of this string as the payload.

Storage Estimation. The storage needed for raw block encoding is $2 * m^{(p,q)} * \log(max(l^{(p)}, l^{(q)}))$. The storage needed for zip block encoding is $l^{(p)} l^{(q)} H(d)$, where $d = \frac{m^{(p,q)}}{l^{(p)} l^{(q)}}$ is the density of $\mathcal{G}^{(p,q)}$. $H(\cdot)$ is the Shannon entropy function: $H(X) = -\sum_x p(x) \log p(x)$ where $p(x)$ is the probability that $X = x$. Note that the number of bits to encode an edge in zip block encoding decreases as d increases, while it is constant in raw block encoding.

8.3.4 Block Placement

After we compress the blocks, we need to store/place them in the file system (e.g., HDFS of HADOOP, relational DB). Here, the main idea is to place several blocks together into a file, and select only relevant files as inputs in the query stage. The question is, how do we place blocks into files? A typical approach is to use vertical placement to place the vertical blocks in a file as shown in Figure 8.2 (a). The other alternative is to use horizontal placement to place the horizontal blocks in a file as shown in Figure 8.2 (b). However, both of the placement techniques are good only for one type of query: for example, horizontal and vertical placement is good for out-neighbor and in-neighbor queries, respectively.

To solve the problem, GBASE uses the grid placement as shown in Figure 8.2 (c). The advantage of the grid placement is that it can answer various types of queries efficiently as we will see in Section 8.4.

Suppose we store all the compressed blocks in K files. With vertical/horizontal placement, we need $O(K)$ file accesses to find the in- and out-neighbors of a given query node. In contrast, we only need $O(\sqrt{K})$ files accesses with grid placement.

8.4 Handling Graph Queries

In this section, we describe query execution in GBASE. GBASE supports both "global" queries, as well as "targeted" queries for one or a few specific nodes. The answer to global queries requires traversal of the whole graph, like, e.g., diameter estimation. In contrast, "targeted" queries need to access only parts of the graph. GBASE supports seven different queries including neighborhoods, induced subgraphs, egonets, K-core, and cross-edges.

8.4.1 Global Queries

Global queries are performed by repeated joins of edge blocks and vector blocks. GBASE supports the following graph queries: degree distribution, PageRank, RWR ("Random Walk with Restart"), radius estimations, and discovery of connected components. Our contribution here is that our proposed storage and compression scheme reduce the graph storage significantly, and enable faster running time as shown in Figure 8.5. The global queries also serve as primitives for targeted queries (see 'T6: K-core' in Section 8.4.2), enabling a variety of applications as shown in Table 8.1.

8.4.2 Targeted Queries

Many graph mining operations can be unified as matrix-vector multiplication. Here that matrix is either the adjacency matrix A of size $n \times n$ or the incidence matrix B of size $m \times n$ where n and m are the number of nodes and edges in the graph, respectively. Each row of the incidence matrix corresponds to an edge, and it has two non-zeros whose column ids are the node ids of the edge.

The matrix-vector multiplication observation has the extra benefit that it corresponds to an SQL join, as we mentioned earlier in Section 4.2.1. Thus, graph mining could use all the highly optimized join algorithms in the literature (hash join, indexed join etc), while still leverages the proposed block compression storage scheme.

In fact, for each of the upcoming primitives, we shall first give the matrix-vector details, and then the SQL code.

T1: 1-step neighbors. The first query is to find 1-step in-neighbors and out-neighbors of a query node v.

Matrix-Vector version

Given a query node v, its 1-step in-neighbors can be found by the following matrix-vector multiplication:

$$in^1(v) = A \times e_v, \tag{8.2}$$

where the matrix is the adjacency matrix of the graph A and the vector is the 'indicator vector' e_v which is the n-vector whose v-th element is 1, and all other elements are 0. The 1-step in-neighbors of the query node v are those nodes whose corresponding values in $in^1(v)$ are 1s.

The 1-step out-neighbors can be obtained in the similar way by replacing A with its transpose A^T.

SQL version

We can also find 1-step in-neighbors and out-neighbors by the standard SQL. Assume we have a table `E(src, dst)` storing the edges, with attributes 'source' (`src`) and 'destination' (`dst`). The 1-step out-neighbors of a query node 'q' are given by

```
SELECT dst
FROM E
WHERE src='q'
```

without even requiring a join. 1-step in-neighbor can be answered in a similar way.

T2: K-step neighbors. The next query is to find 'within k-step' neighbors. Let us only consider the k-step in-neighbors. k-step out-neighbors can be found in similar way - we only need to replace the matrix A by its transpose A^T in the matrix-vector multiplication version; and switch `src` and `dst` in the SQL version.

Matrix-Vector version

The k-step in-neighbors $nh^k(v)$ of the query node v is defined recursively by $(k-1)$-step neighbors $nh^{k-1}(v)$ in terms of matrix-vector multiplication as follows:

$$nh^k(v) = A \times nh^{k-1}(v), \tag{8.3}$$

where the 0-step in-neighbors $nh^0(v)$ is just the indicator vector e_v. After the k multiplications, the k-step in-neighbors are those nodes whose corresponding values in $nh^k(v)$ or $nh^{k-1}(v)$ are 1s.

SQL version

As before, assume we have a table `E` with attributes `src` and `dst`. The k-step in-neighbors can also be found by SQL join. In general, the k-step in-neighbors is a $(k-1)$-way join. For example, the 2-step in-neighbors of a query node 'q' is given by the following SQL join:

```
SELECT E2.src
FROM E as E1, E as E2
WHERE E1.dst='q'
   AND E1.src = E2.dst
```

T3: Induced subgraph. Given a set of nodes V_q in a graph G, the induced subgraph is defined to be a graph whose nodes are V_q and an edge between two nodes v_1 and v_2 exist if they are adjacent in G.

Matrix-Vector version

Let B be the $m \times n$ incidence matrix where m and n are the number of edges and the nodes of the graph, respectively. A row of B corresponds to an edge (i, j), and the elements of the row are 0 except the ith and jth elements which are 1. Let e_{vq} be the n-vector, whose corresponding elements for V_q are 1s, and 0s otherwise.

Then, the induced subgraph $S(V_q)$ from V_q is expressed by the following matrix-vector multiplication:

$$S(V_q) = B \times e_{vq}, \tag{8.4}$$

where the resulting vector $S(V_q)$ is m-vector and the elements in $S(V_q)$ have values of 0, 1, or 2. The induced subgraph is given by those edges whose corresponding values in $S(V_q)$ are 2s since it means that the incident nodes (both the source and the target) of the edges are in V_q.

SQL version

Assume we have an incidence matrix as table B, with attributes eid, $srcid$, and $dstid$, representing the edge id, the source node id, and the destination id of a row in the incident matrix, respectively. Also assume we have a query vector table Q with an attribute $nodeid$. Then the induced subgraph is given by the following join:

```
SELECT B.eid, B.srcid, B.dstid
FROM B, Q as Q1, Q as Q2
WHERE B.srcid=Q1.nodeid
   AND B2.dstid=Q2.nodeid
```

T4: 1-step egonet. Informally, the 1-step-away egonet (or just 'egonet') of a node v is its 1-step-away vicinity. Formally, it is defined as the induced subgraph that includes v and its 1-step neighbors. Extracting the egonet of a query node v is a special case of extracting induced subgraph. That is, the set of nodes V_q is defined to be the v and its 1-step in-neighbors and out-neighbors.

The details are omitted, since we can combine earlier expressions (for both the matrix-vector case, as well as for the SQL case).

T5: K-step egonet. K-step egonet of a node v is defined to be the induced subgraph from v and its within-k step neighbors. Extracting the k-step egonet of a query node v is also a special case of extracting induced subgraph. That is, the set of nodes V_q is defined to be the v and its within-k step neighbors. Thus, the same expression for the k-step neighbors and the induced subgraph can be used for extracting k-step egonet.

T6: K-core. K-core of a graph is a maximal connected subgraph in which all vertices have degree at least K [Alvarez-Hamelin et al.]. K-core is useful for finding communities and visualizing graphs. Although it seems complicated at first, all K-cores of a large graph can be enumerated by GBASE using primitives defined before:

1. Compute degrees of all nodes. Let C be the set of nodes with degree $\geq K$.
2. Compute induced subgraph G' using C.
3. Find connected components of G'. The resulting components are the K-core.

T7: Cross-edges. Given two disjoint sets V_1 and V_2 of nodes, how can we find the cross edges connecting the two sets? Cross-edges are useful for visualizing the interaction of two distinct sets of nodes, as well as anomaly detection (e.g., a set of nodes having few edges to the rest of the world are suspicious). Cross-edges can be computed by GBASE using induced subgraph queries:

1. Computed induced subgraphs $S(V_1)$, $S(V_2)$, $S(V_1 \cup V_2)$ using nodes in V_1, V_2, and $(V_1 \cup V_2)$, respectively.
2. Let E_1, E_2, and E_{12} be the set of edges in $S(V_1)$, $S(V_2)$, and $S(V_1 \cup V_2)$, respectively. The cross edges are exactly the edges in $E_{12} - E_1 - E_2$.

8.4.3 Query Execution Engine

We describe the query execution engine of GBASE built on the top of HADOOP.

Overview. As described in previous sections, the main operation of GBASE is the matrix-vector multiplication. GBASE handles queries by executing appropriate block matrix-vector multiplication modules. The global queries are typically handled by multiple matrix-vector multiplications since the answer to the queries is often a fixed point of the multiplication (e.g., the first eigenvector in case of PageRank). The local queries require one or few multiplications.

Most of the operations require the adjacency matrix of the graph. Thus, GBASE uses the adjacency matrix directly as its input. However, some operations including the induced subgraph require the incidence matrix which is different from the adjacency matrix. We will see how to handle the queries requiring incidence matrix efficiently at the end of this subsection.

Grid Selection. Before running the matrix-vector multiplication, GBASE selects the grids containing the blocks relevant to the queries. Only the files corresponding to the grids are fed into HADOOP jobs that GBASE executes. For global queries, we need to select all the grids since all the blocks are relevant. For targeted queries, however, we can select only relevant grids. For in-neighbor queries, we select grids whose column range contains the query node as shown in Figure 8.3 (a). For out-neighbor queries, we select grids whose row range contains the query node as shown in Figure 8.3 (b). For egonet queries, we select grids whose row or column range contains the query. As we will see in Section 8.5, this grid selection has advantages of decreasing the running time.

Handling Incidence Matrix Queries. While the majority of operations use the adjacency matrix, the induced subgraph queries use the incidence matrix. Thus, GBASE need to access the incidence matrix to support the queries. A naive approach is to build the incidence matrix $B^{m \times n}$ by numbering edges sequentially. However, it requires the storage to save B which is twice the size of the original adjacency matrix. The question is, can we answer incidence matrix queries efficiently without the additional storage?

Our proposed main idea is to derive the incidence matrix from the original adjacency matrix as required. That is, an adjacency matrix element (src, dst) can be interpreted as $([src, dst], src)$ and $([src, dst], dst)$ of the incidence matrix where $[src, dst]$ is the edge id. Thus, the query execution algorithm for handling incidence matrix can work on the original adjacency matrix by treating each adjacency matrix element as two incidence matrix elements.

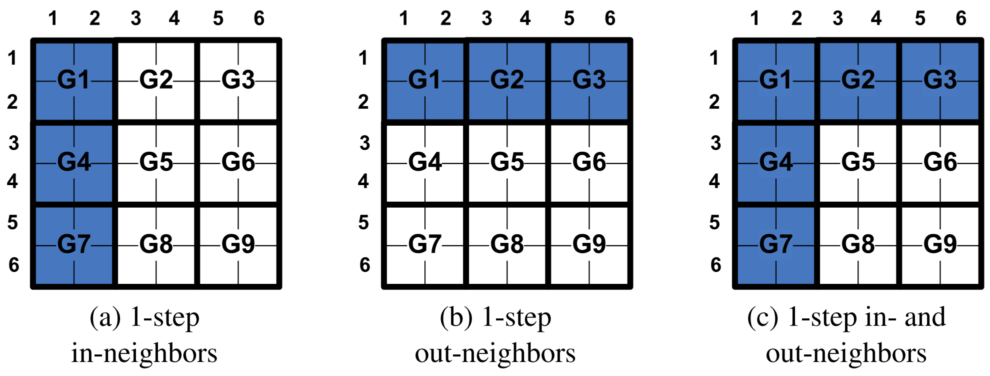


Figure 8.3: Grid selection in 6 by 6 blocks where the query node belongs to the second block. The smallest rectangle corresponds to a block, and a bigger rectangle containing 4 blocks is a grid which is saved in a file. Notice that GBASE selects different grids based on the type of the query and the query node id. For example, GBASE selects G1, G4, and G7, instead of all the grids for in-neighbors query. This reduced input size results in the decreased running time.

The HADOOP algorithm for the induced subgraph, which reflects the main idea, is shown in Algorithm 8.1. The algorithm is composed of two stages. In the first stage, the elements in the incidence matrix and the query vector are grouped together to generate partial results. Notice that two incidence matrix elements are generated (lines 5,6 of Algorithm 8.1) for an adjacency matrix element. In the second stage, the partial results are summed to get the final result. Note that only edges having the sum 2 are included in the egonet since it means that the two incidence nodes of the edges are contained in the query node set.

8.5 Experiments

To evaluate our GBASE system, we perform experiments to answer the following questions:

Q1 How much does our zip block encoding reduce the data size?
Q2 How do our algorithms scale up with the graph sizes and the number of machines?
Q3 How do our indexing and query execution methods save query response time?

