Big Graph Mining: Algorithms and Discoveries

U Kang and Christos Faloutsos
Carnegie Mellon University
{ukang, christos}@cs.cmu.edu

ABSTRACT

How do we find patterns and anomalies in very large graphs with billions of nodes and edges? How to mine such big graphs efficiently? Big graphs are everywhere, ranging from social networks and mobile call networks to biological networks and the World Wide Web. Mining big graphs leads to many interesting applications including cyber security, fraud detection, Web search, recommendation, and many more.

In this paper we describe PEGASUS, a big graph mining system built on top of MAPREDUCE, a modern distributed data processing platform. We introduce GIM-V, an important primitive that PEGASUS uses for its algorithms to analyze structures of large graphs. We also introduce HEIGEN, a large scale eigensolver which is also a part of PEGASUS. Both GIM-V and HEIGEN are highly optimized, achieving linear scale up on the number of machines and edges, and providing 9.2× and 76× faster performance than their naive counterparts, respectively.

Using PEGASUS, we analyze very large, real world graphs with billions of nodes and edges. Our findings include anomalous spikes in the connected component size distribution, the 7 degrees of separation in a Web graph, and anomalous adult advertisers in the who-follows-whom Twitter social network.

1. INTRODUCTION

Graphs are ubiquitous: computer networks [15], social networks [14], mobile call networks [48], biological networks [6], citation networks [18], and the World Wide Web [9], to name a few. Spurred by the lower cost of disk storage, the success of social networking sites (e.g. Facebook, Twitter, and Google+) and Web 2.0 applications, and the high availability of data sources, graph data are being generated at an unparalleled rate. They are now measured in terabytes and heading toward petabytes, with more than billions of nodes and edges. For example, Facebook loads 60 terabytes of new data every day [44]; Yahoo had a 1.4 billion nodes Web graphs at 2002 [26]; Microsoft had 1.15 billion query-URL pairs at 2009 [33]; and Google processes 20 petabytes per day [11]. Mining such big graphs helps us find patterns and anomalies which lead to many interesting applications including fraud detection, cyber security, social network analysis, etc.

Traditional graph algorithms assume the input graph fits in the memory or disks of a single machine. However, the recent growth of the sizes in graphs break this assumption. Since single machine algorithms are not tractable for handling big graphs, we naturally turn to distributed algorithms. Among many candidates, we use HADOOP, the open source version of MAPREDUCE, for its scalability, fault tolerance, and ease of accessibility. On top of HADOOP, we built the PEGASUS graph mining software, available as an open source in `http://www.cs.cmu.edu/~pegasus`, which includes various graph mining algorithms including PageRank [39], Random Walk with Restart (RWR) [40], diameter/radius estimation [27], connected components [21], and eigensolver [22].

This paper has two focuses. The first is the design of scalable graph mining algorithms in MAPREDUCE. We will see several useful techniques including approximation to make linear algorithms (Section 3.1), compression and clustering to decrease the amount of disk accesses (Section 3.2), and handling skewed data distribution (Section 4.2). The second is the discoveries of patterns and anomalies from real world graphs, using PEGASUS. We present results from several big graphs including the Twitter social network and the Web graph snapshot at 2002 from Yahoo!.

The rest of the paper is organized as follows. Section 2 describes the related works. In Section 3 we describe the algorithm for structure analysis, and in Section 4 we describe the algorithm for spectral analysis of large graphs. In Section 5 we present the discoveries in real world, large scale graphs. After discussing future research directions in Section 6, we conclude in Section 7.

2. RELATED WORKS

In this section, we review related works on MAPREDUCE, and distributed big graph mining.

2.1 MapReduce

MAPREDUCE is a programming framework [11] for processing massive amount of data in a distributed fashion. MAPREDUCE provides users a simple interface for programming distributed algorithm, and it handles all the details of data distribution, replication, fault tolerance, and load balancing. A typical MAPREDUCE job consists of three stages: map, shuffle, and reduce. At the map stage, the raw data is read and processed to output (key, value) pairs. At the shuffle stage, the output of the map stage is sent to reducers via network so that the pairs with the same key are grouped together. At the reduce stage, the (key, value) pairs with the same key are processed to output another (key, value) pair. An iterative MAPREDUCE program runs several MAPREDUCE jobs,

by feeding the output of the current job as the input of the next job.
HADOOP [1] is the open source implementation of MAPREDUCE. HADOOP uses the Hadoop Distributed File System (HDFS) for its file system. There are several packages that runs on top of HADOOP, including PIG [38], a high level language for HADOOP, and HBASE [2], a column-oriented data storage on top of HADOOP. Due to the simplicity, scalability, and fault tolerance, big graph mining using HADOOP attracted significant attentions in research community [41; 3; 23; 17].

