Data Mining with MAPREDUCE: Graph and Tensor Algorithms with Applications

Charalampos E. Tsourakakis

March 2010

Machine Learning Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Data Analysis Project

For my parents, Eftichios and Aliki and my sister Maria
For my companion, Maria
To the memory of grandmother Maria and grandfather Lampis

Abstract

This thesis, which serves as the Data Analysis Project, has three different aspects:

1. The Design of efficient algorithms.
2. A Solid Engineering Effort (implementation in the MAPREDUCE framework).
3. Mine the Data.

In Chapters 1,2,3 we focus on the triangle counting problem. Triangles play an important role is several data mining applications and especially in social networks. We treat the problem both from a combinatorial (Doulion, Triangle Sparsifiers) and a spectral perspective (Counting Triangles Using Projections). The former approach works on any graph under mild conditions whereas the latter are based on special spectral properties of the graph. Empirically, these special properties seem to appear frequently in social networks (and other skewed degree networks) but a deeper, theoretical understanding is currently lacking.

In Chapters 4 and 5 we present two solid engineering efforts: HADI and PEGASUS. Both contribute from an engineering and data mining pespective. We use the elegant Flajolet-Martin algorithm to estimate the diameter of a graph and its radius plot and we introduce a set of programming primitives which -to our experience- make a programmer's life easier. We apply our algorithms on the largest publicly available graph ever to be analyzed and extract several surprising patterns.

In the last two chapters the objects of focus are tensors. In Chapter 6 we introduce the "Two Heads are Better than one" method which models multidimensional timeseris as tensors and extracts correlations and patterns using a combination of wavelets and Tucker Decompositions. Finally, in Chapter 7 we introduce MACH[1] a randomized algorithm for computing Tucker decompositions. The efficiency of our method is verified on several monitoring systems.

[1]MACH stands **Ach**lioptas-**M**cSherry work to acknowledge the fact that our method extends their algorithm to the multilinear setting.

Acknowledgements

First, I would like to thank my advisor during the academic years 2007-08, 2008-09 Christos Faloutsos and my current advisors Gary Miller and Russell Schwartz for their support and guidance. Furthermore, I would like to thank David Tolliver, Don Sheehy, Hanghang Tong, Jure Leskovec, U Kang, Christos Boutsidis and all my other collaborators so far. Among them, I would like to especially thank Petros Drineas, Mihalis N. Kolountzakis, Yannis Koutis and Kostas Tsianos for all the interesting discussions we have had so far which have had an important impact on me.

I would like to thank my beloved friends here in Pittsburgh, Felipe Trevizan, Jammie Flasnick and Yannis Mallios. Furthermore, I would like to thank my "people" back in Greece, who I miss every day: my parents Eftichios and Aliki, my sister Maria, cousin Babis, uncle Simeon and Michalis and all the other member of my family, Andreas Varverakis, Dimitris Tsagkarakis, Dimitris Tsaparas, Giorgos Orfanoudakis, Lefteris Pratikakis, Tasos Parasiris, Miros Apostolakis and many others. Finally, words are not enough to express my feelings for Maria Tsiarli, my companion here in Pittsburgh.

Contents

Chapter 1

DOULION: Counting Triangles in Massive Graphs with a Coin

1.1 Introduction

Abundant data nowadays are modeled as graphs: the World Wide Web, social networks (e.g. LinkedIn, Facebook, Flickr) , P2P networks, co-authorship networks, biological networks, computer networks and physical connections just to name a few. Nowadays, due to the recent technology explosion, graphs reaching the planetary scale are available to us for analysis [102]. Triangles play an important role in complex network analysis. For example in social networks, triangles is a well studied subgraph. In particular, two prominent theories according to which triangles are generated in social networks are the homophily and the transitivity. According to the former, people tend to choose friends that are similar to themselves, which is also known as "birds of a feather flock together" [108]. and according to the latter, people who have common friends tend to become friends themselves [164].

The significance of the existence of triangles in networks motivates the definition of metrics that quantify the triangle density in a graph. Two such metrics are the clustering coefficient and the transitivity ratio [112].

Furthermore, it has been shown that triangles can play a significant role in graph mining applications as well. Recently, in [18] it was shown that triangles can be used to detect spamming activity. Eckman and Moses in [55] showed how triangles can be used to uncover the hidden thematic structure of the web. Moreover, according to [17], triangle counting can benefit the query plan optimization in databases. For the aforementioned reasons fast triangle counting algorithms are of high practical value.

In this paper we propose a simple, practical, yet effective algorithm for counting triangles in graphs. Our algorithm DOULION can be used in any graph. In our experiments we focus on real-world networks that exhibit a skewed degree distribution and in Erdős-Rényi graphs ([23]). DOULION is not a competitor of other triangle-counting algorithms. It is rather a “friend” since it can be used as a first step before applying any triangle counting algorithm, streaming or not. We verify the effectiveness of our method in a wide range of experiments on real-world networks and provide a basic mathematical analysis of our algorithm and some connections to the spectral analysis of matrices.

In figure 1.1 we see the results of running DOULION on one snapshot of the Wikipedia Web graph. As we see, even when keeping 10% of the edges accuracy is almost the ideal 100%. For the range of the “edge-keeping” percentages that we used, 10% to 90% with a step of 10% we received speedups 113.1, 28.9, 12.8, 7.1, 4.5, 3.1 2.2, 1.6, 1.3 correspondingly. The mean accuracy is 99.7% and the standard deviation 0.0023. DOULION has the advantage of being “embarassingly” parallel as well, therefore allowing us to easily implement it in any parallel programming framework. For our purposes, we used HADOOP the open source implementation of MAPREDUCE [45].

The outline of the chapter is as follows: Section 1.2 presents an overview of the related work and Section 1.3 the proposed algorithm. Section 1.4 shows the experimental results and we conclude in section 1.5.

1.2 Background and Related Work

In this section, we present the related work on the problem of counting triangles in a graph and briefly give some information on the MAPREDUCE framework and HADOOP.

1.2.1 Triangle Counting algorithms

Let $G(V, E)$, n=$|V|$, m=$|E|$ be an undirected, unweighted, simple graph. A triangle is a three-node subgraph of G which is fully connected.

Exact Counting Algorithms One obvious way to count the number of triangles in a graph is to enumerate all possible $\binom{n}{3}$ combinations of vertices and count how

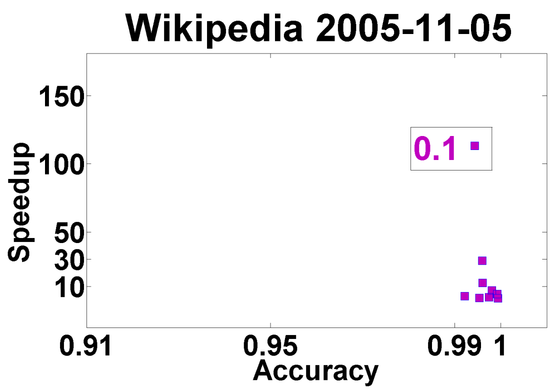


Figure 1.1: Speedup vs. Accuracy for the Wikipedia Graph snapshot on 2005 Nov. The graph has $\approx$ 1,7M nodes and 20M edges. As we see, even when keeping 10% of the edges of the initial graph accuracy is 99.5%. For p's ranging from 10% to 90% the mean accuracy is 99.7%, the accuracy standard deviation 0.0023 and the mean speedup 19.4.

many of them are fully connected. This results in the naive algorithm with $O(n^3)$ time complexity.

A simple algorithm, known as NODEITERATOR, computes for each node its neighborhood and then sees how many edges exist among its neighbors. This algorithm runs asymptotically in $\sum_{v \in V(G)} \binom{d(v)}{2}$ time which by taking a simple union bound give an upper bound of $O(d_{max}^2 n)$, where d_{max} is the maximum degree in G. Another simple algorithm that works in a similar way is the EDGEITERATOR. Rather than checking each node at the time, EDGEITERATOR checks each edge $(u, v) \in E$ and computes the common neighbors of the nodes u and v. Asymptotically EDGEITERATOR runs in the the same time with the NODEITERATOR. This algorithm can be improved through a simple hashing argument so that it runs in $O(m^{\frac{3}{2}})$ [132]. This version of EDGEITERATOR is also called EDGEITERATOR-hashed. The $forward$ algorithm is another refinement of the EDGEITERATOR. The key idea of this algorithm is that there is no need to compare the full neighborhoods of two adjacent nodes. Finally the $compact - forward$ iterator ([98]) further improves the $forward$ algorithm. Itai and Rodeh in [77] gave an algorithm that finds a triangle if it exists in $O(m^{\frac{3}{2}})$. Their algorithm can easily be extended in a triangle counting algorithm with the same time complexity. Their algorithm relies on computing spanning trees of the graph G and removing edges while making sure that each triangle is listed exactly once. In [132] one can find the analysis and an extensive description of these algorithms.