Datasets. We use large graph datasets summarized in Table 8.3 whose entries are repeated from Table 2.2 for convenience. In order to show the performance across different data scales, we use two synthetic graphs: Kronecker [Leskovec et al., 2005] and Erdős-Rényi [Erdős and Rényi, 1959].

Storage Schemes. We use the following notations to distinguish different storage and indexing methods:

- GBASE RAW (original RAW encoding): raw encoding which is the original adjacency matrix format.
- GBASE NNB (No clustering, No compression, Blocking): raw block encoding without compression and clustering.
- GBASE NZB (No clustering, Zip compression, Blocking): zip block encoding without clustering.
- GBASE CZB (Clustering, Zip compression, Blocking): zip block encoding with clustering.
- GBASE CZB+GS (CZB with Grid Selection): grid selection as described in Section 4.3.

Algorithm 8.1: HADOOP algorithm for Induced Subgraph

```
Input: edge $E = \{(src, dst)\}$ of a graph $G = (V, E)$, and
    query node set $V_q = \{nodeid\}$.
Output: edges belonging to the subgraph induced from $V_q$.
1: InducedSubgraph-Map1(Key $k$, Value $v$):
2: if $(k, v)$ is of type $E$ then
3:     $(src, dst) \leftarrow (k, v)$;
4:     // Emit incidence matrix elements
5:     Output($src$, $[src, dst]$);
6:     Output($dst$, $[src, dst]$);
7: else if $(k, v)$ is of type $V_q$ then
8:     $(nodeid) \leftarrow (k, v)$;
9:     Output($nodeid$,'1');
10: end if
11:
12: InducedSubgraph-Reduce1(Key $k$, Value $v[1..r]$):
13: if $v[]$ contains '1' then
14:     Remove '1' from $v[]$;
15:     for $p \in v[1..r-1]$ do
16:         $[src, dst] \leftarrow p$;
17:         // Emit partial multiplication result
18:         Output($[src, dst]$, 1);
19:     end for
20: end if
21:
22: InducedSubgraph-Map2(Key $k$, Value $v$):
23: Output($k, v$); // Identity Mapper
24:
25: InducedSubgraph-Reduce2(Key $k$, Value $v[1..r]$):
26: $sum \leftarrow 0$;
27: for $num \in v[1..r]$ do
28:     $sum = sum + num$;
29: end for
30: // Select edges whose incident nodes belong to the query node set
31: if $sum$=2 then
32:     $[src, dst] \leftarrow k$;
33:     Output($src, dst$);
34: end if
```

We deploy our GBASE HADOOP implementation onto the M45 HADOOP cluster by Yahoo!. As mentioned earlier, the cluster has total 480 machines with 1.5 Petabytes total storage and 3.5 Terabytes memory.

Graph	Nodes	Edges	File Size	Type	Description
YahooWeb	1.4 B	6.6 B	0.12 TB	real	WWW links in 2002
Kronecker	177 K	2 B	25 GB	synthetic	from Kronecker generator [Leskovec et al., 2005]
	121 K	1.1 B	13.9 GB		
	59 K	282 M	3.3 GB		
	20 K	40 M	439 MB		
Erdős-Rényi	177 K	2 B	25 GB	synthetic	random $G_{n,p}$
	121 K	1.1 B	13.9 GB		
	59 K	282 M	3.3 GB		
	20 K	40 M	439 MB		

Table 8.3: Datasets. B: Billion, K: Thousand, TB: Terabytes, GB: Gigabytes, MB: Megabytes.

8.5.1 Space Efficiency Comparison

We show the data size over three different graphs across different storage schemes in Figure 8.4: 1) KR-2B is a graph of 177K nodes and about 2 billion edges generated using the Kronecker generator; 2) ER-2B is of the same size as KR-2B but generated by Erdős-Rényi generator; 3) YahooWeb is a Web graph of 1.4 billion nodes and 6.6 billion edges. We have the following observations:

Size Reduction. The zip block encoding (NZB) reduces the raw data size significantly (50×, 9×, and 3.6× smaller than the original (RAW) size for Kronecker, Erdős-Rényi, and YahooWeb graphs, respectively). In contrast, the raw block encoding (NNB) decrease the size at most 2.3× smaller than the original (RAW).

Density and Compression. The zip block encoding compression ratio is better for the dense graphs (Kronecker and Erdős-Rényi) than the sparse YahooWeb graph. The reason is that the number of nonzero blocks is much smaller in the dense graphs and thus it results in more storage savings by compression.

Block Structure and Compression. The Kronecker graph has more than 5× better compression ratio than the Erdős-Rényi graph. The reason is that the Kronecker graph is block structured by construction, and thus it benefits the compression algorithm better than its random counterpart.

To summarize, zip block encoding has shown great space savings across all datasets, which confirms the design objective of GBASE.

8.5.2 Indexing Time Comparison

So far, we have compared the resulting space efficiency of different methods. Next, we evaluate the indexing time required by each method. In Figure 8.5, we show the running time of GBASE indexing process vs. the number of edges for graphs generated by both Kronecker (KR) and Erdős-Rényi (ER) generators.

Running Time. To our surprise, zip block encoding (NZB) requires much less time compared to raw block encoding (NNB), despite the additional compression step: NZB performs 50× faster than NNB for 1977M edges. The reason is because the resulting compressed block is much smaller than straightforward

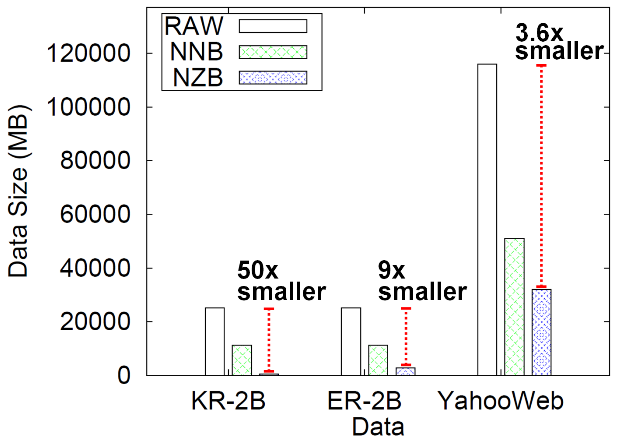


Figure 8.4: Effect of different encoding methods for GBASE. KR-2B: Kronecker graph with 2 billion edges. ER-2B: Erdős-Rényi random graph with 2 billion edges. Notice our proposed zip block encoding (NZB) decreases the input sizes significantly, reducing to 50× smaller than the original (RAW). The Kronecker and the Erdős-Rényi graphs have better performance gain than the YahooWeb graph since the first two are denser than the last and thus take advantage of the compression. The Kronecker graph has better compression than the Erdős-Rényi graph since it has a block-like structure by construction.

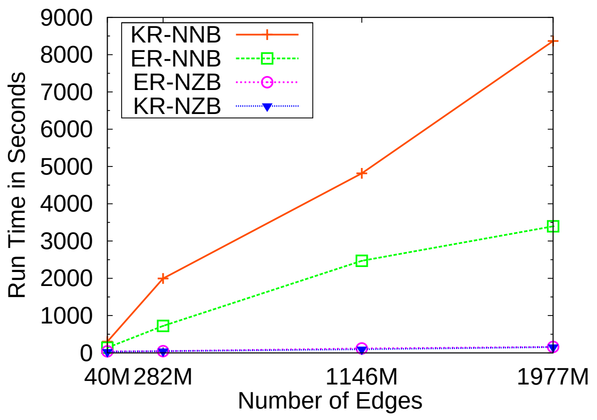


Figure 8.5: Scalability of indexing in GBASE. KR-NNB: Kronecker graph with raw block encoding. ER-NNB: Erdős-Rényi graph with raw block encoding. ER-NZB: Erdős-Rényi graph with zip block encoding. KR-NZB: Kronecker graph with zip block encoding. Notice that the indexing time is linear on the number of edges. Also notice that the zip block encoding (NZB) takes 50× smaller time than the raw block encoding (NNB), since the output size is smaller.

block encoding without compression. Thus, the running time for writing the compressed blocks to disks is much smaller than the uncompressed block.

Linear Scalability. The indexing times for both zip (NZB) and raw block encoding (NNB) increase linearly with the number of edges for both Kronecker and Erdős-Rényi graphs. This confirms the scalability of our encoding schemes.

Thanks to the great storage benefit, the additional compression step of zip block encoding (NZB) is worthwhile. In YahooWeb graph, we observe a similar trend as Kronecker and Erdős-Rényi graphs.

8.5.3 Global Query Time

So far, we confirmed the scalability and efficiency of the indexing phase. Next we evaluate the performance of different schemes on the query phase. Here, we show the scalability of GBASE global queries in Figure 8.6 (a,b). We run the PageRank queries on Kronecker and Erdős-Rényi graphs. All the experiments except NZB are performed on Kronecker graphs. We use the zip blocked Kronecker graphs for the CZB experiment since the Kronecker graphs are block structured from its construction. For NZB experiment, we use the zip blocked Erdős-Rényi graph with the same number of nodes and edges since the Erdős-Rényi graph has nonzeros randomly distributed in the adjacency matrix.

Running Time. We see that CZB, which combines the clustering and the zip block encoding, performs the best. It outperforms RAW, NNB, NZB by 14×, 4.6×, and 2.6×, respectively, for 10 machines. The main reason of the better performance is the decreased I/O time due to reduced storage.

Machine Scalability. All the methods scale up near-linearly with the number of machines as we see in Figure 8.6 (a).

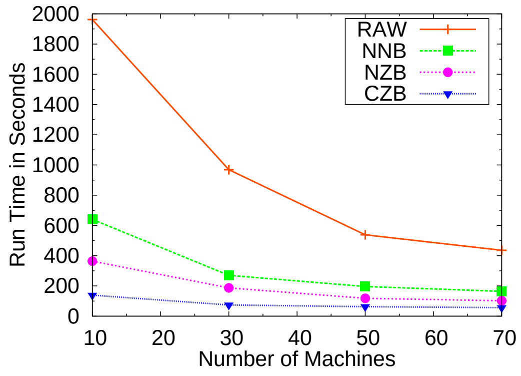
Edge Scalability. The methods also scale up near-linearly with the number of edges as we see in Figure 8.6 (b).

8.5.4 Targeted Query Time

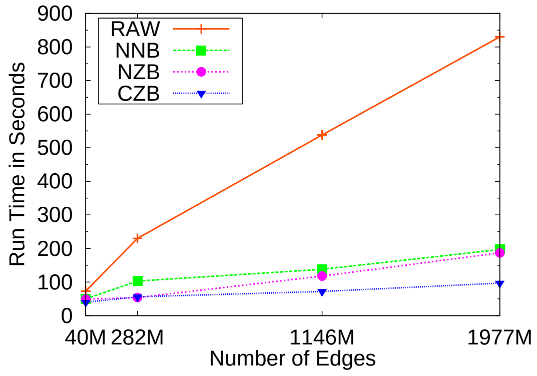
We show the performance on targeted queries in Figure 8.6 (c). Since the targeted queries are often against a small subset of the data, increasing the number of machines does not help speed. Therefore, we only demonstrate the result with fixing the number of machines to 100. All the experiments report the average running time of 5 randomly selected query nodes. The node ids in the YahooWeb graph is encoded in a clustered manner since all the pages in a domain are numbered sequentially. Thus, we use the zip blocked YahooWeb graph for the CZB experiment.

Grid Selection. We see that GBASE CZB+GS, which combines the clustering, the zip block encoding, and the grid selection, works the best for all the targeted queries. Especially, it works the best for 1-neighborhood query outperforming all other competitors from 1.6× to 4×. The reason is that the grid selection method works better if the portion of the relevant grids is small. For 1-neighborhood query, the portion is the smallest ($\sqrt{K}$ for total K grids), while other queries can have many relevant grids depending on the number of neighbors of the query node.

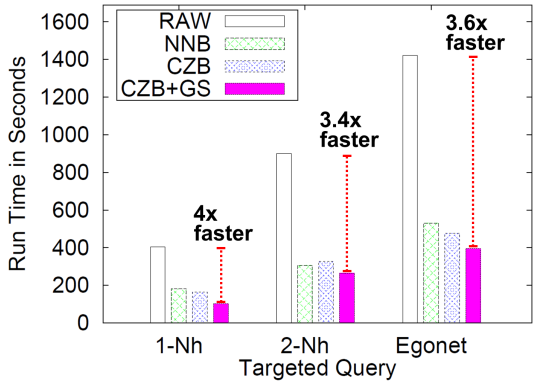
Effect of Zip Block Encoding. The clustered zip block encoding (CZB) performs slightly better than the raw block encoding (NNB) for 1-neighborhood and egonet queries, while it worked slightly worse than NNB for the 2-neighborhood query. The reason is that the size gain of the zip block encoding in



(a) Global query: machine scalability



(b) Global query: edge scalability



(c) Targeted query: running time

Figure 8.6: **(a,b):** Running time and scalability in one iteration of global queries in GBASE on Kronecker and Erdős-Rényi graphs. The CZB method which combines the clustering and the zip block encoding outperforms the RAW method by 14×. Notice also that all the methods scale up near-linearly on the number of machines and edges. **(c):** Running time of targeted queries over different storage and indexing methods, on YahooWeb graph. K-Nh denotes K step neighborhood query. Note that the CZB+GS (grid selection method combined with the clustered zip block encoding) outperforms the others by 4× at maximum.

CZB is not big enough to overshadow the increased running time for the zip compression. However, the performance of zip block encoding will continuously increase as better clustering algorithm is developed, as shown in the well clustered graph results of Figure 8.6 (a,b). Moreover, the zip block encoding enjoys additional benefits of less storage and indexing time.