2.2 Distributed Big Graph Mining

There are several works on distributed big graph mining which can be grouped into two: (1) one not based on MAPREDUCE/HADOOP, and (2) the other on top of it.
The works not based on MAPREDUCE/HADOOP include GraphLab, Pregel, and Trinity. GraphLab [34] provides a framework for parallel machine learning and data mining, in a shared memory setting. Recently, they provide the distributed GraphLab [35] for shared nothing machines. Pregel [36] is a system for large scale graph processing where vertices exchange messages and change their states in memory. Trinity [43] is a memory-based distributed database and computation platform. In general, those systems do not match the MAPREDUCE/HADOOP's high degree of fault tolerance capabilities including 3-way replication and speculative execution.
On the MAPREDUCE/HADOOP side, Apache Mahout [3], a scalable machine learning library on HADOOP, provides a different set of operations compared to PEGASUS.

3. ALGORITHM FOR STRUCTURE ANALYSIS

How can we analyze many structural properties (e.g. PageRank, connected component, diameter, radius, etc.) of big graphs efficiently with a simple and general primitive? In this section, we describe algorithms for mining the structure of big graphs. We first introduce GIM-V, a general primitive for big graph mining, and describe efficient algorithm in MAPREDUCE.

3.1 GIM-V

How can we unify many graph mining algorithms, including connected components, diameter, PageRank, and node proximities? We introduce Generalized Iterative Matrix-Vector multiplication (GIM-V) to answer the question. The main intuition of GIM-V is that many graph mining algorithms can be formulated by iterative message exchanges with adjacent nodes. We observe that the message exchange is equivalent to performing matrix vector multiplication on the adjacency matrix of the graph and the vector containing current states of nodes. Based on the intuition and the observation, our approach is to formulate many graph mining algorithms using a generalized form of matrix vector multiplication where the three internal operations (multiply, sum, and assign) are redefined based on the specific instantiation of the matrix vector multiplication.
Consider multiplying an n by n matrix M and a length n-vector v to produce an output vector v'. The ith element v'_i of the vector v' is determined by

$$v'_i = \sum_{j=1}^{n} M_{i,j} v_j.$$

where $M_{i,j}$ is the (i,j)th element of M. The above equation contains the following three internal operations: (1) multiply $M_{i,j}$ and v_j, (2) sum n multiplication results, and (3) write the new result to v'_i. GIM-V generalizes the three operations as follows:

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

In terms of the generalized operation, the v'_i is expressed by

$$v'_i = \texttt{assign}(v_i, \texttt{combineAll}(\{x_j \mid j = 1..n, \text{ and } x_j = \texttt{combine2}(M_{i,j}, v_j)\})).$$

GIM-V is run iteratively; i.e., the output of the current iteration is fed as the input of the next iteration. The iteration is continued until algorithm-specific termination condition is satisfied. In the following we will see how the new definitions of the three operations leads to many different algorithms. The termination conditions for each algorithm are also described.

3.1.1 PageRank

PageRank is a well-known algorithm for ranking Web pages [39]. Given a graph, PageRank outputs a ranking vector which denotes the stationary probability that a random surfer would end up on specific nodes. The PageRank vector p satisfies the eigenvector equation $p = (cE^T + (1-c)U)p$ where E is the row-normalized adjacency matrix of the graph, c is a damping factor typically set to 0.85, and U is a matrix all of whose elements are $\frac{1}{n}$ where n is the number of nodes in the graph. PageRank is a special case of GIM-V. Let $M = E^T$. Then the next PageRank vector p^{next} is computed from the current vector p^{cur} by $p^{next} = M \times p^{cur}$ where the three operations are defined as follows:

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

The termination condition is when the PageRank vector does not change up to a threshold.

3.1.2 Random Walk with Restart

Random Walk with Restart (RWR) [40; 45] is used to compute the personalized PageRank. The RWR vector p_k, specifying the proximities of nodes from the query node k, satisfies the eigenvector equation $p_k = cE^T p_k + (1-c)e_k$ where c and E are defined as in Section 3.1.1, and e_k is an n-vector whose kth element is 1, and all other elements are 0. As in Section 3.1.1, let $M = E^T$. Then the next RWR vector p_k^{next} is computed from the current vector p_k^{cur} by $p_k^{next} = M \times p_k^{cur}$ where the three operations are defined as follows (δ_{ik} is the *Kronecker delta*: $\delta_{ik} = 1$ if $i = k$. and 0 otherwise):

1. $\texttt{combine2}(m_{i,j}, v_j) = c \times M_{i,j} \times v_j$.