The fastest methods for triangle counting in terms of time complexity are based on fast matrix multiplication. Alon et al. gave in [13] an algorithm of time complexity $O(m^{\frac{2\gamma}{\gamma+1}}) \subset O(m^{1.41})$ where at the time of this write-up γ is 2.37, the exponent of the state-of-the-art algorithm for matrix multiplication ([38]). Exact counting methods however may be slow, even not applicable when the size of the graph fairly large due to high memory requirements. In those cases an approximating algorithm is preferred in the cost of losing the exact number of triangles.

Streaming Algorithms The goal of streaming algorithms is to perform one or at most a constant number of passes over the graph stream (e.g. edges arriving one at a time $\{e_1, e_2, .., e_m\}$) and make provably accurate estimates of the number of triangles. Yossef et al. in their seminal paper [17] gave the first streaming algorithm for counting triangles. They first define all possible different triples that can show up and then reduce the problem of triangle counting to estimating moments for a stream of node triples. Then they use the Alon-Matias-Szegedy algorithms (also known as AMS algorithms) presented in the Gödel awarded work [11]. The space complexity of their algorithms depend on the structure of the graph, and specifically on the cardinalities of the sets of the different types of triples. In [79] three streaming algorithms were presented. Two of them use one pass over the graph stream and the third one three passes. The one-pass algorithms use again the AMS algorithms [11] and the later algorithm uses sampling to reduce the usage of space. The biased sampling is done according to the degree of the vertex chosen. In [26] two random sampling algorithms are proposed to estimate the number of triangles one for the edge stream representation and one for the incidence stream representation of the undirected graph of interest. The sampling procedures are simple. E.g., for the case of the edge stream representation, they sample randomly an edge and a node in the stream and check if they form a triangle.

Semi-Streaming Algorithms Bechetti et al. presented in [18] a semi-streaming algorithm for computing triangles in a graph. Their model relaxes the strict constraint of constant number of passes to obtain an algorithm that performs $log(n)$ passes over the edge file. Their main idea relies on locality sensitivity hashing and the observation that the local triangle counting reduces to estimating the size of the intersection of two sets, namely the neighborhoods of two nodes connected by an edge. In [148] a spectral counting algorithm was introduced. The idea of this algorithm is to take advantage of the properties of the skewed spectra of power-law networks and make a fast approximation of the number of triangles based on a few, top eigenvalues. This algorithm can be viewed both as a semi-streaming algorithm in the sense that it performs a number of passes at worst $O(log(n))$ ([68]) over the non-zero elements of the adjacency matrix (edges) or even as a streaming algorithm by using a linear time algorithm for the SVD ([128]). The performance of the algorithm depends strongly on the spectrum of the graph of interest. Empirically the algorithm works well in many real-world graphs but has no guarantees, mainly due to the limited knowledge on the spectra of real-world graphs. We rather have theoretical knowledge on the few top eigenvalues ([109],[36]) or our knowledge is just empirical ([148],[59]).

1.2.2 MAPREDUCE

MAPREDUCE is a parallel distributed programming framework introduced in [46], which can process huge amounts of data in a massively parallel way using simple commodity machines. It is inspired by the functional programmming concepts of mapping and reducing. HADOOP - rougly speaking - is the open source implementation of MAPREDUCE. It is an emerging technology, which except its reportedly spread-out commercial use, that has already become popular in academia as well. HADOOP provides a powerful programming framework, since the programming concepts are simple and the programmer is freed from all the tedious tasks that one should take care of if he/she would write a distributed piece of code. More details about MAPREDUCE and HADOOP can be found in [97].

1.3 Proposed Method

In this section we present the proposed method, we analyze it and provide the reader with several interesting -at least in our opinion- observations.

1.3.1 Algorithm

Our algorithm DOULION is a “friend” rather than a competitor of the other triangle counting algorithms. Furthermore, it is very useful and applicable in all possible scenarios: a) the graph fits in the main memory, b) the size of the graph exceeds slightly the available memory, c) the size of the graph exceeds the available memory significantly.

Algorithm 1 The DOULION counting framework

```
Require: Unweighted Graph $G(V, E)$
Require: Sparsification parameter $p$
Output: $\Delta'(G)$ global triangle estimation
  for each edge $e_j$ do
    Toss a biased coin with success probability $p$
    if $success$ then
      $w(e_j) \leftarrow \frac{1}{p}$
    else
      $w(e_j) \leftarrow 0$
    end if
  end for
  $\Delta'(G) \leftarrow$ TRIANGLECOUNTINGALGORITHM(G)
  return $\Delta'(G)$
```

The general framework of the proposed method is shown in algorithm 1. DOULION tosses a coin for each edge. It keeps the edge with probability p and with probability $1 - p$ it deletes it. Then each triangle in the resulting graph G' counts as $\frac{1}{p^3}$ triangles. An equivalent way of viewing this procedure is the following:

- Reweight an edge if the edge "survives" with weight equal to $\frac{1}{p}$

- Count each triangle as the product of the weights of the edges comprising the triangle. Since the initial graph G is unweighted each triangle is counted as $(\frac{1}{p})^3 = \frac{1}{p^3}$.

After the tossing-coin stage, *any* triangle counting algorithm can be applied to the obtained graph G'. Algorithm 2 shows the instantiation of the DOULION triangle counting framework using the NODEITERATOR as the triangle counting black box, which was described in section 1.2. However, in case that even after the sparsification the resulting graph cannot fit into the main memory, a streaming or a semi-streaming algorithm should be preferred instead as the black box.

Observe that since we assume that the input graph G is unweighted all edges in G' will have the same weight. Therefore we can still store efficiently G' just as if it were unweighted plus the parameter p.

Algorithm 2 The DOULION-NODEITERATOR algorithm

```
Require: Unweighted Graph $G(V, E)$
Require: Sparsification parameter $p$
Output: $\Delta'(G)$ global triangle estimation
  $\Delta'(G) \leftarrow 0$
  for each edge $e_j$ do
    Toss a biased coin with success probability $p$
    if $success$ then
      $w(e_j) \leftarrow \frac{1}{p}$
    else
      $w(e_j) \leftarrow 0$
    end if
  end for
  for $v \in V(G)$ do
    for all pairs of neighbors $(u, w)$ of $v$ do
      if $(u, w) \in E(G)$ then
        if $u < v < w$ then
          $\Delta'(G) \leftarrow \Delta'(G) + 1$
        end if
      end if
    end for
  end for
  $\Delta'(G) \leftarrow \Delta'(G) * \frac{1}{p^3}$
  return $\Delta'(G)$
```

1.3.2 Analysis of DOULION

Mean and Variance

We first show that the expected number of triangles in G' is the number of triangles Δ in the initial graph G. For each triangle in the initial graph, we attach an indicator variable δ_i, $i = 1..\Delta$. Therefore $\delta_i = 1$ if the i-th triangle[1] exists in G', otherwise $\delta_i = 0$. Let X be the random variable that denotes DOULION 's triangles' estimate.

Theorem 1 (DOULION Expected Value) *The expected number of triangles in G'*

[1]There is no ordering of triangles. Just imaging that the term i-th refers to the i-th triangle of any random ordering of the triangles in graph G.

is equal to the actual number of triangles in G: E[X]=Δ

Proof 1 *We have that the random variable X is the sum of the indicator variables multiplied by $\frac{1}{p^3}$. By simple properties of the expectation we get the following:*
E[X]=E[$\sum_{i=1}^{\Delta} \frac{1}{p^3}\delta_i$*]=*
$\sum_{i=1}^{\Delta}$*E[*$\frac{1}{p^3}\delta_i$*]=* $\frac{1}{p^3}\sum_{i=1}^{\Delta}E[\delta_i]$*=*$\frac{1}{p^3}\sum_{i=1}^{\Delta}p^3$*=* Δ

Theorem 2 (DOULION Variance) *Let Δ be the total number of triangles in G. The variance is equal to:*

$$Var(X) = \frac{\Delta(p^3-p^6)+2k(p^5-p^6)}{p^6}$$

where k is the number of pairs of triangles that are not edge disjoint.