8.6 Related Work

In this section, we review the related work, which can be categorized into three parts: (1) graph indexing techniques, (2) graph queries, and (3) column store.

Graph Indexing. Graph indexing is very active in both databases community as well as data mining community in the recent years. To name a few, Trißl et al [Trißl and Leser, 2007] proposed to index the graph using pre- and postorder number to answer the reachability queries. Chierichetti et al [Chierichetti et al., 2009] explored link reciprocity for adjacency queries. Aggarwal et al [Aggarwal et al., 2009] proposed edge sampling to handle graph connectivity queries. Sarkar et al [Sarkar and Moore, 2010] explored the clustering properties to proximity queries on graphs. Maserra et al [Maserrat and Pei, 2010] proposed a Eulerian data structure for neighborhood queries.

Despite of their success, there are two major limitations of these works. First, all the indexing techniques are designed for *one* particular type of queries. Therefore, their performance might be highly optimized for that particular type of query, but they are far sub-optimal for the remaining, vast majority types of queries. Second, they are implicitly designed for the centralized computational mode, which limits the size of the graph such indexing techniques can support. These limitations are carefully addressed in the GBASE, which supports multiple different types of queries simultaneously and is naturally applicable to the distributed computing environment.

Finally, there are works on indexing *many small* graphs using frequent subgraph [Xin et al., 2005, Zhao et al., 2007a] or significant graph patterns [Yan et al., 2008], which is quite different from our setting where we have *one large* graph.

Graph Queries. There are numerous different queries on graphs. To name a few, graph-level queries answer some global statistics of the whole graph, e.g., estimating diameters (Chapter 3), counting connected components (Chapter 4), etc. Node-level queries, on the other hand, focus on the relationship among individual nodes. Representative queries include neighborhood [Maserrat and Pei, 2010], proximity [Tong et al., 2006], PageRank [Page et al., 1998], centrality [Bader et al., 2007], etc. Between the graph-level and individual node-level, there are also queries on the sub-graph level, e.g., community detection [Karypis and Kumar, 1999b, Andritsos et al., 2004], finding induced subgraph [Addario-Berry et al., 2010], etc. GBASE covers a wide range of queries, including the global and the node-level ones, by a unified matrix-vector multiplication framework.

Column Store. Column-oriented DBMS has gained its popularity in the recent years, due to (among other merits) its excellent I/O efficiency for read-extensive analytical workloads. From research community, some representative works include [Stonebraker et al., 2005, Abadi et al., 2008, 2009, Ivanova et al., 2009, Héman et al., 2010]. A notable work of column store database from industrial side is HBase (http://hbase.apache.org/). HBase is designed for large sparse data, built on the top of HADOOP core. Different from HBase, our GBASE partitions the data in two dimensions (both columns and rows) and it is tailored for large real graphs. By leveraging the block and community-like property which exists in many real graphs, GBASE enjoys the advantages of both row-oriented and column-oriented storages.

8.7 Conclusion

In this chapter, we propose GBASE, a scalable and general graph management system. The main contributions are the following:

1. **Storage**. We carefully design GBASE to efficiently store homogeneous regions of graphs in distributed settings using a novel 'block compression'. Experiments on billion-scale graphs show that the storage and the running time are reduced up to $50\times$ of the original.
2. **Algorithms**. We unify node-based and edge-based queries using matrix-vector multiplications on the adjacency and the incidence matrices. As a result, we get *seven* different types of versatile graph queries supporting various applications.
3. **Query Optimization**. We propose a fast graph query execution algorithm using a grid selection. Also, we provide an efficient MAPREDUCE algorithm to support incidence matrix based queries using the original adjacency matrix, without explicitly building the incidence matrix.

Chapter 9

Edge Layout

Given a real world graph, how should we lay-out its edges in a file of (source, destination) pairs? How can we compress it? These questions are closely related, and the typical approach so far is to find clique-like communities, like the 'cavemen graph', and compress them. We show that the block-diagonal mental image of the 'cavemen graph' is the wrong paradigm, in full agreement with earlier results that real world graphs have no good cuts. Instead, we propose to envision graphs as a collection of hubs connecting spokes, with super-hubs connecting the hubs, and so on, recursively.

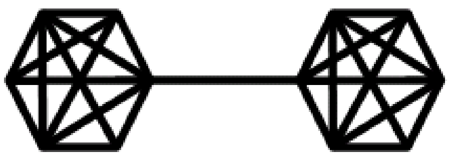
Based on the idea, we propose the SLASHBURN method ("burn" the hubs, and "slash" the remaining graph into smaller connected components). Our view point has several advantages: (a) it avoids the 'no good cuts' problem, (b) it gives better compression, and (c) it leads to faster execution times for matrix-vector operations, which are the back-bone of most graph processing tools.

Experimental results show that our SLASHBURN method consistently outperforms other methods on all datasets, giving good compression and faster running time.

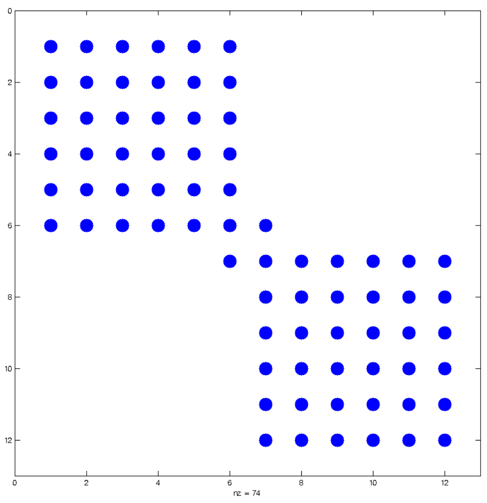
9.1 Introduction

How can we compress graphs efficiently? How should we try to find communities in graphs? The two questions are closely related: if we find good communities, then we can compress the graph well since the nodes in the same community have redundancies (e.g. similar neighborhood) which help us shrink the size of the data (and thus, also shrink the I/O and communication costs for graph processing). Similarly, good compression implies good communities. The traditional research focus was on finding homogeneous regions in the graph so that nodes inside a region are tightly connected to each other than to nodes in other regions. In other words, the focus was to search for 'caveman communities' where a person in a cave knows others in the same cave very well, while knows very little about persons in different caves as shown in Figure 9.1 (a). In terms of the adjacency matrix, the goal was to find an ordering of nodes so that the adjacency matrix is close to block-diagonal, containing more 'square' blocks as in Figure 9.1 (b). Spectral clustering [Shi and Malik, 1997, Ng et al., 2002], co-clustering [Dhillon et al., 2003], cross-associations [Chakrabarti et al., 2004], and shingle-ordering [Chierichetti et al., 2009] are typical examples for such approaches.

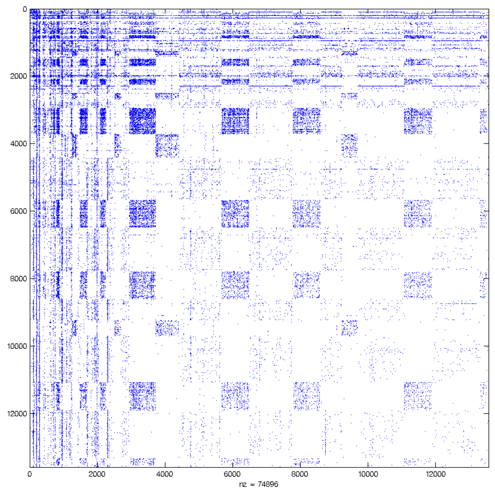
However, real world graphs are much more complicated and inter-connected than caveman graphs. It



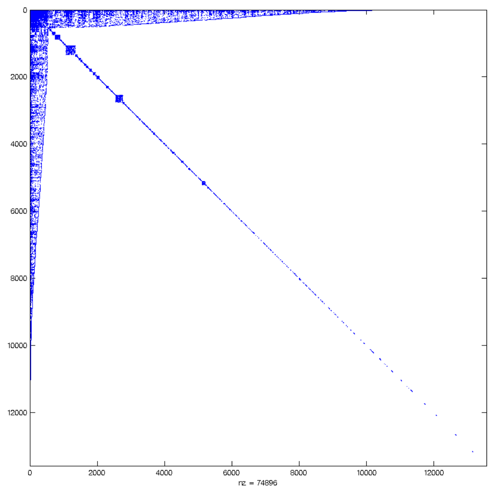
(a) Caveman graph C



(b) Adjacency Matrix of C



(c) Adjacency Matrix of AS-Oregon graph



(d) AS-Oregon after SLASHBURN

Figure 9.1: Caveman graph, real-world graph, and the result from our proposed SLASHBURN ordering. Real world graphs are much more complicated and inter-connected than caveman graph, with few 'hub' nodes having high degrees and majority of nodes having low degrees. Finding a good 'cut' on real world graphs to extract homogeneous regions (like the square diagonal blocks in the caveman adjacency matrix (b)) is difficult due to the hub nodes. Instead, our proposed SLASHBURN finds novel 'skinny' communities which lead to good compression: in (d), the edges are concentrated to the left, top, and diagonal areas while making empty spaces in most of the areas.

Symbol	Definition
A	adjacency matrix of a graph
n	number of nodes in a graph
k	number of hub nodes to slash per iteration in SLASHBURN
$w(G)$	wing width ratio of a graph G, meaning the ratio of the number of total hub nodes to n
b	block width used for block based matrix-vector multiplication

Table 9.1: Table of symbols.

is well known that most real world graphs follow power-law degree distributions with few 'hub' nodes having very high degrees and majority of the nodes having low degrees [Faloutsos et al., 1999]. These hub nodes break the assumption of caveman-like communities since the hubs are well connected to most of the nodes in graphs, effectively combining all the caves into a huge cave. Thus, it is not surprising that well defined communities in real world networks are hard to find [Leskovec et al., 2008].

In this chapter, we propose a novel approach to finding communities and compressions in graphs. Our approach, called SLASHBURN, is to exploit the hubs and the neighbors ('spokes') of the hubs to define an alternative community different from the traditional community. SLASHBURN is based on the observation that real world graphs are easily disconnected by hubs, or high degree nodes: removing hubs from a graph creates many small disconnected components, and the remaining giant connected component is substantially smaller than the original graph. The communities defined using hubs and spokes correspond to skinny blocks in an adjacency matrix as shown in Figure 9.1 (d), in contrast to the square blocks in caveman communities as shown in Figure 9.1 (b). We show that these hubs and spokes can be carefully ordered to get a compact representation of the adjacency matrix, which in turn leads to good compression. Our contributions are the following:

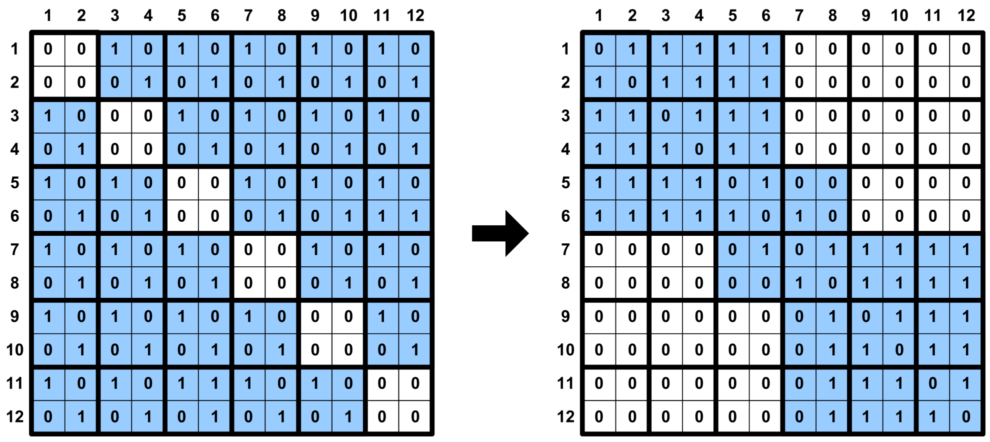
1. **Paradigm shift.** Instead of looking for near-cliques ('caves'), we look for hubs and spokes for a good graph compression. Our approach is much more suitable for real world, power-law graphs like social networks.
2. **Compression.** We show that our method gives good compression results when applied on real world graphs, consistently outperforming other methods on all datasets.
3. **Speed.** Our method boosts the performance of matrix-vector multiplication of graph adjacency matrices, which is the building block for various algorithms like PageRank, connected components, etc.

The rest of the chapter is organized as follows. Section 9.2 precisely describes the problem and our proposed method for laying out edges for better compressing graphs. We give experimental results in Section 9.3, showing the compression and running time enhancements. After discussing related works on Section 9.4, we conclude in Section 9.5.

To enhance the readability, we listed the symbols frequently used in Table 9.1.

9.2 Proposed Method

In this section, we give a formal definition of the problem, describe our proposed method, and analyze its complexity.



$$\begin{array}{c|cccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
2 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
3 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
4 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\
5 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
6 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
7 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
8 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\
9 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
10 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
11 & 1 & 0 & 1 & 0 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\
12 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0
\end{array}
\quad \Rightarrow \quad
\begin{array}{c|cccccccccccc}
 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
1 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 1 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
3 & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
4 & 1 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
6 & 1 & 1 & 1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
7 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
8 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 \\
9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
10 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
11 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
12 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0
\end{array}$$

Figure 9.2: Importance of ordering. Left: adjacency matrix of Figure 9.1 (a) with a random ordering of nodes. Right: adjacency matrix of the same graph, but with a compression-friendly ordering, where nodes 1 to 6 are assigned to the left clique, and nodes 7 to 12 are assigned to the right clique. If we use 2 by 2 blocks to cover all the nonzero elements inside the matrix, the right matrix requires smaller number of denser blocks which lead to better compression.