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

As in the case of PageRank, the termination condition is when the RWR vector does not change up to a threshold.

3.1.3 Connected Component

Computing the weakly connected component is also a special case of GIM-V. The main idea is to propagate the minimum node id in each component until all the nodes in each component see the message with the minimum node id in the component. Let M be the adjacency matrix of a graph. The next connected component vector p^{next} is computed from the current vector p^{cur} by $p^{next} = M \times p^{cur}$ where the three operations are defined as follows:

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

where $MIN()$ is a function that returns the minimum of its arguments. The iteration is continued until the connected component vector does not change. The maximum number of iteration is bound by the graph's diameter which is surprisingly small in many real world graphs [5; 31; 27].

3.1.4 Diameter and Radius

The radius of a node v is the maximum of the shortest distances to other nodes in a graph. The diameter of a graph is the maximum radius over all the nodes. Exactly computing the diameter and the radius is prohibitively expensive ($O(n^2)$); however, fortunately we can use GIM-V for approximating the diameter and radius in a time linear to the number of edges.

The main idea is to use a small, probabilistic bitstring to store the information on the neighborhood of each node [27]. Let b_i^h denote the probabilistic bitstring containing the neighborhood information reachable from node i within h hops, and M be the adjacency matrix of a graph. The $(h+1)$ hop neighborhood b_i^{h+1} from node i is computed from the h hop neighborhood b_i^h by $b_i^{h+1} = M \times b_i^h$ where the three operations are defined as follows:

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

The termination condition is when the bitstrings of all nodes do not change.

3.2 Fast Algorithm for GIM-V

The next question is, how to design fast algorithms for GIM-V in MapReduce? Our main idea is to cluster the nonzero elements of the adjacency matrix, encode it using blocks, and compress it to decrease the amount of data traffic in MapReduce computation. We describe our proposed method in a progressive way; i.e., we first describe the naive implementation of GIM-V in MapReduce, and propose several ideas to make it more efficient.

3.2.1 GIM-V RAW: Naive Algorithm.

The input data is the matrix and the vector stored in disks. We assume that each of the matrix and the vector is too large to fit in the memory of a single machine. Each nonzero element of the matrix is stored in one line as a tuple (source, destination, value); similarly, each non-zero element of the vector is stored in one line as a tuple (row, value). The naive algorithm of GIM-V on MapReduce comprises two stages. In the first stage, the matrix elements and the vector elements are joined to make partial results, where the column id of the matrix elements and the row id (index) of the vector elements are used as keys. In the second stage, the partial results are aggregated to make an output vector. Note that this algorithm takes the original matrix as it is; we will see reorganizing and compressing the matrix greatly helps for improving the efficiency in the following.

3.2.2 GIM-V NNB: Block Multiplication.

The first idea to improve the performance of GIM-V RAW is to perform block multiplication instead of element wise multiplication. We first group matrix elements into w by w square blocks; similarly, vector elements are grouped into length w vectors. It can be easily verified that the original matrix vector multiplication can be expressed by block-based multiplication [26]. This GIM-V NNB method, if combined with the compression idea described in the next section, decrease the size of the data significantly.

3.2.3 GIM-V NCB: Compression.

The advantage of the block encoding introduced in GIM-V NNB is that it opens the possibility to exploit the locality of nonzero elements inside blocks. GIM-V NCB compresses the nonzero elements of each block using standard compression algorithms like Gzip or Elias-γ, and store the compressed data into disks. The compressed block is read from disks, and uncompressed in run-time. Despite the additional time for the uncompression, GIM-V NCB help decrease the overall running time since the disk space decreases significantly, and disk access time dominates the running time. The numerical performance gain is described in the next section.

3.2.4 GIM-V CCB: Clustering.

GIM-V NCB can be further improved by clustering nonzero elements of the matrix. The average bits required to encode an element in the matrix decrease as the density of each block increases. The implication of the fact is that few dense blocks require smaller disk space than many sparse blocks. To get few dense blocks, we can cluster the nonzero elements of the adjacency matrix. GIM-V CCB uses graph clustering algorithm to achieve this. There are many graph clustering algorithms available to be used as a plug-in for GIM-V CCB, including METIS [28], co-clustering [41], Shingle ordering [10], and SlashBurn [20].