Proof 2 *We have that our estimate is a sum of identically distributed but not independently random indicator variables of whether a triangle in the initial graph "survives". The reason that the indicator variables are not independent is shown in figure 1.3.2. The indicator variables δ_i and δ_j for the i-th and j-th triangle are not independent because when the edge that they share does not "survive" then both of them become 0. On the other hand the indicator variables δ_k and δ_p are independent.*

Now, by the definition of the variance of a random variable and its basic properties:

$$Var(X) = Var(\frac{1}{p^3}\sum_{i=1}^{\Delta}\delta_i) = \frac{1}{p^6}\sum_{i=1}^{\Delta}\sum_{j=1}^{\Delta}Cov(\delta_i,\delta_j) \tag{1.1}$$

Now we break up the above summation. There are Δ^2 terms in this sum. Δ of them are the variances of the indicator variables, therefore we get $\Delta(p^3-p^6)$. The rest $2\binom{\Delta}{2}$ terms correspond to the pairs of indicator variables. Let k out of $\binom{\Delta}{2}$ pairs of indicator variables correspond to triangles that share one edge. In that case $Cov(\delta_i,\delta_j) = p^5-p^6$. For the rest $\binom{\Delta}{2}-k$, terms $Cov(\delta_p,\delta_q) = p^6-p^6 = 0$.

Therefore, we get:

$$Var(X) = \frac{1}{p^6}\left(\Delta(p^3-p^6) + 2k(p^5-p^6)\right) \tag{1.2}$$

Using the second moment method ([10]) we get the following theorem.

Theorem 3 $Pr(|X-\Delta| \geq \epsilon\Delta) \leq \frac{(p^3-p^6)}{p^6\epsilon^2\Delta} + 2k\frac{(p^5-p^6)}{p^6\epsilon^2\Delta^2}$

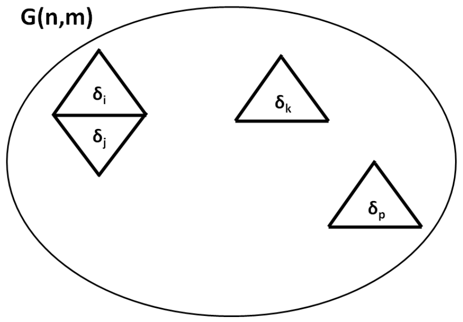


Figure 1.2: The cases should be considered when estimating the variance of DOULION. These are determined by whether the triangles are edge-disjoint or not.

Proof 3 *By applying Chebyshev's inequality, we get:* $Pr(|X - \Delta| \geq \epsilon\Delta) \leq \frac{Var(X)}{\epsilon^2\Delta^2}$ *and by substituting the formula for the variance from theorem 2 we get the bound.*

This theorem gives a first insight in the performance of DOULION. The probability that our estimate is away from the real number of triangles by some factor ϵ depends on the number of triangles in the graph as well as the structure of the graph and of course on the sparsification value p in the following way: the larger the number of triangles in the graph, the probability to obtain a good estimate increases. Also, the more edge-disjoint triangles exist in the graph, the better the estimate is. Finally, as $p \rightarrow 0$ the quality of the estimate gets worse, as expected.

Speedup

Consider now a simple triangle listing algorithm, namely the node iterator which was described in Section 1.2. If R is its running time after the removal of edges then $R = \sum_{v=1}^{k} D(v)^2$ where $D(v) =$ degree of vertex v after coin-tossing, hence

$$E[R] \sim p^2 \left[\sum_v (\deg v)^2\right]. \tag{1.3}$$

Hence the expected speedup is $\frac{1}{p^2}$.

1.3.3 Random Sampling

Let's consider the interesting case of a graph that is so large that exceeds the available main memory significantly. A well-known technique to select k random records sequentially from a file that resides in a hard disk is the rejection method [159]. More sampling algorithms can be found in the same work [159] and in [88]. Observe that the number of disk pages fetched may in the worst case be equivalent to performing a sequential scan over the file. However, if k is significantly smaller than the size of the file then we expect to have significant savings with the sampling approach. In our case, where we assume that the graph is represented as a stream of edges or equivalently resides in an edge file, e.g. a file whose each line is of the form (endpoint$_1$, endpoint$_2$), $k \approx mp$.

1.3.4 A Pleasant Side-effect: Preserving the Epidemic Threshold

As shown in [148] the number of triangles is equal to the sum of the cubes of the eigenvalues divided by six. Given the spectra properties observed in many real-world networks one can approximate the number of triangles in the graph just by using few eigenvalues. Achlioptas and McSherry showed in [6] that one can "throw " away many of the elements of a matrix and still keep the top eigenvalues the same. This is an observation that lead in [149], an improvement of the algorithm presented in [148].

Given the aforementioned observation, the top adjacency eigenvalue of G' will be very close to the top one of G. This is an interesting approach since the top eigenvalue of the adjacency matrix representation of any graph is closely related to the epidemic threshold [163]. Therefore, DOULION has the effect of not only preserving in expectation the number of triangles but also approximately the epidemic threshold.

Just for the sake of illustration, figure 1.3.4 plots the real epidemic threshold of graph G vs. the estimate, i.e. the epidemic threshold of graph G' for 14 different datasets (Flickr, Epinions, AS Newman, EAT RS, Lederberg, Patents (main), Patents, Internet, HEP-th (new), Journals, AS Oregon, AS CAIDA (3 timestamps)). As we see from the plot, the results are almost ideal, differing in the first or second decimal digit.

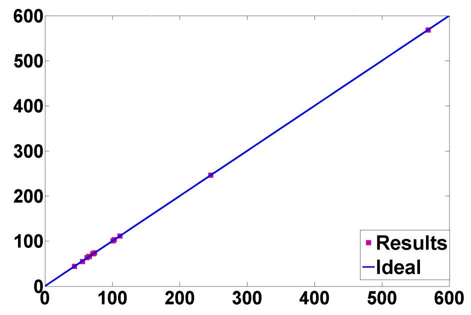


Figure 1.3: Real Inverse Epidemic Threshold (λ_1) vs. our estimate for 14 different datasets . As we see, the estimates are almost ideal, in most cases differing in the second decimal digit. Similar results hold for other graphs as well.

1.3.5 Can we parallelize DOULION?

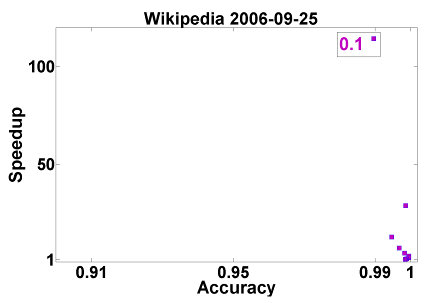
We implemented in HADOOP a prototype for the DOULION-NODEITERATOR. As one can easily observe, the sparsification step is trivially parallel. Each mapper receives a subset of edges of the initial graph and tosses a coin for each edge. If the edge survives, the mapper emits the corresponding edge. The JAVA and HADOOP code of our implementations will be open-sourced. [2]

1.4 Experiments

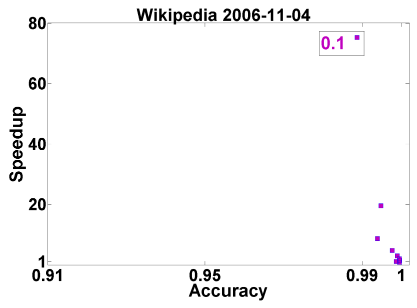
1.4.1 Experimental Setup

We implemented DOULION-NODEITERATOR in JAVA and in
HADOOP. The HADOOP code ran on Erdős-Rényi graphs and on the real-world networks we ran the JAVA piece of code. The experiments ran on a 4GB RAM, Intel(R) Core(TM)2 Duo CPU at 2.4GHz Windows Vista machine (JAVA code) and in M45 (HADOOP code), one of the fifty most powerful supercomputers in the world (480 hosts, each with 2 quad-core Intel Xeon 1.86 GHz, running RHEL5, with 3Tb aggregate RAM, and over 1.5 PetaByte aggregate disk capacity.) after allocating two commodity machines. The graphs we used in our experiments are described in the table 1.1. The directed ones were made undirected by removing

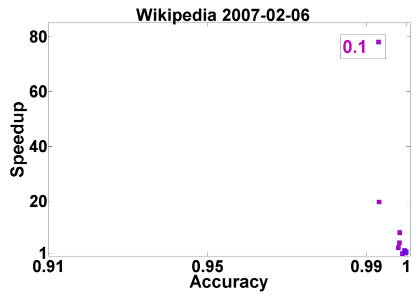
[2] http://www.cs.cmu.edu/ ctsourak/projects/triangles.htm.