9.2.1 Problem Definition

Given a large graph, we want to layout its edges so that the graph can be compressed well, and graph mining queries can be answered quickly. Specifically, we consider the application of large scale matrix-vector multiplication which is the building block of many graph mining algorithms including PageRank, diameter estimation, and connected components, as described in Chapter 4. The state-of-the art method for the large scale matrix-vector multiplication is the block multiplication method (Section 4.3.2), where the original matrix is divided into b by b square matrix blocks, the original vector is divided into length b vector blocks, and the matrix-vector blocks are multiplied.

For example, see Figure 4.1 for the block multiplication method where a 6 by 6 matrix is multiplied with a length 6 vector using 2 by 2 matrix blocks and length 2 vector blocks. We assume that each block is stored independently from each other, without requiring neighbor or reciprocal blocks to decode its edges, since such independency among blocks allows more scalable processing in large scale, distributed platforms like MAPREDUCE [Dean and Ghemawat, 2004].

In this scenario, it is desired that the adjacency matrix has clustered edges: *smaller* number of *denser* blocks is better than *larger* number of *sparser* blocks. There are two reasons for this. First, smaller number of denser blocks reduces the number of disk accesses. Second, it provides better opportunity for compression. For example, see Figure 9.2. The left matrix is the adjacency matrix of Figure 9.1 (a) with a random ordering of nodes, while the right matrix is the adjacency matrix of the same graph with a compression-friendly ordering where nodes 1 to 6 are assigned to the left clique, and nodes 7 to 12 are assigned to the right clique. Assume we use 2 by 2 blocks to cover all the nonzero elements inside the matrix. Then the right matrix requires smaller number of blocks than the left matrix. Furthermore, each block in the right matrix is denser than the one in the left matrix, which could lead to better compression of graphs.

Formally, our main problem is as follows.
Problem 2. *Given a graph with the adjacency matrix A, find a permutation* $\pi : V \to [n]$ *such that the*

storage cost function $cost(A)$ is minimized.

The notation $[n]$ means the ordering of n nodes. Following the motivation that *smaller* number of *denser* blocks is better for compression than *larger* number of *sparser* blocks, the first cost function we consider is the number of nonempty, b by b square blocks in the adjacency matrix:

$$cost_{nz}(A, b) = \texttt{number of nonempty blocks}, \tag{9.1}$$

where b is the block width. The second, and more precise cost function uses the required number of bits to encode the adjacency matrix using a block-wise encoding (divide the matrix into blocks, and encode each block using standard compression algorithms like gzip). The required bits are decomposed into two parts: one for the nonzero elements inside blocks, the other for storing the meta information about the blocks.

- *Nonzeros inside blocks.* Bits to compress nonzero elements inside blocks.
- *Meta information on blocks.* Bits to store the row and column ids of blocks.

Using the decomposition, we define a cost function $cost_{it}(A, b)$ assuming a compression method achieving the information theoretic lower bound [Rissanen and Langdon Jr., 1979, Chakrabarti et al., 2004]:

$$cost_{it}(A, b) = |T| \cdot 2 \log \frac{n}{b} + \sum_{\tau \in T} b^2 \cdot H(\frac{z(\tau)}{b^2}), \tag{9.2}$$

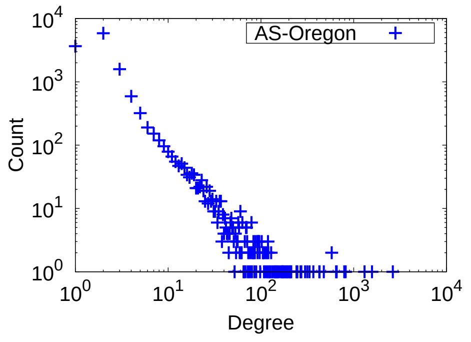
where n is the number of nodes, T is the set of nonempty blocks of size b by b, $z(\tau)$ is the number of nonzero elements within a block τ, and $H(p) = p \log \frac{1}{p} + (1-p) \log \frac{1}{1-p}$ is the binary Shannon entropy function. The first term $|T| \cdot 2 \log \frac{n}{b}$ in Equation (9.2) represents the bits to encode the meta information on blocks. Since each block requires two $\log \frac{n}{b}$ bits to encode the block row and the block column ids, the total required bits are $|T| \cdot 2 \log \frac{n}{b}$. The second term in Equation (9.2) is the bits to store nonzeros inside blocks: we use information theoretic lower bound for encoding the bits, since it gives the minimum number of bits achievable by any coding methods. Note b^2 is the maximum possible edge counts in a b by b block, and $\frac{z(\tau)}{b^2}$ is the density of the block.

The two cost functions defined in Equation (9.1) and (9.2) will be evaluated and compared on different ordering methods in Section 9.3.

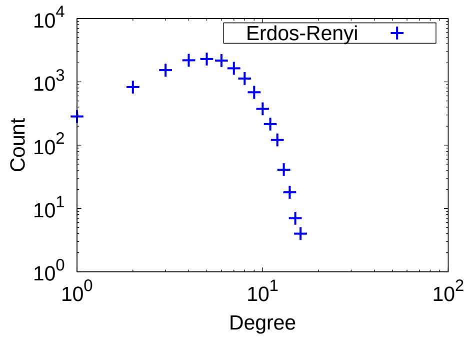
9.2.2 Why Not Classic Partitioning?

In general, directly minimizing the cost functions is a difficult combinatorial problem which could require $n!$ trials in the worst case. Traditional approach is to use graph partitioning algorithms to find good 'cuts' and homogeneous regions so that nodes inside a region form a dense community, and thereby leading to better compressions. Examples include spectral clustering [Shi and Malik, 1997, Ng et al., 2002], co-clustering [Dhillon et al., 2003], cross-associations [Chakrabarti et al., 2004], and shingle-ordering [Chierichetti et al., 2009]. However, such approaches do not work well for real world, power law graphs since there exists no good cuts in such graphs [Leskovec et al., 2008], which we also experimentally show in Section 9.3.

The reason of the 'no good cut' in most real world graphs is explained by their power-law degree distributions and the existence of 'hub' nodes. Such hub nodes combine the communities to blend into each other,



(a) AS-Oregon



(b) Erdős-Rényi

Figure 9.3: Degree distributions of power-law vs. random graphs. The left is from a real-world graph showing a power law degree distribution with few hub nodes having very high degrees, and majority of nodes having low degrees. The right is from a random (Erdős-Rényi) graph which has the same number of nodes and edges as the graph for the left plot. The random graph has an exponential tail in its degree distribution without distinct hubs.

making the cut-based algorithms fail. Rather than resorting to the cut-based algorithms that are not designed to work on power-law graphs, we take a novel approach to finding communities and compressions, which we explain next.

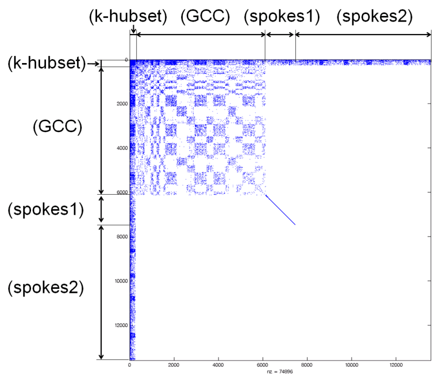
9.2.3 Graph Shattering

As described in the previous section, finding homogeneous regions in real world graphs is infeasible due to hub nodes. Our main idea to solve the problem is to exploit the hubs to define an alternative community different from the traditional community. Remember that most real-world graphs have a power law in its degree distribution: there exist few hub nodes with very high degrees, while majority of the nodes having low degrees, as shown in Figure 9.3 (a). In contrast, random graphs have degree distributions whose tails drop exponentially: this means there are no hubs with extremely high degrees, as shown in Figure 9.3 (b).

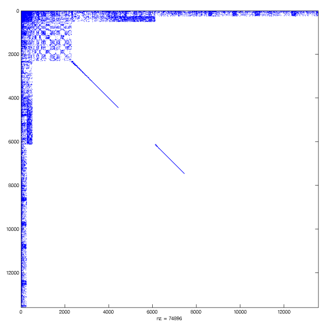
We start with an observation that real-world graphs are easily shattered by removing hub nodes from them: while majority of the nodes still belong to the giant connected component, a nontrivial portion of the nodes belong to small disconnected components by the removal. The nodes belonging to the small disconnected components after the removal of the hub nodes can be regarded as satellite nodes connected to the hub nodes. In other words, those satellite nodes have links only to the hub nodes, and completely disconnected from the rest of the nodes in the graph. This is the exact property we are utilizing.

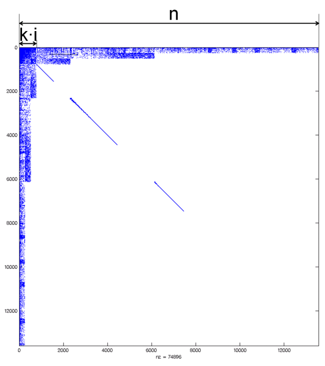
To precisely describe our method, we define related terms.
Definition 9 (k-hubset)**.** *The k-hubset of a graph G is the set of nodes with top k highest centrality scores.*

We use the degree of a node as the centrality score in this chapter, but any centrality (e.g., closeness, betweenness [Borgatti and Everett, 2006], PageRank, "eigendrop" [Prakash et al., 2010b], etc.) can be used for the score. Removing k-hubset from a graph leads to the definition of k-shattering.
Definition 10 (k-shattering)**.** *The k-shattering of a graph G is the process of removing the nodes in k-hubset, as well as edges incident to k-hubset, from G.*



(a) AS-Oregon after 1 iteration


(b) .. after 1 more iteration



(c) .. after 1 more iteration

Figure 9.4: SLASHBURN in action: adjacency matrices of AS-Oregon graph after applying SLASHBURN ordering. After 1 iteration, the nodes are decomposed into k-hubset, GCC, and the spokes. The spokes are only connected to k-hubset, while completely disconnected to the GCC, which makes large empty spaces in the bottom-right area of the adjacency matrix. The same process applies to the remaining GCC recursively. Notice that the nonzero elements in the matrix are concentrated to the left, top, and diagonal areas of the matrix, making an arrow-like shape. Compared to the original adjacency matrix in Figure 9.1 (c), the final matrix has much larger empty spaces, enabling better compression.

Let us consider the following shattering process. Given a graph G, we perform a k-shattering on G. Among the remaining connected components, choose the giant connected component (GCC). Perform a k-shattering on the GCC, and do the whole process recursively. Eventually, we stop at a stage where the size of the GCC is less than or equal to k. A natural question is, how quickly is a graph shattered? To measure the speed of the shattering process, we define the *wing width ratio* $w(G)$ of a graph G.

Definition 11. *The wing width ratio $w(G)$ of a graph G is $\frac{k \cdot i}{n}$ where k is the number used for the k-shattering, i is the number of iterations until the shattering finishes, and n is the number of nodes in G.*

Intuitively, the wing width ratio $w(G)$ corresponds to the width of the blue wing of the typical spyplot (visualization of the adjacency matrix; see Figure 9.4 (c)); notice that for all real world graphs, the corresponding spyplots look like ultra-modern airplanes, with the blue lines being their wings. $w(G)$ is the ratio of 'wing' width to the number of nodes in the graph. A low $w(G)$ implies that the graph G is shattered quickly, while a high $w(G)$ implies that it takes long to shatter G. As we will see in Section 9.3.3, real-world, power-law graphs have low $w(G)$. Our proposed SLASHBURN method utilizes the low wing width ratio in real world graphs.

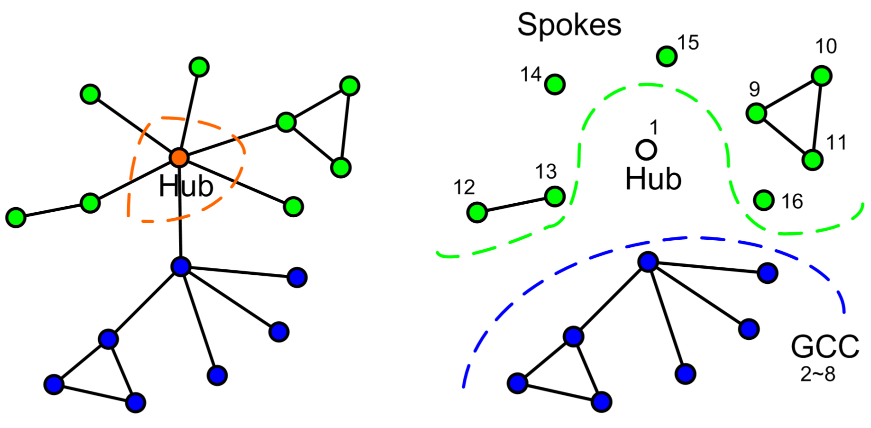
9.2.4 Slash-and-Burn

In this section, we describe SLASHBURN, our proposed ordering method for compressing graphs. Given a graph G, the SLASHBURN method defines a permutation $\pi : V \rightarrow [n]$ of a graph so that nonzero elements in the adjacency matrix of G are grouped together. Algorithm 9.1 shows the high-level idea of SLASHBURN.