The effect of applying these three techniques is significant. In our experiment [25], GIM-V CCB achieved 43× smaller storage, as well as 9.2× faster running time for matrix-vector multiplication compared to GIM-V RAW. In addition, GIM-V CCB scales linearly on the number of machines and edges, thanks to the matrix-vector formulation.

4. ALGORITHM FOR SPECTRAL ANALYSIS

How can we perform the spectral analysis (e.g. Singular Value Decomposition (SVD), triangles, etc.) on very large

graphs? In this section we describe HEigen, our proposed distributed algorithm for spectral graph analysis.

4.1 HEigen

Spectral analysis on graphs using top k eigenvalues and eigenvectors is a crucial technique with many applications including SVD, dimensionality reduction, triangle counting [47], and community detection [42].

Despite the importance of the task, existing eigensolvers do not scale well, handling up to millions of nodes and edges at most. To address the problem we developed HEigen, a distributed eigensolver on MapReduce for very large graphs with more than billions of nodes and edges. There are several candidates of the serial eigensolver algorithms that we considered for HEigen; we list them and discuss their advantages and disadvantages.

- **Power Method.** The first candidate is the power method [19], a well-known algorithm to compute the principal eigenvector of matrices. The power method is very efficient since the most expensive operation per iteration is a matrix-vector multiplication. However, it cannot compute top k eigenvalues which is a significant drawback for our purpose.
- **Simultaneous Iteration.** The second candidate is the simultaneous iteration (or QR) algorithm which essentially applies power method to several vectors at once [46]. Although it computes top k eigenvalues, it is not efficient since it requires several matrix-matrix multiplications which are expensive.
- **Basic Lanczos.** The third candidate is the basic Lanczos [30] algorithm. Lanczos algorithm computes top k eigenvalues, and it is efficient since the most expensive operation per iteration is the matrix-vector multiplication, as in the power method. A drawback of the basic Lanczos algorithm is that it requires orthogonalization of the output vector after each iteration with all previous output vectors, which is prohibitively expensive.

An improvement of the basic Lanczos algorithm is to selectively choose previous output vectors to be orthogonalized against the new basis vector; this Lanczos-SO (Selective Orthogonalization) [12] requires much fewer orthogonalizations than the basic Lanczos, and thus much more efficient than the basic Lanczos. For the reason HEigen uses Lanczos-SO as the basis for the eigensolver.

4.2 Fast Algorithm for HEigen

How can we design a fast algorithm for HEigen in MapReduce? There are two main operations in HEigen: matrix-vector multiplication for main iterations, and skewed matrix-matrix multiplication for post-processing the results of main iterations to compute the eigenvectors (see [22] for details). The matrix-vector multiplication is performed efficiently in HEigen using the block encoding idea as described in Section 3.2. The challenging part is the skewed matrix-matrix multiplication where the first matrix $A^{n \times m}$ is much larger than the second matrix $B^{m \times k}$, where $n \gg m > k$. A naive algorithm of multiplying the two matrices is to use two stages of MapReduce jobs. In the first stage, partial multiplication results are generated using the column ids of the first matrix and the row ids of the second matrix as keys. That is, for each column i of A, the cross product from the elements of A having i as column id, and the elements of B having i as row id, is computed. In the second stage, the partial results from the first stage are summed together. The problem of this naive algorithm is that it requires many disk I/Os. Let $|\cdot|$ denote the number of nonzero elements in a matrix or a vector. For example, $|A|$ is the number of nonzero elements of the matrix A, $|A_{:,i}|$ is the number of nonzero elements in the ith column of A, and $|B_{i,:}|$ is the number of nonzero elements in the ith row of B. Let $T = \sum_{i=1}^{m} |A_{:,i}| \cdot |B_{i,:}|$ be the number of intermediate elements to be written to and read from disk after the first stage. Then, the naive algorithm requires the following disk accesses and network transfer:

- Disk read: $2(|A| + |B| + T)$.
- Disk write: $|AB| + |A| + |B| + 2T$.
- Network transfer (shuffle stage): $|A| + |B| + T$.

HEigen uses the distributed cache based algorithm where the second, small matrix is broadcasted all the mappers. Then, mappers themselves join the first matrix with the small matrix, and the intermediate results are aggregated in reducers, finishing the task using one MapReduce job. This CBMM (Cache-Based Matrix-Matrix multiplication) algorithm has smaller disk and network costs:

- Disk read: $|A| + |B| + T$.
- Disk write: $|AB| + T$.
- Network transfer (shuffle stage): T.