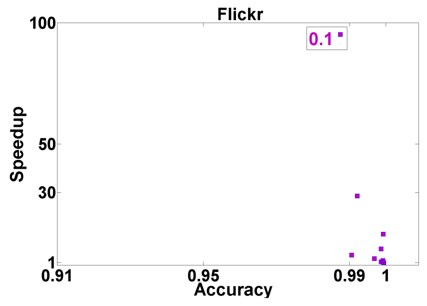
(a) **Wikipedia 2006, 25 Sep.**



(b) **Wikipedia 2006, 4 Nov.**



(c) **Wikipedia 2007, Feb. 6**



(d) **Flickr**

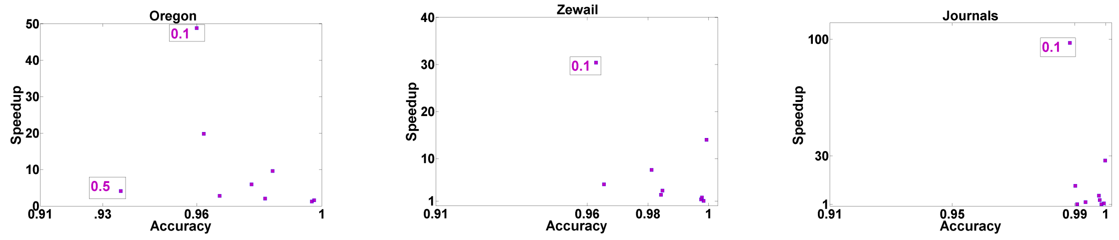
Figure 1.4: Ideal behavior of DOULIONin graphs with several million of edges. We observe that for all p values ranging from 0.1 to 0.9 the estimate of DOULION is strongly concentrated around its mean value, i.e. the real number of triangles in the graph. The speedups are important, ranging from ≈ 80 to ≈ 130.

the arcs of the edges and the self-loops -if any- were removed. Most of the datasets we used are publicly available[3].

1.4.2 Experimental Results

We divide and present the experiments into four different categories: DOULION on large-, medium- and small-sized real world graphs and on Erdős-Rényi. We run DOULION-NODEITERATOR using nine different values for p, ranging from 0.1 to 0.9 with a step of 0.1. All the figures presented in the following refer to a single, random run of DOULION on the graphs.

[3] Can be found in the url: http://www.cise.ufl.edu/research/sparse/matrices/



(a) **Oregon** (b) **Zewail** (c) **Journals**

Figure 1.5: Results of DOULION on the smallest graphs (less than 40K edges) for one random run of DOULION. Again, we observe an excellent performance of DOULION. Compared to the results for the larger graphs, the variance is bigger for the small values of p, though still small. Speedup can be even ≈ 100 (Journals).

Large-sized Graphs Figures 1.1 and 1.4 show the experimental results for the largest real-world graphs we used: the four different snapshots of the Wikipedia Web graph and Flickr. All these networks have size greater than 2M edges. The behavior of DOULION in these graphs is the ideal. The accuracy is always greater than 99% and speedups are significant, ranging from ≈ 80 to ≈ 130 times faster. As expected, the maximum speedups are obtained for $p = 0.1$. Also observe how more significant the speedups become when moving from $p = 0.2$ to $p = 0.1$, due to the quadratic speedup. As already mentioned before, observe that the speedup refers to the running time of a straight-forward exact triangle counting method vs. *itself* using DOULION, i.e. NODEITERATOR VS. DOULION-NODEITERATOR. This verifies the fact DOULION is a friend of triangle counting algorithms.

Medium-sized Graphs We conducted 158 experiments on medium-sized graphs, whose sized ranged from ≈ 40K to ≈ 400K edges. Figure 1.6 shows the performance of DOULION on these graphs. For the 150 omitted timestamps/graphs of AS CAIDA, similar results hold as in figure 1.6(h).

Edinburgh Thesaurus and AS Newman graphs (figures 1.6(c),(g) exhibit the almost ideal behavior of the large graphs: accuracy always greater than 99% and important speedups. Very close to this behavior, is also behavior of the Epinions (who-trusts-whom), the Reuters' graph and the HEP-TH graph, shown in figures 1.6(a), (b) and (e). Speedups are still important and accuracy is again high, always more than 97%. In the rest of the graphs (figures 1.6(d),(f),(h)) results are still satisfactory. However we observe that there is larger variance around the real

Nodes	Edges	Description
Real-world Networks		
13,579	37,448	AS Oregon
23,389	47,448	CAIDA AS 2004 to 2008 (means over 151 timestamps)
22,963	48,436	AS NEWMAN
1,634,989	18,540,603	Wikipedia 2005-11-05
2,983,494	35,048,116	Wikipedia 2006-09-25
3,148,440	37,043,458	Wikipedia 2006-11-04
3,566,907	42,375,912	Wikipedia 2007-02-06
27,770	352,285	Hep-th-new
27,240	341,923	Hep-th
8,843	41,532	Lederberg
124	5,972	Journals
13,332	148,038	Reuters
23,219	304,937	Edinburgh Associative Thesaurus (EAT RS)
75,877	405,740	Epinions network
404,733	2,110,078	Flickr
6752	54182	Zewail

Table 1.1: Summary of real-world networks used.

number of triangles in the graph. Still though, the accuracy is always greater than 96%. The maximum speedup in the case of medium sized graphs can reach 100 times, which corresponds

Small-sized Graphs We used three small graphs to experiment with, AS Oregon, Journals and Zewail. Journals graph exhibits an ideal behavior, just like the large graphs. DOULION gives more than 99% accuracy for all values of p we tried and a speedup of almost 100 times. Oregon and Zewail exhibit larger variance than Journals graph over our single random run. Accuracy is almost always greater than 95% , with the single exception of using $p = 0.5$ in the Oregon graph. However, running DOULION three times, moves these “outlier”-like points closer to 1, just like in all other plots. This was the worst case behavior of DOULION that we saw during our experiments.

Nodes	Speedup	Accuracy
80M	13.1	99.7
100M	19.8	99.3

Table 1.2: Results of DOULION on $G_{n,\frac{1}{2}}$ for sparsification value equal to $\frac{1}{2}$.

Observations To sum up, the following observations hold for all the experiments we conducted on real graphs with size ranging from $\approx 6K$ edges (Journals graph) to $\approx 42M$ (Wikipedia 2007):

- Keeping 10% of the edges yields in the most significant speedups. These speedups ranged from ≈ 30 to ≈ 130 times.
- Notice that reducing the edges to 10% of the initial amount does not necessarily imply 10x speedup, but much more. In general, the speedups also depend on the structure of the initial and the sparsified graph.
- Running DOULION three times verifies the fact that the results we obtained were not "random": for most of the graphs the results are almost identical (speedups and accuracies are more or less the same) whereas for few graphs (Oregon and some AS CAIDA timestamps) we see slight larger changes, still though small (e.g. Oregon for p=0.5 gives 93% accuracy).

DOULION on Erdős-Rényi $G_{n,\frac{1}{2}}$ Using our HADOOP implementation we run DOULION on large Erdős-Rényi $G_{n,p}$ graphs. As expected in the case large of random Erdős-Rényi the results are excellent in terms of accuracy for the sparsification values we tested. The reason is the following: after applying DOULION to a $G_{n,p}$ graph with the sparsification parameter equal to 0.1 the result is an Erdős-Rényi $G_{n,p'}$ with $p' = 0.1p$. Therefore, as long as p' is a constant and does not cause any threshold phenomena in the number of cycles in the graph (e.g. $p' = \frac{1}{n}$, see [23]) we have a concentrated estimate around the real number of triangles. The results of running DOULION-NODEITERATOR with $p = 0.1$ on two Erdős-Rényi graphs with 80M and 100M nodes are shown in Table 1.2. As we see, the speedups are 13.1 and 19.8 respectively for the two graphs and the accuracy in both cases is greater than 99%.

1.5 Conclusions

In this paper we presented DOULION, an algorithm which tosses a coin in order to obtain a smaller, weighted graph in which the number of triangles is very close to the true value. Our contributions can be summarized in the following points:

- DOULION is a "friend" rather than a competitor to other triangle counting algorithms: any other triangle counting triangle algorithm, streaming or not, use the idea of DOULION as a preprocessing step.