The lines 1 and 2 of Algorithm 9.1 removes ('slash-and-burn') top k highest centrality scoring nodes, thereby decomposing nodes in G into the following three groups:

Algorithm 9.1: SLASHBURN

```
Input: edge set $E$ of a graph $G = (V, E)$, and
    a constant $k$ (default = 1).
Output: array $\Gamma$ containing the ordering $V \rightarrow [n]$.
1: Remove $k$-hubset from $G$ to make the new graph $G'$. Add the removed $k$-hubset to the front of $\Gamma$.
2: Find connected components in $G'$. Add nodes in non-giant connected components to the back of
   $\Gamma$, in the decreasing order of sizes of connected components they belong to.
3: Set $G$ to be the giant connected component (GCC) of $G'$. Go to step 1 and continue, until the
   number of nodes in the GCC is smaller than $k$.
```



(a) Before SLASHBURN (b) After SLASHBURN

Figure 9.5: [Best Viewed In Color] A graph before and after 1 iteration of SLASHBURN. Removing a hub node creates many smaller 'spokes', and the GCC. The hub node gets the lowest id (1), the nodes in the spokes get the highest ids (9∼16)) in the decreasing order of the connected component size they belong to, and the GCC takes the remaining ids (2∼8). The next iteration starts on the GCC.

- k-hubset: top k highest centrality scoring nodes in G.
- GCC: nodes belonging to the giant connected component of G'. Colored blue in Figure 9.5.
- Spokes to the k-hubset: nodes belonging to the non-giant connected component of G'. Colored green in Figure 9.5.

Figure 9.5 shows a graph before and after 1 iteration of SLASHBURN. After removing the 'hub' node at the center, the graph is decomposed into the GCC and the remaining 'spokes' which we define to be the non-giant connected components connected to the hubs. The hub node gets the lowest id (1), the nodes in the spokes get the highest ids (9∼16)) in the decreasing order of the connected component size they belong to, and the GCC takes the remaining ids (2∼8). The same process applies to the nodes in GCC, recursively.

Figure 9.4 (a) shows the AS-Oregon graph after the lines 1 and 2 of Algorithm 9.1 are executed for the first time with $k = 256$. In the figure, we see that a k-hubset comes first with GCC and spokes following after them. The difference between (spokes1) and (spokes2) is that the nodes in (spokes2) are connected only to some of the nodes in k-hubset, thereby making large empty spaces in the adjacency matrix. Notice also that nodes in (spokes1) make a thin diagonal line, corresponding to the edges among themselves.

A remarkable result is that the remaining GCC takes only 45% of the nodes in the original graph, after removing 256 (=1.8 %) high degree nodes. Figure 9.4 (b) and (c) shows the adjacency matrix after doing the same operation on the remaining GCC, recursively. Observe that nonzero elements in the final adjacency matrix are concentrated on the left, top, and diagonal areas of the adjacency matrix, creating an arrow-like shape. Observe also that the final matrix has huge empty spaces which could be utilized for better compression, since the empty spaces need not be stored.

An advantage of our SLASHBURN method is that it works on any power-law graphs without requiring any domain-specific knowledge or a well defined natural ordering on the graph for better permutation. Finally, we note that setting k to 1 often gives the best compression by making the wing width ratio $w(G)$ minimum or close to minimum. However, setting k to 1 requires many iterations and longer running time. We found that setting k to 0.5% of the number of nodes gives good compression results with small number of iterations on most real world graphs.

9.2.5 Analysis

We analyze the time and the space complexities of the SLASHBURN algorithm.

Lemma 9.1 (Time Complexity of SLASHBURN). SLASHBURN *takes* $O(|E| + |V| \log |V|)i$ *time where* $i = \frac{|V| \cdot w(G)}{k}$ *is the number of iterations.*

Proof. In Algorithm 9.1, step 1 takes $O(|V| + |E|)$ time to compute the degree of nodes, and to remove k-hubset. Step 2 requires $O(|E| + |V| \log |V|)$ time since connected components require $O(|V| + |E|)$ time, and sorting takes $|V| \log |V|$ time. Thus, 1 iteration of SLASHBURN takes $O(|E| + |V| \log |V|)$ time, and the lemma is proved by multiplying the number i of iterations to it. □

Lemma 9.1 implies that smaller wing width ratio $w(G)$ will result in faster running time. We note that real world, power-law graphs have small wing width ratio, which we show experimentally in Section 9.3.3.

For space complexity, we have the following result.

Lemma 9.2 (Space Complexity of SLASHBURN). SLASHBURN *requires* $O(|V|)$ *space.*

Proof. In step 1, computing the degree requires $O(|V|)$ space. In step 2, connected component requires $O(|V|)$ space, and sorting requires at most $O(|V|)$ space. The lemma is proved by combining the space requirements for the two steps. □

9.3 Experiments

In this section, we present experimental results to answer the following questions:

Q1 How well does SLASHBURN compress graphs compared to other methods?
Q2 How does SLASHBURN decrease the running time of large scale matrix-vector multiplication?
Q3 How quickly can we shatter real world graphs? What are the wing width ratio of real world, power-law graphs?

We compare SLASHBURN with the following six methods.

Graph	Nodes	Edges	File Size	Type	Description
LiveJournal	4.8 M	69 M	1.1 GB	real	friendship social network
WWW-Barabasi	325 K	1.5 M	20 MB	real	WWW links in nd.edu
Flickr	404 K	2.1 M	28 MB	real	person-person
Enron	80 K	313 K	11 MB	real	Enron email
Epinions	75 K	508 K	5 MB	real	who trusts whom
AS-Oregon	14 K	75 K	385 KB	real	router connetions

Table 9.2: Datasets. M: Million, K: Thousand, GB: Gigabytes, MB: Megabytes, KB: Kilobytes.

- **Random.** Random ordering of the nodes.
- **Natural.** Natural ordering of the nodes, that is, the original adjacency matrix. For some graphs, the natural ordering provides high locality among consecutive nodes (e.g. lexicographic ordering in Web graphs [Boldi and Vigna, 2004]).
- **Degree Sort (DegSort).** Ordering based on the decreasing degree of the nodes.
- **Cross Association (CA).** Cross-association [Chakrabarti et al., 2004] based ordering so that nodes in a same group are numbered consecutively.
- **Spectral Clustering.** Normalized spectral clustering [Shi and Malik, 1997], also known as the normalized cut. Order nodes by the second smallest eigenvector score of a generalized eigenvector problem.
- **Shingle.** Shingle ordering is the most recent method for compressing social networks [Chierichetti et al., 2009]. It groups nodes with similar fingerprints (min-wise hashes) obtained from the out-neighbors of nodes.

The graphs used in our experiments along with their descriptions are summarized in Table 9.2 whose entries are repeated from Table 2.2 for convenience.

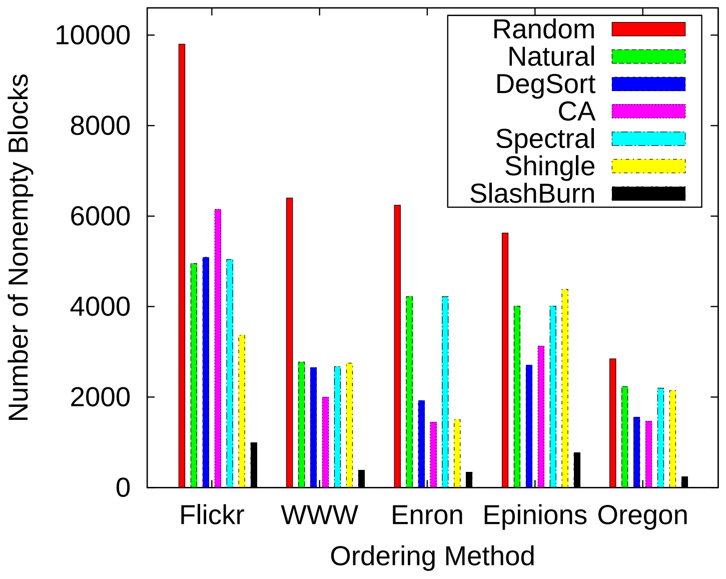
9.3.1 Compression

We compare the ordering methods based on the cost of compression using the two cost functions defined in Equation (9.1) and (9.2) of Section 9.2:

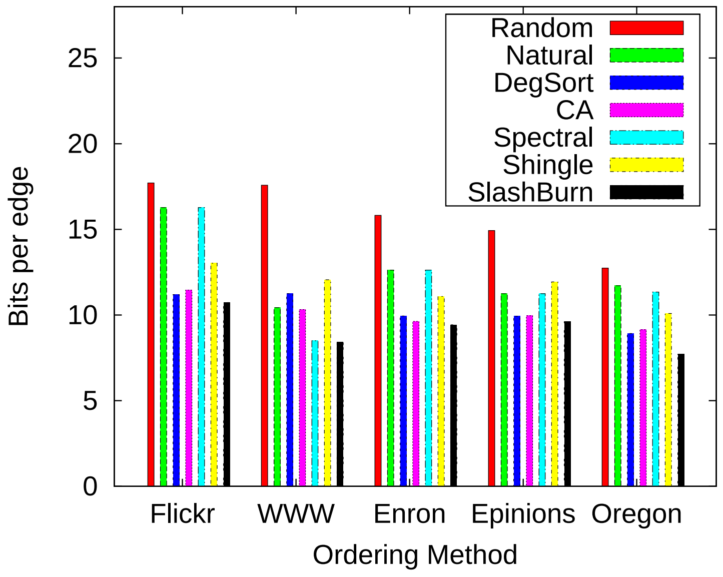
- $cost_{nz}(A, b)$: number of nonempty blocks.
- $cost_{it}(A, b)$: required bits using information-theoretic coding methods.

Figure 9.6 shows the costs of ordering methods. Figure 9.6 (a) shows the number of nonempty blocks ($cost_{nz}(A)$), and Figure 9.6 (b) shows the bits per edge computed using $cost_{it}(A, b)$. The exact numbers are listed in Table 9.3 and 9.4, respectively. Notice that for all the cost functions, SLASHBURN performs the best. For the number of nonempty blocks, SLASHBURN reduces the counts by up to 20$\times$ compared to the random ordering, and by up to 6.1$\times$ compared to the second best orderings. For the bits per edge, SLASHBURN reduces the bits by up to 2.1$\times$ compared to the random ordering, and by up to 1.2$\times$ compared to the second best orderings.

The amount of compression can be checked visually. Figure 9.7 show the spyplots, which are nonzero patterns in the adjacency matrices, of real world graphs permuted from different ordering methods. Random ordering makes the spyplot almost filled; natural ordering provides more empty space than random ordering, meaning that the natural ordering exploits some form of localities. Degree sort makes the upper-



(a) $cost_{nz}(A, b)$: number of nonempty blocks



(b) $cost_{it}(A, b)$: information theoretic cost

Figure 9.6: Compression comparison of ordering methods. DegSort: degree sort, CA: cross association, and Spectral: spectral clustering. For all the cost functions, SLASHBURN performs the best. **(a):** SLASHBURN reduces the number of nonempty blocks by up to 20× compared to the random ordering, and by up to 6.1× compared to the second best orderings. **(b):** SLASHBURN reduces the bits per edge by up to 2.1× compared to the random ordering, and by up to 1.2× compared to the second best orderings.

Graph	Random	Natural	Degree Sort	CA	SC	Shingle	SB
LiveJournal (bw=4096)	1401856	1060774	885153	*	*	960642	**873469**
Flickr (bw=4096)	9801	4950	5091	6149	5042	3366	**994**
WWW-Barabasi (bw=4096)	6400	2774	2647	1997	2671	2751	**384**
Enron (bw=1024)	6241	4220	1922	1442	4220	1498	**339**
Epinions (bw=1024)	5624	4010	2703	3124	4010	4381	**768**
AS-Oregon (bw=256)	2845	2232	1552	1463	2197	2142	**239**

Table 9.3: Number of nonempty blocks for the competing ordering methods. CA: Cross Association, SC: Spectral Clustering, SB: SLASHBURN. 'bw' denotes the block width, and the winners are in bold fonts. For the LiveJournal data, cross association and spectral clustering (marked *) could not be performed since the algorithms are too heavy to run on such a large graph. Notice that SLASHBURN, formatted in bold fonts, outperforms all others. The results were similar for other block widths.

Graph	Random	Natural	Degree Sort	Cross Association	Spectral Clustering	Shingle	SB
LiveJournal	19.89	16.82	16.87	*	*	18.52	**16.67**
Flickr	17.71	16.27	11.19	11.45	16.27	13.02	**10.73**
WWW-Barabasi	17.58	10.43	11.25	10.32	8.5	12.06	**8.41**
Enron	15.82	12.62	9.94	9.63	12.62	11.08	**9.43**
Epinions	14.93	11.24	9.93	9.96	11.24	11.93	**9.61**
AS-Oregon	12.74	11.71	8.92	9.14	11.34	10.09	**7.71**

Table 9.4: Bits per edge for the competing ordering methods, according to the information theoretic lower bound. SB: SLASHBURN. For the LiveJournal data, cross association and spectral clustering (marked *) could not be performed since the algorithms are too heavy to run on such a large graph. Note that the result from SLASHBURN, formatted in bold fonts, outperforms all others.

left area of the adjacency matrix more dense. Cross association makes many rectangular regions that are homogeneous. Spectral clustering tries to find good cuts, but obviously cannot find such cuts on the real world graphs. In fact, for all the graphs except AS-Oregon in Figure 9.7, the spyplot after the spectral clustering looks very similar to that of the natural ordering. Shingle ordering makes empty spaces on the top portion of the adjacency matrix of some graphs: the rows of such empty spaces correspond to nodes without outgoing neighbors, However, the remaining bottom portion is not concentrated well. Our SLASHBURN method collects nonzero elements to the left, top, and the diagonal lines of the adjacency matrix, thereby making an arrow-like shape. Notice that SLASHBURN requires the smallest number of square blocks to cover the edges, leading to the best compression as shown in Table 9.4.