The effect of this smaller disk accesses and network transfers is that the running time of CBMM is about 76× smaller than the naive algorithm [22]. As in the GIM-V, HEigen also enjoys linear running time on the number of machines and edges, since the major operation is the matrix-vector multiplication.

5. DISCOVERIES ON BIG GRAPHS

In this section, we present interesting discoveries on large, real world graphs, that we found using Pegasus. We mainly focus on the following two data.

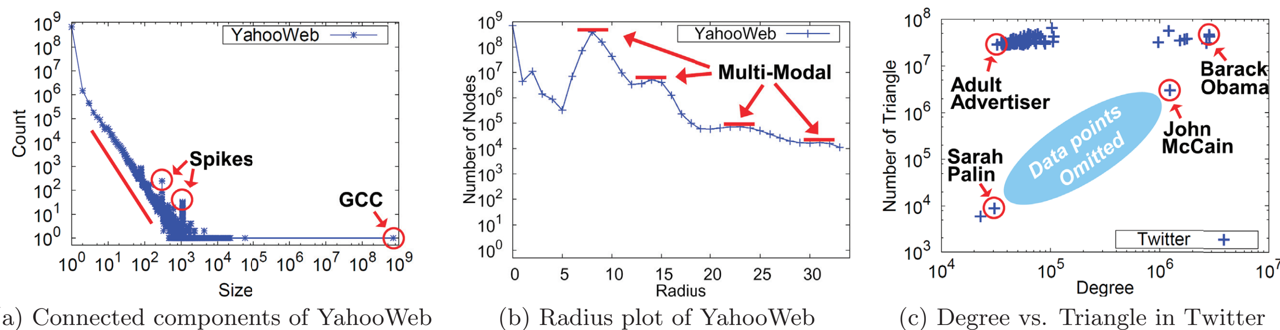
- YahooWeb: a snapshot of the World Wide Web at year 2002, crawled by Yahoo!. It contains 1.4 billion Web pages and 6.6 billion links. The edge list size is 116 gigabytes.
- Twitter: the Twitter 'who-follows-whom' graph at November 2009. It contains 63 million people and 1.8 billion connections. The edge list size is 24.2 gigabytes.

The experiments were performed in Yahoo!'s M45 Hadoop cluster (now OCC-Y [4]), one of the largest Hadoop clusters available to academia with 480 machines, 1.5 petabyte storage and 3.5 Terabyte memory in total.

The discoveries include the patterns and anomalies in connected components, diameter/radius, near cliques, and triangles.

5.1 Connected Components

What are the patterns and anomalies in the connected components of real world graphs? We report interesting discoveries in the connected component of real world graphs.



(a) Connected components of YahooWeb (b) Radius plot of YahooWeb (c) Degree vs. Triangle in Twitter

Figure 1: Discoveries in large, real world graphs. The plots are adopted from [26; 27; 22]. (a) Connected components size distribution of the YahooWeb graph. In addition to the GCC (Giant Connected Component) in the right side, there are many small disconnected components in the left side, whose sizes follows the power law (marked by red line). The two red circles (marked 'Spikes') deviating from the power law line comes from replication of Web sites. (b) Radius plot of the YahooWeb graph. Notice the multiple modes, marked by horizontal red lines, which possibly come from the loosely connected communities in the Web. (c) The degree vs. triangles in the Twitter who-follows-whom social network. Some famous U.S. politicians have mildly connected followers, while adult advertisers have tightly connected followers, creating many triangles. The reason is that adult accounts are often from the same provider, and the accounts follow each other to boost their rankings, thereby creating cliques containing triangles.

Power Laws in Component Sizes. In real world graphs, the giant connected component (GCC) contains the majority of the nodes. However, most graphs also contain many small disconnected components which are separated from the GCC. Moreover, the sizes of these small disconnected components follow the power law [37], as we see in the red line in Figure 1 (a), implying that 'rich-gets-richer' phenomenon [13] happens in the connected component formation process. Another interesting observation is that the slope of the fitting line of the small disconnected components sizes is constant over time [26].

Anomalous Connected Components. The power law pattern in connected component sizes can be used to detect anomalous connected components which deviate significantly from the pattern. The size distribution of connected components in YahooWeb graph, shown in Figure 1 (a), contains two outstanding spikes (circled in red) deviating significantly from the patterns in neighbors. These spikes come from anomalous activities. In the first spike (component size 300), more than half of the components are isomorphic, and they were made from a domain selling company where each component represents a domain in sale. The spike happened because the domain selling company *replicated* sites using a template; this process of creating web pages is different from the typical evolution process of the Web (e.g. preferential attachment [7]). In the second spike (component size 1101), more than 80 % of the components are adult sites disconnected from the GCC, and they are generated from a template, as in the first spike. As we see in both of the spikes, the distribution plot of connected component sizes reveals interesting communities with special purposes which are disconnected from the rest of the Internet.