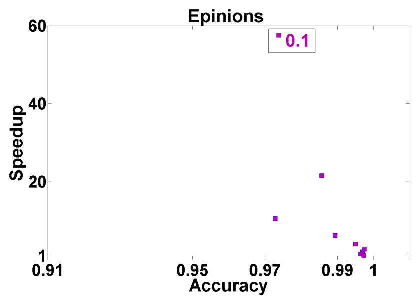
- DOULION is "embarrassingly" parallel, enjoying therefore optimal scale-up in HADOOP.

- We provide a first, basic mathematical analysis which gives some insight in the performance of DOULIONwith respect to the mean and the variance of the estimator and the expected speedup for the instatiation we used.

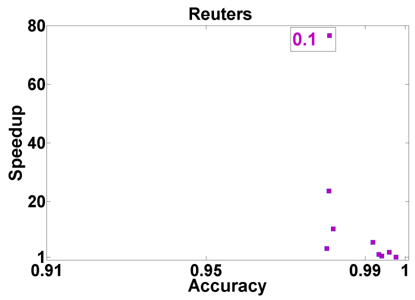
- We show that an additional benefit of DOULION is that it maintains the epidemic threshold of the graph.

- We conducted several experiments on real world graphs and for p ranging from 0.1 to 0.9 the accuracy is almost 100% and the speedup can be even $\approx$130x of a simple exact counting algorithm vs. itself but using DOULION as a first step.

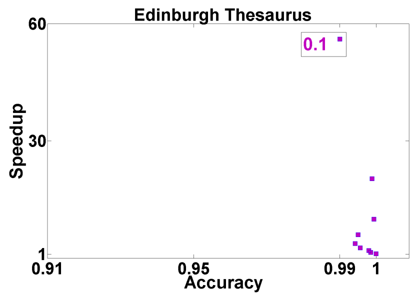
Finally, as a topic of future research, we propose a tighter theoretical analysis that will yield the optimal p, namely the smallest possible one which yields an exponential concentration around the real number of triangles.



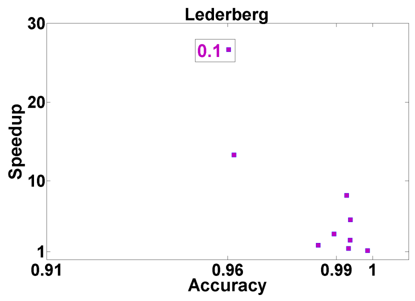
(a) **Epinions**



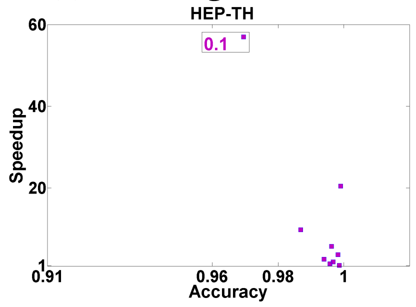
(b) **Reuters**



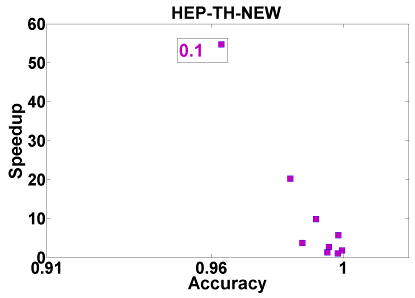
(c) **Edinburgh Thesaurus**



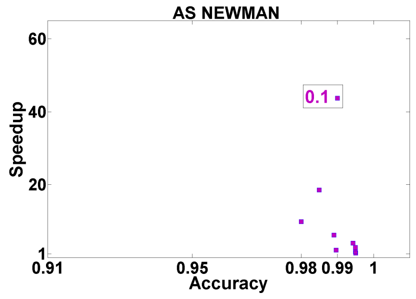
(d) **Lederberg**



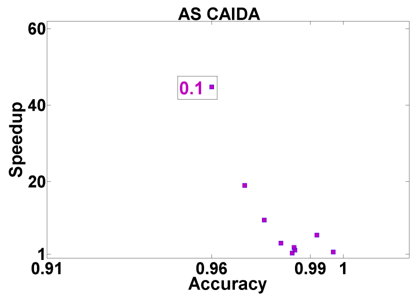
(e) **HEP-TH**



(f) **HEP-TH-NEW**



(g) **AS Newman**



(h) **AS CAIDA**

Figure 1.6: Behavior of DOULION in graphs with several medium sized networks ($\approx$ 40K to $\approx$ 400K edges). As in the case of large and small graphs, we observe that for all p values ranging from 0.1 to 0.9 the estimate of DOULION is strongly concentrated around real number of triangles in the graph. Speedups again are important, ranging from $\approx$ 30 to $\approx$ 60.

Chapter 2

Triangle Sparsifiers

2.1 Introduction

Graphs are ubiquitous: the Internet, the World Wide Web (WWW), social networks, protein interaction networks and many other complicated structures are modeled as graphs. The problem of counting subgraphs is one of the typical graph mining tasks that has attracted a lot of attention, e.g., [167]. The most basic, non-trivial subgraph, is the triangle. Given a simple, undirected graph $G(V, E)$, a triangle is a three node fully connected subgraph. Many social networks are abundant in triangles, since typically friends of friends tend to become friends themselves [164]. This phenomenon is observed in other types of networks as well (biological, online networks etc.) and is one of the main reasons which gave rise to the definitions of the transitivity ratio and the clustering coefficients of a graph in complex network analysis [112]. Triangles are used in several applications such as uncovering the hidden thematic structure of the web [55], as a feature to assist the classification of web activity [18] and for link recommendation in online social networks [153]. Furthermore, triangles are used as a network statistic in the exponential random graph model.

A sparsifier of a graph $G(V, E, w)$ is a sparse graph $G'(V, E', \hat{w})$ which is similar to G under a certain notion. For instance, [138] present algorithms for generating high-quality spectral sparsifiers and [21] introduces cut-preserving sparsifiers. In this paper, we present a simple randomized algorithm which generates high quality triangle-preserving sparsifiers for unweighted graphs under mild restrictions. We analyze our algorithm and show that we can achieve significant speedups and an accurate estimate of the number of triangles at the same time.

For instance, if one uses a listing algorithm for a graph with n nodes and t triangles, where $t \geq n^{3/2+\epsilon}$ one can set the sparsification parameter $p = n^{-1/2}$ resulting in a linear $O(n)$ expected speedup and a concentration of the estimate T around the true number of triangles t. We verify the efficiency of our method in large networks where our method results in three to four orders of magnitude speedup and excellent accuracy.

The chapter is organized as follows: Section 2.2 presents briefly the existing work and the theoretical background, Section 2.3 presents our proposed optimal sampling method and Section 2.4 presents the experimental results on several large graphs. In Section 2.5 we conclude.

2.2 Preliminaries

In this section, we briefly present the existing work on the triangle counting problem and the necessary theoretical background for our analysis.

2.2.1 Existing work

There exist two general categories of triangle counting algorithms, the exact and the approximate counting algorithms. It is worth noting that for the applications described in Section 2.1 the exact number of triangles in not crucial. Thus, approximate counting algorithms which are faster and output a high quality estimate are desirable.

Exact Counting The state of the art algorithm is due to Alon, Yuster and Zwick [13] and runs in $O(m^{\frac{2\omega}{\omega+1}})$, where currently the fast matrix multiplication exponent ω is 2.371 [38]. Thus, the Alon et al. algorithm currently runs in $O(m^{1.41})$ time. Algorithms based on matrix multiplication are not used in practice due to the high memory requirements. Even for medium sized networks, matrix-multiplication based algorithms are not applicable. In planar graphs, triangles can be found in $O(n)$ time [77, 119]. Furthermore, in [77] an algorithm which finds a triangle in any graph in $O(m^{\frac{3}{2}})$ time is proposed. This algorithm can be extended to list the triangles in the graph with the same time complexity. Even if listing algorithms solve a more general problem than the counting one, they are preferred in practice for large graphs, due to the smaller memory requirements compared to the matrix multiplication based algorithms. Simple representative algorithms are the node- and the edge-iterator algorithms. In the former, the algorithm counts for each node the number of edges among its neighbors, whereas the latter counts for each edge (i, j) the common neighbors of nodes' i, j. Both have the same asymptotic complexity $O(mn)$, which in dense graphs results in $O(n^3)$ time, the complexity of the naive counting algorithm. Practical improvements over this family of algorithms have been achieved using various techniques, such as hashing and sorting by the degree [133].