9.3.2 Running Time

We show the performance implication of SLASHBURN for large scale graph mining on distributed platform, using HADOOP, an open source MAPREDUCE framework. We test the performance of block-based PageRank using HADOOP on graphs created from different ordering methods. For storing blocks, we used the standard gzip algorithm to compress the 0-1 bit sequences. Figure 9.8 shows file size vs. running time on different ordering methods on LiveJournal graph. The running time is measured for one iteration of PageRank on HADOOP. Notice that SLASHBURN results in the smallest file size, as well as the smallest running time. We note that LiveJournal is one of the dataset that is very hard to compress. In fact, a similar dataset was analyzed in the paper that proposed the shingle ordering [Chierichetti et al., 2009]: however, their proposed 'compression' method *increased* the bits per edge, compared to the original graph. Our SLASHBURN outperforms all other methods, including the shingle and the natural ordering, even on this 'hard to compress' dataset.

9.3.3 Real World Graphs Shatter Quickly

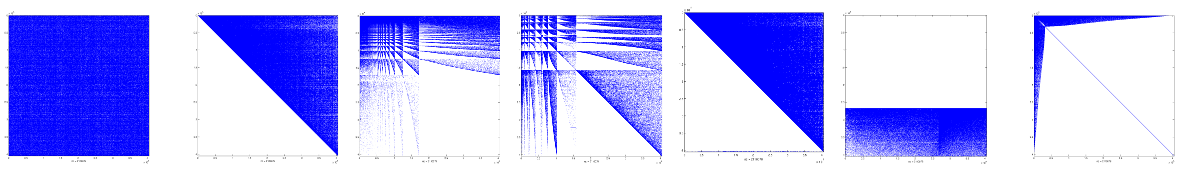
How quickly can a *real world* graph be shattered into tiny components? What are the differences of the wing width ratio between real world, power-law graphs and random [Erdős and Rényi, 1959] graphs? Table 9.5 shows the wing width ratio $w(G)$ of real world and random graphs. We see that real world graphs have coefficients between 0.037 and 0.099 which are relatively small. For WWW-Barabasi graph, it means that removing 3.7 % of high degree nodes can shatter the graph.

In contrast, random (Erdős-Rényi) graphs have higher wing width ratio $w(G)$. We generated two random graphs, 'ER-Epinions', and 'ER-AS-Oregon', which have the same number of nodes and edges as 'Epinions', and 'AS-Oregon', respectively. The wing width ratios of the two random graphs are 0.611 and 0.358, respectively, which are at least $6.2\times$ larger than their real world counterparts.

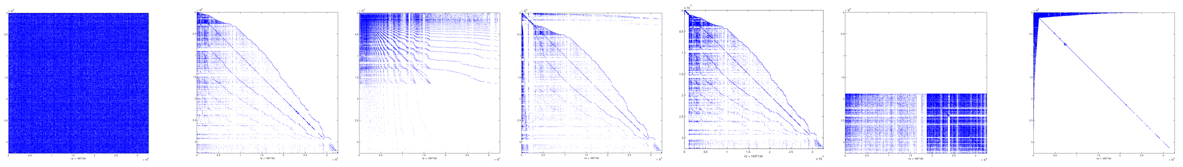
9.4 Related Work

The related works form two groups: structure of networks, and graph partition/compression.

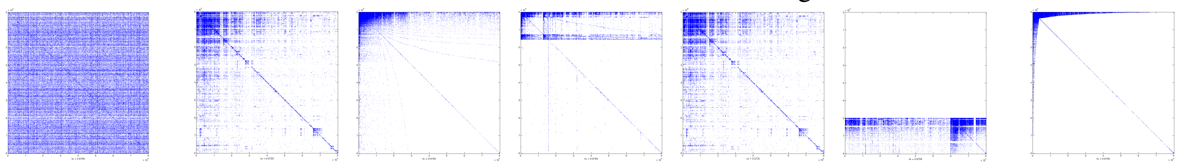
Structure of Networks. Research on the structure of complex networks has been receiving significant amount of attention. Most real world graphs have power law in its degree distribution [Faloutsos et al., 1999], a property that distinguishes them from random graphs [Erdős and Rényi, 1959] with exponential tail distribution. The graph shattering has been researched in the viewpoint of attack tolerance [Albert



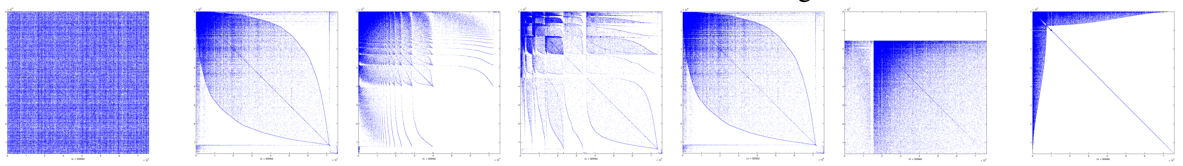
Flickr:
(a) Random (b) Natural (c) Degree Sort (d) Cross Association (e) Spectral Clustering (f) Shingle (g) SlashBurn



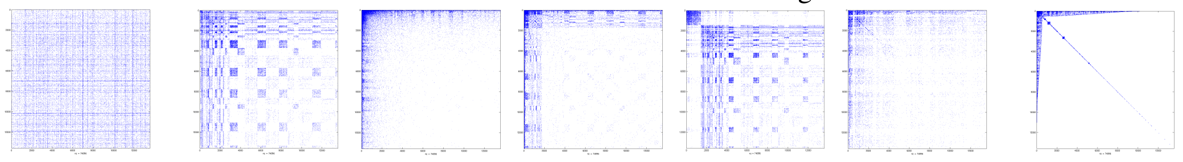
WWW-Barabasi:
(a) Random (b) Natural (c) Degree Sort (d) Cross Association (e) Spectral Clustering (f) Shingle (g) SlashBurn



Enron:
(a) Random (b) Natural (c) Degree Sort (d) Cross Association (e) Spectral Clustering (f) Shingle (g) SlashBurn



Epinions:
(a) Random (b) Natural (c) Degree Sort (d) Cross Association (e) Spectral Clustering (f) Shingle (g) SlashBurn



AS-Oregon:
(a) Random (b) Natural (c) Degree Sort (d) Cross Association (e) Spectral Clustering (f) Shingle (g) SlashBurn

Figure 9.7: Adjacency matrix of real world graphs on different ordering methods. Notice that SlashBurn requires the smallest number of square blocks to cover the edges, leading to the best compression as shown in Table 9.4.

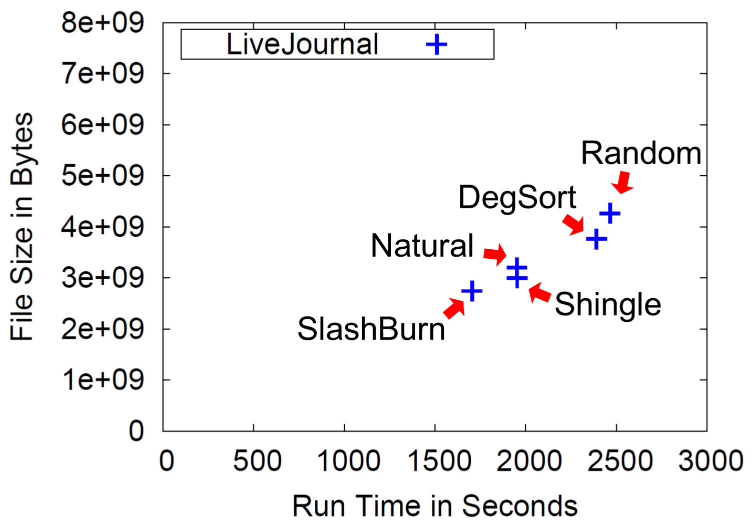


Figure 9.8: File size vs. running time of different ordering methods on LiveJournal graph. The running time is measured for one iteration of PageRank on HADOOP. Notice that SLASHBURN results in the smallest file size, as well as the smallest running time.

Graph Type	**Graph**	$w(G)$
Real world	Flickr	0.078
Real world	WWW-Barabasi	0.037
Real world	Enron	0.044
Real world	Epinions	0.099
Real world	AS-Oregon	0.040
Erdős-Rényi	ER-Epinions	0.611
Erdős-Rényi	ER-AS-Oregon	0.358

Table 9.5: Wing width ratio $w(G)$ of real world and random (Erdős-Rényi) graphs. Notice that $w(G)$'s are small for all the real world graphs, meaning that SLASHBURN works well on such graphs. In contrast, random graphs have high $w(G)$ (at least 6.2× larger than their real world counterparts), meaning that they cannot be shattered quickly.

et al., 2000] and characterizing real world graphs [Appel et al., 2009]. Chen et al. [Chen et al., 2007] studied the statistical behavior of a fragmentation measure from the removal of nodes in graphs. None of the previous works relate the shattering and the power law to the problem of node permutation for graph compression.

Graph Partition and Compression. There has been a lot of works on network community detection, including METIS and related works [Karypis and Kumar, 1999a, Satuluri and Parthasarathy, 2009], edge betweenness [Girvan and Newman, 2002], co-clustering [Dhillon et al., 2003, Papadimitriou and Sun, 2008], cross-associations [Chakrabarti et al., 2004], spectral clustering [Ng et al., 2002, Luxburg, 2007], and shingle-ordering [Chierichetti et al., 2009]. All of them aimed to find homogeneous regions in the graph so that cross edges between different regions are minimized. A recent result [Leskovec et al., 2008] studied real world networks using conductance, and showed that real world graphs do not have good cuts.

Graph compression has also been an active research topic. Boldi [Boldi and Vigna, 2004] studied the compression of Web graphs using the lexicographic localities; Chierichetti et al. [Chierichetti et al., 2009] extended it to the social networks; Apostolico et al. [Apostolico and Drovandi, 2009] used BFS based method for compression. Maserrat et al. [Maserrat and Pei, 2010] used multi-position linearizations for better serving neighborhood queries. Our SLASHBURN is the first work to take the power-law characteristic of most real world graphs into advantage for addressing the 'no good cut' problem and graph compression. Furthermore, our SLASHBURN is designed for large scale block based matrix vector multiplication where each square block is stored independently from each other for scalable processing in distributed platforms like MAPREDUCE [Dean and Ghemawat, 2004]. The previously mentioned works are not designed for this purpose: the information of the outgoing edges of a node is tightly inter-connected to the outgoing edges of its predecessor or successor, making them inappropriate for square block based distributed matrix vector multiplication.

9.5 Conclusion

In this chapter, we propose SLASHBURN, a novel method for laying out the edges of real world graphs, so that they can be easily compressed, and graph mining algorithms based on block matrix-vector multiplication can run quickly.

The main novelty is the focus on real world graphs, that typically have *no good cuts* [Leskovec et al., 2008], and thus cannot create good *caveman-like* communities and graph partitions. On the contrary, our SLASHBURN is tailored towards *jellyfish*-type graphs [Siganos et al., 2006], with spokes connected by hubs, and hubs connected by super-hubs, and so on, recursively. Our realistic view-point pays off: the resulting graph lay-outs enjoy

- faster processing times (e.g., matrix-vector multiplications, that are in the inner loop of most typical graph mining operations, like PageRank, connected components, etc), and
- lower disk space requirements.

Part IV

Conclusion

Chapter 10

Conclusion

Graphs are everywhere, from computer networks to the World Wide Web and social networks. Finding patterns and anomalies in very large graphs leads to useful applications including cyber-security, fraud-detection, and recommendation. Our goal is to design and implement large scale graph mining system, and use it to discover important patterns and anomalies. Toward the goal, in this thesis we present the theory, engineering, and discoveries of mining very large graphs using distributed MAPREDUCE/HADOOP platform. We carefully select a set of fundamental graph mining operations, and package them in PEGASUS (`http://www.cs.cmu.edu/~pegasus`), which, to the best of our knowledge, is the first such library, implemented on the top of the HADOOP platform.

One of the main research focus in this thesis is the structure analysis on large graphs. We developed graph structure analysis algorithms including HADI, a diameter estimation algorithm, and GIM-V, a unifying primitive for many different operations. We analyze real world graphs using HADI and GIM-V, and show surprising patterns including the 7-degrees of separation and anomalous connected components in the Web.

Another research focus is to develop advanced graph algorithms including inference in graph, eigenvalue analysis, and tensor analysis. We develop efficient algorithms for the tasks. We use the algorithms to analyze Twitter who-follows-whom graph to spot anomalous adult advertisers, and a large knowledge base tensor to discover potential synonyms among millions of noun phrases.

Lastly, we design algorithms for efficient graph management in distributed systems. Our graph management system reduces storage space, as well as accelerates queries. We also develop an edge layout algorithm to best compress graphs.

In the following, we summarize our contribution and present the vision for the future.

10.1 Summary of Contributions

We summarize the contributions and the impacts of this thesis.

10.1.1 Contributions

The contributions span three areas: basic graph algorithms, advanced graph algorithms, and graph management.

Basic Graph Algorithms:

- We propose HAdoop DIameter and radii estimator(HADI), a carefully designed and fine-tuned distributed algorithm to compute the radii and the diameter of massive graphs. Our optimization on HADI leads to 7.6× faster performance than the naive algorithm.
- We then generalize HADI to Generalized Iterated Matrix-Vector multiplication(GIM-V), a unifying primitive for many different graph mining operations including PageRank, spectral clustering, diameter estimation, and connected components. GIM-V is highly optimized, achieving good scale-up on the number of available machines, and linear running time on the number of edges. Our optimization on GIM-V leads to more than 5× faster performance than the naive algorithm.
- We employ HADI and GIM-V to study large, real world graphs. We are the first to discover the 7-degrees of separation of the Web.