5.2 Diameter and Radius

What do real world graphs look like? We present discoveries on the diameter and the radius in real world graphs.

Small Web. How close are the two randomly sampled nodes in a graph? The study of diameter in real networks have attracted many attentions from researchers in diverse backgrounds including social science [31], physics [5], and computer science [31]. With Pegasus, we analyzed the diameter of real world graphs. It turns out many real world graphs have small diameter (e.g. YahooWeb's diameter is 7.62) [27].

Structure of Large Networks. What is the structure of large, real world graphs? What do they look like? The analysis of the radius, combined with the analysis of the connected components, reveals that many real world graphs are composed of three areas [27]. The first area is the 'core' which contains nodes belonging to the giant connected component (GCC) of the graph, and having small radii. Majority of high degree nodes belong to the 'core' area. The second area is the 'whiskers' which contains nodes still belonging to GCC, but have large radii and loosely connected to the 'core'. The third area is the 'outsiders' which contains nodes belonging to non-GCC. Nodes in 'outsiders' typically have small radii, since the sizes of disconnected components are small in many real world graphs.

Multi Modality of Web Graph. The three areas 'core', 'whiskers', and 'outsiders' create bi-modal structures [27] in the radius plot (distribution plot of radius) of many real world graphs. That is, it creates a mode in the beginning (radius 1), and another mode right after the dip near the 'core' area. However, the radius plot of the YahooWeb graph shows an interesting pattern: it has a multi-modal structure, creating many modes, as shown in Figure 1 (b). Our conjecture is that they possibly come from the characteristic of the Web graph which contains several communities (e.g. English-speaking community, Spanish-speaking community, etc.) which are loosely connected to each other.

Expansion and Contraction. How do graphs evolve over time? We studied the radius plots over time to answer the question. Contrary to the intuition, the radius plot does not change in a monotonic way. In the early stage of the evolution, the radius plot expands to the right; however, after a point we call the 'gelling point', the radius plot shrinks to the left [27]. This 'expansion-contraction' pattern happens because many disconnected components in the early stage

are joined together to form the strong 'core' after the gelling point.

5.3 Spectral Analysis

What are the patterns and anomalies from the spectral analysis of real world graphs? We present interesting discoveries on the tightly connected communities and anomalous nodes.

Near-Clique Detection. The spectral analysis of graphs can reveal near cliques, or tightly connected nodes. For this task we analyze the eigenspoke [42] pattern in real world graphs. Eigenspoke is a set of clear lines in the *EE-plot* which is the scatter plot of the two eigenvectors of the adjacency matrix of the graph. Many real world graphs have clear eigenspokes; moreover, nodes having high scores in the EE-plot often forms near-cliques or bipartite cores [22].

Anomalous Triangles vs. Degree Ratio. Figure 1 (c) shows the degree and the number of participating triangles of accounts in the Twitter 'who follows whom' graph at year 2009 [22]. The triangles are computed from eigenvalues and eigenvectors, using the close connection between them [47]. In the figure, some U.S. politicians (Sarah Palin, John McCain, and Barack Obama) have mildly connected followers compared to their degrees, while adult accounts have extremely well connected followers, generating a lot of triangles. The reason is that an adult content provider often creates many Twitter accounts, and make them follow each other so that they look more popular. For the reason they belong to cliques, and they generate many triangles which are spotted in the triangle vs. degree plot. This analysis can be used to spot and eliminate harmful accounts such as those of adult advertisers and spammers, from social networks.

6. RESEARCH DIRECTIONS

Forecasting the future, the ever-growing sizes of graphs and diverse application needs will open many new opportunities for interesting researches in big graph mining. We list five of the important research directions.