Approximate Counting Most of the approximate triangle counting algorithms have been developed in the streaming setting. In this scenario, the graph is represented as a stream. Two main representations of a graph as a stream are the edge stream and the incidence stream. In the former, edges are arriving one at a time. In the latter scenario all edges incident to the same vertex appear successively in the stream. The ordering of the vertices is assumed to be arbitrary. A streaming algorithm produces a relative ϵ approximation of the number of triangles with high probability, making a constant number of passes over the stream. However, sampling algorithms developed in the streaming literature can be applied in the setting where the graph fits in the memory as well.

Monte Carlo sampling techniques have been proposed to give a fast estimate of the number of triangles. According to such an approach, a.k.a. naive sampling [133], we choose three nodes at random repetitively and check if they form a triangle or not. If one makes

$$r = \log(\frac{1}{\delta})\frac{1}{\epsilon^2}(1 + \frac{T_0 + T_1 + T_2}{T_3})$$

independent trials where $T_i = \#$triples with i edges and outputs as the estimate of triangles the random variable $T_3' = \binom{n}{3}\frac{\sum_{i=1}^{r} X_i}{r}$ then

$$(1 - \epsilon)T_3 < T_3' < (1 + \epsilon)T_3$$

with probability at least $1 - \delta$. For graphs that have $T_3 = o(n^2)$ triangles this approach is not suitable. This is the typical case, when dealing with real-world networks.

In [17] the authors reduce the problem of triangle counting efficiently to estimating moments for a stream of node triples. Then, they use the Alon-Matias-Szegedy algorithms [11] (a.k.a. AMS algorithms) to proceed. The key is that the triangle computation reduces in estimating the zero-th, first and second frequency moments, which can be done efficiently. Again, as in the naive sampling, the denser the graph the better the approximation. The AMS algorithms are also used by [79], where simple sampling techniques are used, such as choosing an edge from the stream at random and checking how many common neighbors its two endpoints share considering the subsequent edges in the stream. Along the same lines, [27] proposed two space-bounded sampling algorithms to estimate the number of triangles. Again, the underlying sampling procedures are simple. E.g., for the case of the edge stream representation, they sample randomly an edge and a node in the stream and check if they form a triangle. Their algorithms are the state-of-the-art algorithms to the best of our knowledge. The three-pass algorithm presented therein, counts in the first pass the number of edges, in the second pass it samples uniformly at random an edge (i, j) and a node $k \in V - \{i, j\}$ and in the third pass it tests whether the edges $(i, k), (k, j)$ are present in the stream. The number of draws that have to be done in order to get concentration (these draws are done in parallel), is of the order

$$r = \log(\frac{1}{\delta})\frac{2}{\epsilon^2}(3 + \frac{T_1 + 2T_2}{T_3})$$

Even if the term T_0 is missing compared to the naive sampling, the graph has still to be fairly dense with respect to the number of triangles in order to get an ϵ approximation with high probability.

In the case of "power-law" networks it was shown in [**?**] that the spectral counting of triangles can be efficient due to their special spectral properties and [151] extended this idea using the randomized algorithm by [51] by proposing a simple biased node sampling. This algorithm can be viewed as a special case of a streaming algorithm, since there exist algorithms, e.g., [129], that perform a constant number of passes over the non-zero elements of the matrix to produce a good low rank matrix approximation. In [18] the semi-streaming model for counting triangles is introduced, which allows $\log n$ passes over the edges. The key observation is that since counting triangles reduces to computing the intersection of two sets, namely the induced neighborhoods of two adjacent nodes, ideas from locality sensitivity hashing are applicable to the problem.

In [154] an algorithm which tosses a coin independently for each edge with probability p to keep the edge and probability $q = 1 - p$ to throw it away is proposed. Then, one counts the number of triangles t' in G'. The estimate of the algorithm is the random variable $T = \frac{t'}{p^3}$. It was shown in [154] that the estimator T is unbiased, i.e., $\mathbb{E}[T] = t$. The authors however did not answer a critical question: how small can p be? In [154] only constant factor speedups were achieved.

2.2.2 Concentration of Boolean Polynomials

A common task in combinatorics is to show that if Y is a polynomial of independent boolean random variables then Y is concentrated around its expected value. In the following we state the necessary definitions and the main concentration result we use in our analysis.

Let $Y = Y(t_1, \ldots, t_m)$ be a polynomial of m real variables. The following definitions are from [145]. Y is totally positive if all of its coefficients are non-negative variables, regular if all of its coefficients are between zero and one, simplified if all of its monomials are square free and homogeneous if all of its monomials have the same degree. Given any multi-index $\alpha = (\alpha_1, \ldots, \alpha_m) \in \mathbb{Z}_+^m$, define the partial derivative $\partial^\alpha Y = (\frac{\partial}{\partial t_1})^{\alpha_1} \ldots (\frac{\partial}{\partial t_m})^{\alpha_m} Y(t_1, \ldots, t_m)$ and denote by $|\alpha| = \alpha_1 + \cdots \alpha_m$ the order of α. For any order $d \geq 0$, define $\mathbb{E}_d(Y) = \max_{\alpha:|\alpha|=d} \mathbb{E}(\partial^\alpha Y)$ and $\mathbb{E}_{\geq d}(Y) = \max_{d' \geq d} \mathbb{E}_{d'}(Y)$.

Typically, when Y is smooth, it is also strongly concentrated. By smoothness one means that Y has a small Lipschitz constant,i.e., when one changes the value of one variable t_j, the value Y changes no more than a constant. However, as stated in [160] this is restrictive in many cases. Thus one can demand "average smoothness" as defined in [160]. For the purposes of this work, consider a random variable $Y = Y(t_1, \ldots, t_m)$ which is a positive polynomial of m boolean variables $[t_i]_{i=1..m}$ which are independent. Observe that a boolean polynomial is always regular and simplified.

Now, we refer to the main theorem of Kim and Vu of [86, §1.2] as phrased in Theorem 1.1 of [160] or as Theorem 1.36 of [145].

Theorem 4 *There is a constant c_k depending on k such that the following holds. Let $Y(t_1, \ldots, t_m)$ be a totally positive polynomial of degree k, where t_i can have arbitrary distribution on the interval $[0, 1]$. Assume that:*

$$\mathbb{E}[Y] \geq \mathbb{E}_{\geq 1}(Y) \tag{2.1}$$

Then for any $\lambda \geq 1$:

$$\mathbb{P}|Y - \mathbb{E}[Y]| \geq c_k \lambda^k (\mathbb{E}[Y] \, \mathbb{E}_{\geq 1}(Y))^{1/2} \leq e^{-\lambda + (k-1)\log m}. \tag{2.2}$$

2.3 Proposed Method

2.3.1 Algorithm

Algorithm 3 Triangle Sparsifier

```
Require: Set of edges $E \subseteq \binom{[n]}{2}$ {Unweighted graph $G([n], E)$}
Require: Sparsification parameter $p$
  Pick a random subset $E'$ of edges such that the events $\in \mathcal{E}'$, for all $e \in E$ are
  independent and the probability of each is equal to $p$.
  $t' \leftarrow$ count triangles on the graph $G'([n], E')$
  return $T \leftarrow \frac{t'}{p^3}$
```

Our proposed algorithm *Triangle Sparsifier* is shown in Algorithm 1 (see also [154]). The algorithm takes an unweighted, simple graph $G(V, E)$, where without loss of generality we assume that the nodes are numbered from $1, \ldots, n$, i.e., $V = [n]$ and a sparsification parameter $p \in (0, 1)$ as input. The algorithm first chooses a random subset E' of the set E of edges. The random subset is such that the events

$$\{e \in E'\}, \text{for all } e \in E,$$

are independent and the probability of each is equal to p.

Then, any triangle counting algorithm can be used to count triangles on the sparsified graph with edge set E'. Clearly, the expected size of E' is pm where $m = |E|$. The output of our algorithm is the number of triangles in the sparsified graph multiplied by $\frac{1}{p^3}$, or equivalently we are counting the number of weighted triangles in G' where each edge has weight $\frac{1}{p}$.

How to choose the random set in sublinear expected time We do not "toss a p-coin" m times in order to construct E'. This would be very wasteful if p is small. Instead we construct the random set E' with the following procedure which produces the right distribution. Observe that the number X of unsuccessful events, i.e., edges which are not selected in our sample, until a successful one follows a geometric distribution. Specifically, $\mathbb{P}X = x = (1-p)^{x-1}p$. To sample from this distribution it suffices to generate a uniformly distributed variable U in $[0, 1]$ and set $X \leftarrow \left\lceil \frac{\ln U}{1-p} \right\rceil$. Clearly the probability that $X = x$ is equal to $\mathbb{P}(1-p)^{x-1} > U \geq (1-p)^x = (1-p)^{x-1} - (1-p)^x = (1-p)^{x-1}p$ as required. This provides a practical and efficient way to pick the subset E' of edges in subliner expected time $O(pm)$. For more details see [88].