Advanced Graph Algorithms:

- We propose HADOOP LINE GRAPH FIXED POINT (HA-LFP), an efficient distributed algorithm for inference in billion-scale graphs, using HADOOP platform. HA-LFP scales up linearly on the number of edges and machines.
- We propose HEIGEN, an eigensolver to perform spectral analysis on large graphs. HEIGEN handles 1000× larger matrices than the state of the art. We employ HEIGEN to analyze Twitter who-follows-whom graph and find anomalous accounts with huge number of triangles.
- We generalize the spectral analysis algorithm to multiple dimensions, and propose GIGATENSOR, a large scale tensor decomposition algorithm which can handle more than 100× larger tensors than the state of the art. We study a large knowledge base tensor, and present interesting discoveries which include the potential synonyms among millions of noun-phrases.

Graph Management:

- We propose GBASE, a scalable and general graph management system which provides a parallel indexing mechanism for graph mining operations that both saves storage space, as well as accelerates queries. GBASE reduces the storage space and the running time up to 50×.
- We propose SLASHBURN, an edge layout algorithm which utilizes the power-law characteristic of real world graphs for better compressing graphs. SLASHBURN consistently outperforms all the state of the art algorithms in terms of the compression ratio, and the running time for graph mining queries.

10.1.2 Technology Transfer

Our work in large graph mining and the PEGASUS system have impacts in academia as well as in industry. We summarize the technology transfers of our work.

- The PEGASUS system has been downloaded more than 410 times from 83 countries. It led to two U.S. patents, and won the award at the open source software world challenge.
- Microsoft included PEGASUS as part of their HADOOP distribution for Windows Azure.

- PEGASUS system is used as one of the core systems for several DARPA projects including Anomaly Detection At Multiple Scale (ADAMS).

10.2 Vision & Research Directions

Our vision is to design and implement a *big data analytics system* which finds useful patterns and anomalies, and thereby transforming massive raw data into valuable sources of knowledge. Toward this goal, we have researched on algorithms for scalable graph mining. In the near future, we intend to extend the algorithms and systems to support time evolving graphs, constrained problems, and approximate algorithms. In the long run, we plan to support rich data type, near-real time processing, and low-level distributed platform optimization in our big data analytics system to solve many real world problems. We elaborate our future vision and research directions in the following.

10.2.1 Medium Term Goals

Time Evolving Graphs

In most of the works in this thesis we assumed static graphs. However, many real world graphs are evolving over time, where edges and nodes are added or removed continuously. We want to extend our graph management system to handle time evolving graphs efficiently. For example, a promising direction is to study how to update the compressed graphs in distributed systems efficiently for the addition or removal of edges or nodes. Another example is to study how to update the computed graph features (e.g. connected components) on the new graph without re-computing the features from scratch. The effect is to answer graph mining queries quickly.

Large Graph Mining with Constraints

We want to enrich our graph mining algorithms to handle constraints. For example, we want to impose non-negativity constraints on the eigensolver or tensor decomposition algorithm to compute non-negative matrix or tensor factorizations. Another example is to impose sparsity on the tensor factorization to enable sparse factorization. The goal of adapting constraints is to apply graph mining algorithms in broader contexts with more interesting applications.

Approximate Computing

For very large graphs with more than billions of nodes and edges, exact algorithms can be very expensive in time or space. For example, computing neighborhoods from all nodes require quadratic storage to the number of nodes, which is too expensive. In such cases approximation algorithm, which gives reasonable accuracy while is much efficient, is very useful. A promising direction is on approximate graph mining algorithms. For example, we want to study how to compute approximate PageRanks to find top k highest PageRank pages. The goal is to enable computations whose exact algorithms are intractable.

10.2.2 Long Term Goals

Rich Data Types

Our current work focuses on simple graphs with weighted edges. However, there are many other types of data which can be supported by big data analytics systems: ‘colorful’ graphs (i.e. attributes in nodes and edges), time series, multivariate variables, gene information, text, and images. A promising direction is to incorporate these rich data types in our big data analytics system which can be applied to more diverse domains including astronomy, biology, accounting, health care, and environment monitoring, among others.

Near-Real Time Analytics

The MAPREDUCE/HADOOP platform is best for offline batch processing, but not suitable for online real-time processing. However some applications (e.g. network attack monitoring) require near-real time processing and responses to queries. A promising direction is to redesign the big data analytics system to support the near real time processing capability. The approaches include quickly combining precomputed statistics and new data. The goal is to enhance the applicability of the big data analytics system to broader areas.

Redesigning Distributed Computing Platform

In this thesis we developed algorithms on top of MAPREDUCE/HADOOP: we did not modify the design of the underlying distributed computing platform. Although MAPREDUCE/HADOOP provides easy interface, nice scalability, and fault tolerance, however, there are areas to improve on the underlying platform. A promising direction is to combine the best of the memory based systems and the disk based systems to improve performance (e.g. [Zaharia et al., 2010]). Another direction is to use and influence the location of files in the distributed file system (e.g. HDFS) to exploit the locality even better. The goal is to provide the best execution strategies based on the data access pattern of applications.

Bibliography

Divisi information. http://csc.media.mit.edu/docs/divisi2/. 102

Hadoop information. http://hadoop.apache.org/. 104

Jama information. http://math.nist.gov/javanumerics/jama/. 102

Mahout information. http://lucene.apache.org/mahout/. 102

D. J. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD Conference*, pages 967–980, 2008. 139

D. J. Abadi, P. A. Boncz, and S. Harizopoulos. Column oriented database systems. *PVLDB*, 2009. 139

E. Acar, C. Aykut-Bingol, H. Bingol, R. Bro, and B. Yener. Multiway analysis of epilepsy tensors. *Bioinformatics*, 23(13):i10–i18, 2007. 104, 119

L. Addario-Berry, W. S. Kennedy, A. D. King, Z. Li, and B. A. Reed. Finding a maximum-weight induced k-partite subgraph of an i-triangulated graph. *Discrete Applied Mathematics*, 158(7):765–770, 2010. 139

C. C. Aggarwal, Y. Xie, and P. S. Yu. Gconnect: A connectivity index for massive disk-resident graphs. *PVLDB*, 2009. 139

G. Aggarwal, M. Data, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. *Proceedings of FOCS*, 2004. 5

L. Akoglu, M. McGlohon, and C. Faloutsos. Oddball: Spotting anomalies in weighted graphs. In *PAKDD (2)*, pages 410–421, 2010. 125, 126

R. Albert, H. Jeong, and A.-L. Barabasi. Diameter of the world wide web. *Nature*, (401):130–131, 1999. 7, 27

R. Albert, H. Jeong, and A.-L. Barabasi. Error and attack tolerance of complex networks. *Nature*, 2000. 153

N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments, 1996. 14

P. Alpatov, G. Baker, C. Edward, J. Gunnels, G. Morrow, J. Overfelt, R. van de Gejin, and Y.-J. Wu. Plapack: Parallel linear algebra package - design overview. *SC97*, 1997. 102

I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani. k-core decompositions: A tool for the visualization of large scale networks. *http://arxiv.org/abs/cs.NI/0504107*. 125, 131

C. Andersson and R. Bro. The n-way toolbox for matlab. *Chemometrics and Intelligent Laboratory Systems*, 52(1):1–4, 2000. 104

P. Andritsos, R. J. Miller, and P. Tsaparas. Information-theoretic tools for mining database structure from large data sets. In *SIGMOD*, 2004. 139

A. Apostolico and G. Drovandi. Graph compression by bfs. *Algorithms*, 2(3):1031–1044, 2009. 156

A. P. Appel, D. Chakrabarti, C. Faloutsos, R. Kumar, J. Leskovec, and A. Tomkins. Shatterplots: Fast tools for mining large graphs. In *SDM*, 2009. 156

B. Awerbuch and Y. Shiloach. New connectivity and msf algorithms for ultracomputer and pram. *ICPP*, 1983. 55

B. Bader and T. Kolda. Matlab tensor toolbox version 2.2. *Albuquerque, NM, USA: Sandia National Laboratories*, 2007a. 104, 117

B. Bader, R. Harshman, and T. Kolda. Temporal analysis of social networks using three-way dedicom. *Sandia National Laboratories TR SAND2006-2161*, 2006. 119

B. W. Bader and T. G. Kolda. Efficient MATLAB computations with sparse and factored tensors. *SIAM Journal on Scientific Computing*, 30(1):205–231, December 2007b. 104, 109, 111

D. A. Bader and K. Madduri. A graph-theoretic analysis of the human protein-interaction network using multicore parallel algorithms. *Parallel Comput.*, 2008. 34

D. A. Bader, S. Kintali, K. Madduri, and M. Mihail. Approximating betweenness centrality. *WAW*, 2007. 139

B. T. Bartell, G. W. Cottrell, and R. K. Belew. Latent semantic indexing is an optimal special case of multidimensional scaling. *SIGIR*, pages 161–167, 1992. 82

M. W. Berry. Large scale singular value computations. *International Journal of Supercomputer Applications*, 1992. 82, 97

K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. SIGMOD, 2007. ISBN 978-1-59593-686-8. doi: http://doi.acm.org/10.1145/1247480.1247504. 14

L. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, and I. Dhillon. Scalapack users's guide. *SIAM*, 1997. 102

P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *WWW*, 2004. 127, 150, 156

S. P. Borgatti and M. G. Everett. A graph-theoretic perspective on centrality. *Social Networks*, 2006. 146

S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004. ISBN 0521833787. 110

S. Brin and L. Page. The anatomy of a large-scale hypertextual (web) search engine. In *Proc. 7th International World Wide Web Conference (WWW7)/Computer Networks*, pages 107–117, 1998. Published as Proc. 7th International World Wide Web Conference (WWW7)/Computer Networks, volume 30, number 1-7. 37, 83, 89

R. Bro. Parafac. tutorial and applications. *Chemometrics and intelligent laboratory systems*, 38(2):149–171, 1997. 107, 108, 119

A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. Wiener. Graph structure in the web. *Computer Networks 33*, 2000. 1, 27, 34

A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. H. Jr., and T. M. Mitchell. Toward an architecture for never-ending language learning. In *AAAI*, 2010. 103, 104, 116

R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: easy and efficient parallel processing of massive data sets. *VLDB*, 2008. 6

D. Chakrabarti, S. Papadimitriou, D. S. Modha, and C. Faloutsos. Fully automatic cross-associations. In *KDD*, pages 79–88, 2004. 141, 145, 150, 156

M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. PODS, 2000. ISBN 1-58113-214-X. doi: http://doi.acm.org/10.1145/335168.335230. 14

D. H. Chau, S. Pandit, and C. Faloutsos. Detecting fraudulent personalities in networks of online auctioneers. *PKDD*, 2006. 60, 61

Y. Chen, G. Paul, R. Cohen, S. Havlin, S. P. Borgatti, F. Liljeros, and H. E. Stanley. Percolation theory and fragmentation measures in social networks. In *Physica A 378*, pages 11–19, 2007. 156

P. Chew, B. Bader, T. Kolda, and A. Abdelali. Cross-language information retrieval using parafac2. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 143–152. ACM, 2007. 119

F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *KDD*, pages 219–228, 2009. 139, 141, 145, 150, 153, 156

R. L. F. Cordeiro, C. T. Jr., A. J. M. Traina, J. López, U. Kang, and C. Faloutsos. Clustering very large multi-dimensional datasets with mapreduce. In *KDD*, pages 690–698, 2011. 6

T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990. 34

J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*, 2004. 1, 5, 104, 144, 156

S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, Sept. 1990. 82, 97, 119

J. W. Demmel. Applied numerical linear algebra. *SIAM*, 1997. 94

I. S. Dhillon, S. Mallela, and D. S. Modha. Information-theoretic co-clustering. In *KDD*, pages 89–98, 2003. 141, 145, 156

R. Dunbar. Grooming, gossip, and the evolution of language. *Harvard Univ Press*, October 1998. 52

D. M. Dunlavy, T. G. Kolda, and E. Acar. Temporal link prediction using matrix and tensor factorizations. *TKDD*, 5(2):10, 2011. 82

C. Eckart and G. Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1 (3):211–218, 1936. 97, 106

P. Erdős and A. Rényi. On random graphs. *Publicationes Mathematicae*, 6:290–297, 1959. 7, 23, 133, 153

M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. *SIGCOMM*, pages 251–262, Aug-Sept. 1999. 86, 87, 143, 153

P. Felzenszwalb and D. Huttenlocher. Efficient belief propagation for early vision. *International journal of computer vision*, 70(1):41–54, 2006. 60, 61

J.-A. Ferrez, K. Fukuda, and T. Liebling. Parallel computation of the diameter of a graph. In *HPCSA*, 1998. 34

P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 1985. URL `citeseer.ist.psu.edu/`

`flajolet85probabilistic.html`. 14, 15

M. N. Garofalakis and P. B. Gibbons. Approximate query processing: Taming the terabytes. In *VLDB*, 2001. 14

M. Girvan and M. Newman. Community structure in social and biological networks. *National Academy of Sciences*, 99:7821–7826, 2002. 156

G. H. Golub and C. F. Van Loan. Matrix computations. *Johns Hopkins University Press*, 1996. 90

J. Gonzalez, Y. Low, C. Guestrin, and D. O'Hallaron. Distributed parallel inference on large factor graphs. In *Conference on Uncertainty in Artificial Intelligence (UAI)*, Montreal, Canada, July 2009a. 62