1. **Big Graph Analysis Platform.** The first area is to re-design the big graph analytics platform to provide fast and scalable computation infrastructure. A challenge for the direction is to balance the speed and scalability. MapReduce is a disk based system, and thus it is scalable and robust while it is not optimized for speed. On the other hand, memory-based system like MPI is fast, but suffers from scalability. An ideal big data analytics platform would exploit the best of both world (disk and memory) to provide fast and scalable computations (e.g. see [49] which uses both disk and memory cache for iterative computations).
2. **Algorithm Redesign.** The second area is to transform existing serial algorithms into distributed algorithms. A challenge here is to remove dependency in serial algorithms so that the resulting distributed algorithms run in an 'embarrassingly parallel' way (e.g., see [16; 8]). Another challenge is to transform exact, inefficient algorithms into approximate, efficient algorithms. The reason is that for very big graphs, any algorithm having more than linear or near-linear running time would take prohibitively long to finish. Thus, approximation algorithms which run much faster than the exact counterparts, while providing reasonable accuracy (e.g. [27]) would be perfect fits for big graph analysis.
3. **Graph Compression.** An effective way to tackle the problem of growing graphs sizes is to use compression. Compression not only saves disk space, but it also helps running time because disks are much slower than CPU. A challenge in graph compression is to support various graph queries without uncompressing all the data; e.g., in GBASE [25] only small portion of the compressed data is uncompressed to answer graph queries. Another challenge is to design a good graph partition or clustering algorithm for real world graphs, since sparsely connected dense clusters can improve the compression rate. The problem is that real world graphs are often very hard to be partitioned [32]. A promising direction is to exploit the power law characteristic of real world graphs for clustering the edges (e.g. see [20]).
4. **Time Evolving Graphs.** Many real world graphs are evolving over time; thus efficiently and effectively mining time evolving graphs can lead to interesting discoveries that could not be observed in static graphs. A promising approach is to use tensor (multi-dimensional array) to model time evolving graphs by using the time as the 3rd dimension, and find correlations between dimensions using tensor decompositions (e.g. see [24; 29]).
5. **Visualization and Understanding of Graphs.** A graph forms a complicated object with many interaction between nodes. Visualization of graphs helps us better understand the structure and the interactions in graphs. The challenge is to effectively summarize the graphs so that users can easily understand the graphs in a screen with limited resolution.

7. CONCLUSION

In this paper we describe Pegasus, an open source graph mining library available in `http://www.cs.cmu.edu/~pegasus`, for finding patterns and anomalies in massive, real-world graphs. Pegasus provides algorithms for connected component, diameter/radius, PageRank, RWR, and spectral analysis. At the heart of Pegasus, there is a general primitive called GIM-V (Generalized iterative Matrix-Vector multiplication) which includes many graph mining algorithms as special cases. GIM-V is highly optimized, achieving 43× smaller disk space and 9.2× faster running time.

Using Pegasus, we analyze large, real world graphs. Our findings include (a) anomalous spikes in the connected component size distribution, (b) the 7-degrees of separation in a snapshot of the Web graph, and (c) anomalous advertisers in the Twitter who-follows-whom graph.

Overall, many interesting research challenges are waiting ahead of us, and we just started to address them.

Acknowledgements

Funding was provided by the U.S. ARO and DARPA under Contract Number W911NF-11-C-0088, by DTRA under contract No. HDTRA1-10-1-0120, and by ARL under Cooperative Agreement Number W911NF-09-2-0053. The views and conclusions are those of the authors and should not be interpreted as representing the official policies, of the U.S. Government, or other funding parties, and no official endorsement should be inferred. The U.S. Government is

authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

8. REFERENCES

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

[2] Hbase information. http://hbase.apache.org/.

[3] Mahout information. http://lucene.apache.org/mahout/.

[4] The open cloud consortium. http://opencloudconsortium.org/.

[5] R. Albert, H. Jeong, and A.-L. Barabasi. Diameter of the world wide web. *Nature*, (401):130–131, 1999.

[6] D. A. Bader and K. Madduri. A graph-theoretic analysis of the human protein-interaction network using multicore parallel algorithms. *Parallel Comput.*, 2008.

[7] A.-L. Barabasi and R. Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[8] J. K. Bradley, A. Kyrola, D. Bickson, and C. Guestrin. Parallel coordinate descent for l1-regularized loss minimization. In *ICML*, pages 321–328, 2011.

[9] 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.

[10] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan. On compressing social networks. In *KDD*, pages 219–228, 2009.

[11] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[12] J. W. Demmel. Applied numerical linear algebra. *SIAM*, 1997.

[13] D. A. Easley and J. M. Kleinberg. *Networks, Crowds, and Markets - Reasoning About a Highly Connected World*. Cambridge University Press, 2010.

[14] N. B. Ellison, C. Steinfield, and C. Lampe. The benefits of facebook friends: social capital and college students use of online social network sites. *Journal of Computer-Mediated Communication*, 12(4):1143–1168, 2007.

[15] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. *SIGCOMM*, pages 251–262, Aug-Sept. 1999.