2.3.2 Analysis

Our main result is the following theorem.

Theorem 5 *Suppose G is an undirected graph with n vertices, m edges and t triangles. Let also Δ denote the size of the largest collection of triangles with a common edge. Let G' be the random graph that arises from G if we keep every edge with probability p and write T for the number of triangles of G'. Suppose that $\gamma > 0$ is a constant and*

$$\frac{pt}{\Delta} \geq \log^{6+\gamma} n, \quad \textit{if } p^2\Delta \geq 1, \tag{2.3}$$

and

$$p^3 t \geq \log^{6+\gamma} n, \quad \textit{if } p^2\Delta < 1. \tag{2.4}$$

for $n \geq n_0$ sufficiently large. Then

$$\mathbb{P}|T - \mathbb{E}[T]| \geq \epsilon\mathbb{E}[T] \leq n^{-K}$$

for any constants $K, \epsilon > 0$ and all large enough n (depending on K, ϵ and n_0).

Proof 4 *Write $X_e = 1$ or 0 depending on whether the edge e of graph G survives in G'. Then $T = \sum_{\Delta(e,f,g)} X_e X_f X_g$ where $\Delta(e,f,g) = \mathbf{1}$ (edges e, f, g form a triangle). Clearly $\mathbb{E}[T] = p^3 t$.*

Refer to Theorem 4. We use T in place of Y, $k = 3$.

We have

$$\mathbb{E}\left[\frac{\partial T}{\partial X_e}\right] = \sum_{\Delta(e,f,g)} \mathbb{E}[X_f X_g] = p^2|\Delta(e)|,$$

where $\Delta(e)$ = to how many triangles edge e participates. We first estimate the quantities $\mathbb{E}_j(T), j = 0, 1, 2, 3,$ defined before Theorem 4. We get

$$\mathbb{E}_1(T) = p^2\Delta \tag{2.5}$$

where $\Delta = \max_e |\Delta(e)|$.

We also have

$$\mathbb{E}\left[\frac{\partial^2 T}{\partial X_e \partial X_f}\right] = p\mathbf{1}\left(\exists g : \Delta(e,f,g)\right),$$

hence

$$\mathbb{E}_2(T) \leq p. \tag{2.6}$$

Obviously $\mathbb{E}_3(T) \leq 1$.

Hence

$$\mathbb{E}_{\geq 3}(T) \leq 1,\ \mathbb{E}_{\geq 2}(T) \leq 1,$$

and

$$\mathbb{E}_{\geq 1}(T) \leq \max\,,\,,\ \mathbb{E}_{\geq 0}(T) \leq \max\,,\,,\,.$$

• CASE 1 *($p^2\Delta < 1$):*

We get $\mathbb{E}_{\geq 1}(T) \leq 1$*, and, from* (2.4)*,* $\mathbb{E}_{\geq 0}(T) = p^3 t$*.*

• CASE 2 *($p^2\Delta \geq 1$):*

We get $\mathbb{E}_{\geq 1}(T) \leq p^2\Delta$*, and, from* (2.3)*,* $\mathbb{E}_{\geq 0}(T) = p^3 t$*.*

We get, for some constant $c_3 > 0$*, from Theorem 4:*

$$\mathbb{P}|T - \mathbb{E}[T]| \geq c_3\lambda^3(\mathbb{E}[T]\,\mathbb{E}_{\geq 1}(T))^{1/2} \leq e^{-\lambda + 2\log n}. \tag{2.7}$$

Notice that in both cases we have $\mathbb{E}[T] \geq \mathbb{E}_{\geq 1}(T)$*.*

We now select λ *so that the lower bound inside the probability on the left-hand side of* (2.7) *becomes* $\epsilon\mathbb{E}[T]$*. In Case 1 we pick*

$$\lambda = \frac{\epsilon^{1/3}}{c_3^{1/3}}(p^3 t)^{1/6}$$

while in Case 2

$$\lambda = \frac{\epsilon^{1/3}}{c_3^{1/3}}\left(\frac{pt}{\Delta}\right)^{1/6}$$

to get

$$\mathbb{P}|T - \mathbb{E}[T]| \geq \epsilon\mathbb{E}[T] \leq \exp(-\lambda + 2\log n) \tag{2.8}$$

Since $\lambda \geq (K+2)\log n$ *follows from our assumptions* (2.3) *and* (2.4) *if* n *is sufficiently large, we get* $\mathbb{P}|T - \mathbb{E}[T]| \geq \epsilon\mathbb{E}[T] \leq n^{-K}$*, in both cases.*

Complexity Analysis The expected running time of edge sampling is sublinear, i.e., $O(pm)$. The complexity of the counting step depends on which algorithm we use to count triangles. For instance, if we use [13] as our triangle counting algorithm, the expected running time of Triangle Sparsifier is $O(pm + (pm)^{\frac{2\omega}{\omega+1}})$, where ω currently is 2.371 [38]. If we use the node-iterator (or any other standard listing triangle algorithm) the expected running time is $O(pm + p^2\sum_i d_i^2)$. The expected speedups with respect to the triangle counting task are therefore $p^{-\frac{2\omega}{\omega+1}}$, i.e., currently $p^{-1.41}$, and p^{-2} respectively.

2.3.3 Discussion

This theorem states the important result that the estimator of the number of triangles is concentrated around its expected value, which is equal to the actual number of triangles t in the graph [154] under mild conditions on the triangle density of the graph. The mildness comes from condition (2.3): picking $p = 1$, given that our graph is not triangle-free, i.e., $\Delta \geq 1$, gives that the number of triangles t in the graph has to satisfy $t \geq \Delta \log^{6+\gamma} n$. This is a mild condition on t since $\Delta \leq n$ and thus it suffices that $t \geq n \log^{6+\gamma} n$ (after all, we can always add two dummy connected nodes that connect to every other node, as in Figure 1(a), even if practically -experimentally speaking- Δ is smaller than n). The critical quantity besides the number of triangles t, is Δ. Intuitively, if the sparsification procedure throws away the common edge of many triangles, the triangles in the resulting graph may differ significantly from the original.

A significant problem is the choice of p for the sparsification. Conditions (2.3) and (2.4) tell us how small we can afford to choose p, but the quantities involved, namely t and Δ, are unknown. One way around this obstacle would be to first estimate the order of magnitude of t and Δ and then choose p a little suboptimally. It may be possible to do this by running the algorithm a small number of times and deduce concentration if the results are close to each other. If they differ significantly then we sparsify less, say we double p, and so on, until we observe stability in our results. This would increase the running time by a small logarithmic factor at most. As we will describe in Section 2.4, in practice the doubling p idea, works well.

From the theoretical point of view, this ambiguity of how to choose p to be certain of concentration in our sparsification preprocessing does not however render our result useless. Under very general assumptions on the nature of the graph one should be able to get a decent value of p. For instance, if we know $t \geq n^{3/2+\epsilon}$ and $\Delta \sim n$, we get $p = n^{-1/2}$. This will result in a linear $O(n)$ expected speedup, as already mentioned in section 2.2.

2.4 Experiments

In order to show the efficiency of our method, we perform a set of experiments on several large networks. In this section we describe first the experimental setup, and then we present the experimental results.

2.4.1 Experimental Setup

Table 2.1 provides a description of the networks we used in our experiments after the preprocessing (all graphs were first made undirected and all self-loops were removed). We implemented the node iterator algorithm which was described in Section 2.2. The code is written in JAVA and in Hadoop, the open source version of MapReduce. We used two machines to run our experiments. The experiments for the three smallest graphs (Wikipedia 2005/9, Flickr, Youtube) were executed in a 2GB RAM, Intel(R) Core(TM)2 Duo CPU at 2.4GHz Ubuntu Linux machine. For the three larger graphs (WB-EDU, Wikipedia 2006, Wikipedia 2005), we used the M45 supercomputer, one of the fifty most powerful supercomputers in the world.