J. E. Gonzalez, Y. Low, and C. Guestrin. Residual splash for optimally parallelizing belief propagation. *AISTAT*, 2009b. 62

J. Greiner. A comparison of parallel algorithms for connected components. *Proceedings of the 6th ACM Symposium on Parallel Algorithms and Architectures*, June 1994. 55

R. L. Grossman and Y. Gu. Data mining using high performance data clouds: experimental studies using sector and sphere. *KDD*, 2008. 6

M. R. Guarracino, F. Perla, and P. Zanetti. A parallel block lanczos algorithm and its implementation for the evaluation of some eigenvalues of large sparse symmetric matrices on multicomputers. *Int. J. Appl. Math. Comput. Sci.*, 2006. 102

R. Harshman. Foundations of the parafac procedure: Models and conditions for an" explanatory" multi-modal factor analysis. 1970. 107, 108

S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. A. Boncz. Positional update handling in column stores. In *SIGMOD*, 2010. 139

D. Hirschberg, A. Chandra, and D. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979. 55

M. Ivanova, M. L. Kersten, N. J. Nes, and R. Goncalves. An architecture for recycling intermediates in a column-store. In *SIGMOD*, 2009. 139

M. Kamel. Computing the singular value decomposition in image processing. In *Proceedings of Conference on Information Systems*, Princeton, 1984. 82, 97

U. Kang, C. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations. *ICDM*, 2009. 6, 125

U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Radius plots for mining tera-byte scale graphs: Algorithms, patterns, and observations. In *SDM*, pages 548–558, 2010. 6, 125

U. Kang, D. H. Chau, and C. Faloutsos. Mining large graphs: Algorithms, inference, and discoveries. In *ICDE*, pages 243–254, 2011a. 6

U. Kang, B. Meeder, and C. Faloutsos. Spectral analysis for billion-scale graphs: Discoveries and implementation. In *PAKDD (2)*, pages 13–25, 2011b. 6

U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: a scalable and general graph management system. In *KDD*, pages 1091–1099, 2011c. 6

U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Hadi: Mining radii of large graphs. *ACM Trans. Knowl. Discov. Data*, 5:8:1–8:24, February 2011d. ISSN 1556-4681. doi: http://doi.acm.org/10.1145/1921632.1921634. 6

U. Kang, C. E. Tsourakakis, and C. Faloutsos. Pegasus: mining peta-scale graphs. *Knowl. Inf. Syst.*, 27

(2):303–325, 2011e. 6

G. Karypis and V. Kumar. Multilevel -way hypergraph partitioning. In *DAC*, pages 343–348, 1999a. 127, 156

G. Karypis and V. Kumar. Parallel multilevel k-way partitioning for irregular graphs. *SIAM Review*, 41 (2):278–300, 1999b. 139

J. Kleinberg. Authoritative sources in a hyperlinked environment. *Journal of the ACM (JACM)*, 46(5): 604–632, 1999. 99, 119

B. Klimt and Y. Yang. The enron corpus: A new dataset for email classification research. In *ECML*, pages 217–226, 2004. 7

T. Kolda and B. Bader. The tophits model for higher-order web link analysis. In *Workshop on Link Analysis, Counterterrorism and Security*, volume 7, pages 26–29, 2006. 103, 104, 119

T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3):455–500, 2009. 82, 107, 108

T. G. Kolda and J. Sun. Scalable tensor decompositions for multi-aspect data mining. In *ICDM*, pages 363–372, 2008. 82, 103, 104

J. B. Kruskal and M. Wish. *Multidimensional scaling*. SAGE publications, Beverly Hills, 1978. 82

R. Lämmel. Google's mapreduce programming model – revisited. *Science of Computer Programming*, 70:1–30, 2008. 6

C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *J. Res. Nat. Bur. Stand.*, 1950. 90

R. B. Lehoucq, D. C. Sorensen, and C. Yang. *ARPACK users' guide - solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*. Software, environments, tools. SIAM, 1998. ISBN 978-0-89871-407-4. 102

J. Leskovec and C. Faloutsos. Sampling from large graphs. *KDD*, pages 631–636, 2006. 5

J. Leskovec, D. Chakrabarti, J. M. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. *PKDD*, pages 133–145, 2005. 7, 12, 23, 25, 45, 46, 72, 83, 100, 133, 135

J. Leskovec, J. Kleinberg, and C. Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1(1):2, 2007. ISSN 1556-4681. doi: http://doi.acm.org/10.1145/1217299.1217301. 27, 32

J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Statistical properties of community structure in large social and information networks. In *WWW*, pages 695–704, 2008. 28, 143, 145, 156

T. G. Lewis. Network science: Theory and applications. *Wiley*, 2009. 12

C.-Y. Lin, N. Cao, S. Liu, S. Papadimitriou, J. Sun, and X. Yan. Smallblue: Social network analysis for expertise search and collective intelligence. In *ICDE*, pages 1483–1486, 2009. 125

C. Liu, F. Guo, and C. Faloutsos. Bbm: bayesian browsing model from petabyte-scale data. In *KDD*, pages 537–546, 2009. 123

C. Liu, H. chih Yang, J. Fan, L.-W. He, and Y.-M. Wang. Distributed nonnegative matrix factorization for web-scale dyadic data analysis on mapreduce. In *WWW*, pages 681–690, 2010. 115

U. Luxburg. A tutorial on spectral clustering. *Statistics and Computing*, 17(4):395–416, 2007. ISSN 0960-3174. doi: http://dx.doi.org/10.1007/s11222-007-9033-z. 82, 99, 156

J. Ma and S. Ma. Efficient parallel algorithms for some graph theory problems. *JCST*, 1993. 34

K. Maruhashi, F. Guo, and C. Faloutsos. Multiaspectforensics: Pattern mining on large-scale heterogeneous networks with tensor analysis. In *Proceedings of the Third International Conference on Advances in Social Network Analysis and Mining*, 2011. 103, 104, 119

H. Maserrat and J. Pei. Neighbor query friendly compression of social networks. In *KDD*, 2010. 139, 156

M. Mcglohon, L. Akoglu, and C. Faloutsos. Weighted graphs and disconnected components: patterns and a generator. *KDD*, pages 524–532, 2008. 32, 48, 52, 78

M. McGlohon, S. Bay, M. Anderle, D. Steier, and C. Faloutsos. Snare: a link analytic system for graph labeling and risk detection. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1265–1274. ACM, 2009. 60, 61

A. Mendiburu, R. Santana, J. Lozano, and E. Bengoetxea. A parallel framework for loopy belief propagation. *GECCO*, 2007. 62

M. Newman. A measure of betweenness centrality based on random walks. *Social Networks*, 2005. 12

M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, (46):323–351, 2005. 48

A. Y. Ng, M. I. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. *NIPS*, 2002. 82, 99, 141, 145, 156

C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08*, pages 1099–1110, 2008. 6

L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998. URL `http://dbpubs.stanford.edu/pub/1999-66`. 125, 139

C. R. Palmer, P. B. Gibbons, and C. Faloutsos. Anf: a fast and scalable tool for data mining in massive graphs. *KDD*, pages 81–90, 2002. 14

J.-Y. Pan, H.-J. Yang, C. Faloutsos, and P. Duygulu. Automatic multimedia cross-modal correlation discovery. *ACM SIGKDD*, Aug. 2004. 38

G. Pandurangan, P. Raghavan, and E. Upfal. Using pagerank to characterize web structure. *COCOON*, August 2002. 55

S. Papadimitriou and J. Sun. Disco: Distributed co-clustering with map-reduce. *ICDM*, 2008. 6, 41, 127, 156

A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. Dewitt, S. Madden, and M. Stonebraker. A comparison of approaches to large-scale data analysis. SIGMOD, June 2009. URL `http://database.cs.brown.edu/sigmod09/benchmarks-sigmod09.pdf`. 24

J. Pearl. Reverend Bayes on inference engines: A distributed hierarchical approach. In *Proceedings of the AAAI National Conference on AI*, pages 133–136, 1982. 60, 61, 62

J. Pearl. *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufmann, 1988. 62

K. Pearson. On lines and planes of closest fit to systems of points in space. *Philosophical Magazine Series 6*, 2(11):559–572, 1901. 82, 97

R. Penrose. A generalized inverse for matrices. In *Proc. Cambridge Philos. Soc*, volume 51, pages 406–413. Cambridge Univ Press, 1955. 98, 106

R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal*, 2005. 6

B. A. Prakash, M. Seshadri, A. Sridharan, S. Machiraju, and C. Faloutsos. Eigenspokes: Surprising patterns and community structure in large graphs. *PAKDD*, 2010a. 84

B. A. Prakash, H. Tong, N. Valler, M. Faloutsos, and C. Faloutsos. Virus propagation on time-varying networks: Theory and immunization algorithms. In *ECML/PKDD*, 2010b. 146

M. Richardson, R. Agrawal, and P. Domingos. Trust management for the semantic web. In *International Semantic Web Conference*, pages 351–368, 2003. 7

J. Rissanen and G. G. Langdon Jr. Arithmetic coding. *IBM Journal of Research and Development*, 23(2): 149–162, 1979. 145

P. Sarkar and A. W. Moore. Fast nearest-neighbor search in disk-resident graphs. In *KDD*, pages 513–522, 2010. 139

V. Satuluri and S. Parthasarathy. Scalable graph clustering using stochastic flows: applications to community discovery. *KDD*, 2009. 156

J. Shi and J. Malik. Normalized cuts and image segmentation. *CVPR*, 1997. 82, 99, 141, 145, 150

Y. Shiloach and U. Vishkin. An o(logn) parallel connectivity algorithm. *Journal of Algorithms*, pages 57–67, 1982. 55

N. Sidiropoulos, G. Giannakis, and R. Bro. Blind parafac receivers for ds-cdma systems. *Signal Processing, IEEE Transactions on*, 48(3):810–823, 2000. 119

G. Siganos, S. L. Tauro, and M. Faloutsos. Jellyfish: A conceptual model for the as internet topology. *Journal of Communications and Networks*, 2006. 156

B. P. Sinha, B. B. Bhattacharya, S. Ghose, and P. K. Srimani. A parallel algorithm to compute the shortest paths and diameter of a graph and its vlsi implementation. *IEEE Trans. Comput.*, 1986. 34

Y. Song, W. Chen, H. Bai, C. Lin, and E. Chang. Parallel spectral clustering. In *ECML*, 2008. 83, 102

M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented dbms. In *VLDB*, 2005. 139

J. Sun, H. Qu, D. Chakrabarti, and C. Faloutsos. Neighborhood formation and anomaly detection in bipartite graphs. In *ICDM*, pages 418–425, 2005a. 125, 126

J. Sun, H. Zeng, H. Liu, Y. Lu, and Z. Chen. Cubesvd: a novel approach to personalized web search. In *Proceedings of the 14th international conference on World Wide Web*, pages 382–390. ACM, 2005b. 119

J. Sun, S. Papadimitriou, and P. S. Yu. Window-based tensor analysis on high-dimensional and multi-aspect streams. In *ICDM*, pages 1076–1080, 2006a. 103

J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: dynamic tensor analysis. *KDD*, pages 374–383, 2006b. 82

D. Tao, X. Li, X. Wu, W. Hu, and S. Maybank. Supervised tensor learning. *Knowledge and Information Systems*, 13(1):1–42, 2007. 119

D. Tao, M. Song, X. Li, J. Shen, J. Sun, X. Wu, C. Faloutsos, and S. Maybank. Bayesian tensor approach for 3-d face modeling. *Circuits and Systems for Video Technology, IEEE Transactions on*, 18(10): 1397–1410, 2008. 119

H. Tong, C. Faloutsos, and J.-Y. Pan. Fast random walk with restart and its applications. In *ICDM*, pages 613–622, 2006. 125, 126, 139

L. N. Trefethen and D. Bau III. Numerical linear algebra. *SIAM*, 1997. 87, 89, 92

S. Trißl and U. Leser. Fast and practical indexing and querying of very large graphs. In *SIGMOD*, 2007. 139

C. E. Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *ICDM*, pages 608–617, 2008. 82, 86

C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *KDD*, pages 837–846, 2009. 86

Y. Weiss and W. Freeman. Correctness of belief propagation in gaussian graphical models of arbitrary topology. *Neural Computation*, 13(10):2173–2200, 2001. 62

K. Wu and H. Simon. A parallel lanczos method for symmetric generalized eigenvalue problems. *Computing and Visualization in Science*, 1999. 83, 102

D. Xin, J. Han, X. Yan, and H. Cheng. Mining compressed frequent-pattern sets. In *VLDB*, pages 709–720, 2005. 139

X. Yan, H. Cheng, J. Han, and P. S. Yu. Mining significant graph patterns by leap search. In *SIGMOD Conference*, pages 433–444, 2008. 139

J. S. Yedidia, W. T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. *Exploring Artificial Intelligence in the New Millenium*, 2003. 60, 62

M. Zaharia, N. M. M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. Technical Report UCB/EECS-2010-53, EECS Department, University of California, Berkeley, May 2010. 161

H. Zha, X. He, C. Ding, M. Gu, and H. Simon. Spectral relaxation for k-means clustering. *Advances in Neural Information Processing Systems*, 2:1057–1064, 2002. 97

P. Zhao, J. X. Yu, and P. S. Yu. Graph indexing: Tree + delta >= graph. In *VLDB*, 2007a. 139

Y. Zhao, X. Chi, and Q. Cheng. An implementation of parallel eigenvalue computation using dual-level hybrid parallelism. *Lecture Notes in Computer Science*, 2007b. 102