[16] R. Gemulla, E. Nijkamp, P. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 69–77. ACM, 2011.

[17] A. Ghoting, R. Krishnamurthy, E. P. D. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, Y. Tian, and S. Vaithyanathan. Systemml: Declarative machine learning on mapreduce. In *ICDE*, pages 231–242, 2011.

[18] C. L. Giles, K. D. Bollacker, and S. Lawrence. Citeseer: an automatic citation indexing system. In *INTERNATIONAL CONFERENCE ON DIGITAL LIBRARIES*, pages 89–98. ACM Press, 1998.

[19] G. H. Golub and C. F. Van Loan. Matrix computations. *Johns Hopkins University Press*, 1996.

[20] U. Kang and C. Faloutsos. Beyond 'caveman communities': Hubs and spokes for graph compression and mining. In *ICDM*, 2011.

[21] U. Kang, M. McGlohon, L. Akoglu, and C. Faloutsos. Patterns on the connected components of terabyte-scale graphs. In *ICDM*, pages 875–880, 2010.

[22] U. Kang, B. Meeder, and C. Faloutsos. Spectral analysis for billion-scale graphs: Discoveries and implementation. In *PAKDD (2)*, pages 13–25, 2011.

[23] U. Kang, S. Papadimitriou, J. Sun, and H. Tong. Centralities in large networks: Algorithms and observations. In *SDM*, pages 119–130, 2011.

[24] U. Kang, E. E. Papalexakis, A. Harpale, and C. Faloutsos. Gigatensor: scaling tensor analysis up by 100 times - algorithms and discoveries. In *KDD*, pages 316–324, 2012.

[25] U. Kang, H. Tong, J. Sun, C.-Y. Lin, and C. Faloutsos. Gbase: an efficient analysis platform for large graphs. *VLDB J.*, 21(5):637–650, 2012.

[26] U. Kang, C. Tsourakakis, and C. Faloutsos. Pegasus: A peta-scale graph mining system - implementation and observations. *ICDM*, 2009.

[27] 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 2011.

[28] G. Karypis and V. Kumar. Multilevel -way hypergraph partitioning. In *DAC*, pages 343–348, 1999.

[29] T. G. Kolda and J. Sun. Scalable tensor decompositions for multi-aspect data mining. In *ICDM*, pages 363–372, 2008.

[30] C. Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *J. Res. Nat. Bur. Stand.*, 1950.

[31] J. Leskovec, J. M. Kleinberg, and C. Faloutsos. Graphs over time: densification laws, shrinking diameters and possible explanations. In *KDD*, pages 177–187, 2005.

[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.

[33] C. Liu, F. Guo, and C. Faloutsos. Bbm: bayesian browsing model from petabyte-scale data. In *KDD*, pages 537–546, 2009.

[34] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Graphlab: A new framework for parallel machine learning. In *UAI*, pages 340–349, 2010.

[35] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 5(8):716–727, 2012.

[36] G. Malewicz, M. H. Austern, A. J. C. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146, 2010.

[37] M. E. J. Newman. Power laws, pareto distributions and zipf's law. *Contemporary Physics*, (46):323–351, 2005.

[38] 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.

[39] 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.

[40] J.-Y. Pan, H.-J. Yang, C. Faloutsos, and P. Duygulu. Automatic multimedia cross-modal correlation discovery. In *KDD*, pages 653–658, 2004.

[41] S. Papadimitriou and J. Sun. Disco: Distributed co-clustering with map-reduce. *ICDM*, 2008.

[42] B. A. Prakash, M. Seshadri, A. Sridharan, S. Machiraju, and C. Faloutsos. Eigenspokes: Surprising patterns and community structure in large graphs. *PAKDD*, 2010.

[43] B. Shao, H. Wang, and Y. Li. The trinity graph engine. In *Microsoft Research*, 2012.

[44] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. S. Sarma, R. Murthy, and H. Liu. Data warehousing and analytics infrastructure at facebook. In *SIGMOD Conference*, pages 1013–1020, 2010.

[45] H. Tong, C. Faloutsos, and J.-Y. Pan. Fast random walk with restart and its applications. In *ICDM*, pages 613–622, 2006.

[46] L. N. Trefethen and D. Bau III. Numerical linear algebra. *SIAM*, 1997.

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

[48] D. Wang, D. Pedreschi, C. Song, F. Giannotti, and A.-L. Barabási. Human mobility, social ties, and link prediction. In *KDD*, pages 1100–1108, 2011.

[49] 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.