Name	Nodes	Edges	Description
WB-EDU	9,845,725	46,236,105	Web Graph (page to page)
Wikipedia 2007/2	3,566,907	42,375,912	Web Graph (page to page)
Wikipedia 2006/6	2,983,494	35,048,116	Web Graph (page to page)
Wikipedia 2005/9	1,634,989	18,540,603	Web Graph (page to page)
Flickr	404,733	2,110,078	Person to Person
Youtube[110]	1,157,822	4,945,382	Person to Person

Table 2.1: Description of datasets

2.4.2 Experimental Results

Given that the order of magnitute of the number of nodes n in the majority of our graphs is 6 we begin with a sparsification value $p = 0.005$ which is of the order $1/\sqrt{n}$. We keep doubling the sparsification parameter until we deduce concentration and stop. Table 2.2 summarizes the results. In more detail, each row corresponds to the p^* value, that we first deduced concentration using the doubling procedure for each of the datasets we used (column 1). Ideally we would

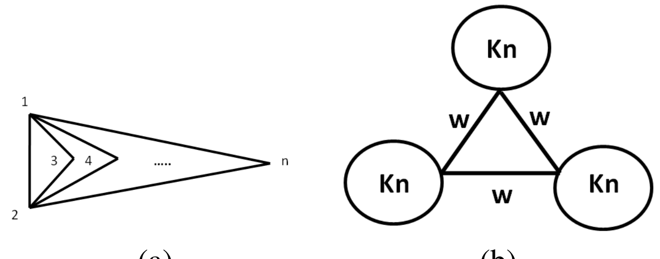


Figure 2.1: (a) Linear number of triangles. (b) Weighted graphs.

like to find p_I^*, the minimum p value for which we observe concentration, but we settle with a p^* value which is at most 2 times more than p_I^*. The third column of table 2.2 describes the quality of the estimator. Particularly, it contains values of the ratio $\frac{T}{t}$ averaged over six experiments. The next column contains the running time of the sparsification code, i.e., how much time it takes to make one pass over the edge file[1] and generate a second edge file containing the edges of the sparsified graph. The fourth column $\times$*faster 1* contains the speedup of the node iterator vs. itself when applied to the original graph and to the sparsified graph, i.e., the sample. The last column, $\times$*faster 2*, contains the speedup of the whole procedure we suggest, i.e., the doubling procedure, counting and repeat until concentration deduction, vs. running node iterator on the original graph.

Few observations concerning the experimental results are the following: a) The concentration we obtain is strong for small values of p, which implies directly large speedups. b) The speedups typically are close to the expected ones, i.e., $\frac{1}{p^2}$ for the experiments that we conducted in whole in the Ubuntu machine. For the three experiments that were conducted in M45, the speedups were larger than the expected ones due to the parallel overhead (network communication, time for the JVM (Java Virtual Machine) to load in M45 etc.) c) Even if the "doubling-and-checking for concentration" procedure may have to be repeated several times the sparsification algorithm is still of significant practical value, something witnessed by the last column of the table. d) The overall speedups shown in the last column can easily be increased if one is willing to deduce concentration with less experiments. e) Finally, when concentration is deducted, the average of the concentrated estimates is a reasonable estimator of high accuracy.

[1]A file containing the edges of the graph. Each line is of the form (i,j) representing a single edge.

G	p^*	Mean acc.	Sparsify (secs)	×*faster* 1	×*faster* 2
WB-EDU	0.005	95.8	8	70090	370.4
Wiki-2007	0.01	97.7	17	21000	332
Wiki-2006	0.02	94.9	14	4000	190.47
Wiki-2005	0.02	96.8	8.6	2812	172.1
Flickr	0.01	94.7	1.2	12799	45
Youtube	0.02	95.7	2.3	2769	56

Table 2.2: Experimental results. Observe how small can p be (second column), resulting in huge savings during the triangle counting time. The "doubling-and-checking" procedure to deduce concentration that one would employ in practice gives important speedups (fifth column) and high accuracy (third column) at the same time (last column). The drop-off in the total speedup is dominated by the sparsification time rather than the triangle counting time.

2.5 Conclusions & Future Work

In this paper we present a randomized algorithm which generates for graphs with sufficiently many triangles a high-quality triangle sparsifier graph. The theoretical speedups are significant, e.g., $O(n)$ for graphs with $t \geq n^{\frac{3}{2}+\epsilon}$, a fact which is also validated on several large networks.

One may ask how the algorithm performs in graphs where the number of triangles is linear, i.e., $O(n)$. Consider the graph of Figure 1(a). If the coin decides that edge $(1, 2)$ should be removed then our estimator is 0, making the sparsification procedure unsuitable for such graphs. Consider now the case of weighted graphs, where our algorithm can naturally be extended by changing the weight w of a "survivor"-edge to w/p. Figure 1(b) shows a weighted graph, where for w large enough, the removal of one of the weighted edges will introduce a large error in our estimate. Both cases require a sophisticated sampling procedure (e.g., [138, 21]), and are topics of future work.

Chapter 3

Counting Triangles in Real-World Networks using Projections

3.1 Introduction

Finding patterns in large scale graphs, with millions and billions of edges is attracting increasing interest with numerous applications in computer network security (e.g., intrusion detection, spamming), in web applications (e.g., community detection, blog analysis), in social networks such as Facebook and LinkedIn (e.g., for link prediction) and many more. One of the operations of interest in such a setting is the estimation of the clustering coefficients and the transitivity ratio of the graph, which effectively translates in computing the number of triangles that each node participates in or the total number of triangles in the graph respectively. Furthermore, triangles are a frequently used network statistic in the exponential random graph model and naturally appear in models of real-world network evolution [101]. Furthermore, triangles have been used in several applications such as spam detection [18], uncovering the hidden thematic structure of the web [55] and for link recommendation in online social networks [153]. It is worth noting that in social networks triangles have a natural interpretation: friends of friends are frequently friends themselves [164].

However, triangle counting is computationally expensive. In this chapter, we propose the EIGENTRIANGLE and EIGENTRIANGLELOCAL algorithms to compute the total number of triangles and the number of triangles that each node participates in respectively, in an undirected graph. Our algorithms work for any type of graph but they are effective when the graph possesses certain spectral properties. Real-world networks empirically exhibit such properties, making our algorithms a viable option for counting triangles therein. We verify this claim experimentally, by performing 160 experiments on different types of real-world networks (Web Graphs, social, co-authorship, information and Internet networks). We observe significant speedups, i.e., between 34$\times$ to 1075$\times$ faster performance, for accuracy at least 95% compared to a straight-forward counting algorithm.

We use Lanczos method to compute the low rank eigendecomposition, and we explain how the spectral properties of real-world networks allow Lanczos to converge fast. Viewing the adjacency representation of the graph as a set of n points in the n-dimensional Euclidean space $\mathbb{R}^n$ and observing that EIGENTRIANGLE performs an optimal (in the least squares sense) projection on a k-dimensional hyperplane, we show that at the cost of some accuracy fast SVD algorithms can be used instead to estimate the number of triangles. Finally we give two new laws related to triangles and a theorem providing a closed formula for the number of triangles in Kronecker graphs [101], a model for generating graphs which mimic properties of real-world networks.

The rest of the chapter is organized as follows: Section 3.2, presents briefly existing triangle-counting methods and the Singular value Decomposition. In Section 3.3 we present the EIGENTRIANGLE and EIGENTRIANGLELOCAL algorithms, for global and local triangle counting respectively and we explain why they are efficient. Section 3.4 presents the experimental results on several real data sets. In Section 3.5 we present a simple sampling algorithm which allows us to improve further the underlying idea of the EIGENTRIANGLE and several other theoretical ramifications. We conclude in Section 3.6.

3.2 Related work

In this section we briefly present previous work related to the triangle counting problem and basic background knowledge on the Singular Value Decomposition.

3.2.1 Counting Triangles

Let $G(V, E)$, n=$|V|$, m=$|E|$ be an undirected, unweighted, simple graph. A triangle is a set of three fully connected nodes. In this section we briefly review the state-of-the-art work related to the problems of global and local triangle counting. By global we refer to the problem of counting the total number of triangles in G and by local to the problem of counting the number of triangles per each node. Two other problems related to triangles are *(i)* deciding whether G contains a triangle or not and *(ii)* for each triangle in G, list the participating nodes.

Exact Counting: The brute force approach enumerates all possible triples of nodes resulting in a naive algorithm of $O(n^3)$ time complexity. Using this naive algorithm we can list exactly the triangles in G. Other listing methods include the *Node Iterator* and the *Edge Iterator*. The *Node Iterator* considers each one of the n nodes and examines which pairs of its neighbors are connected. The *Edge Iterator* algorithm computes for each edge the number of triangles that contain it. Asymptotically, both methods have the same time complexity $O(\sum_{v \in V} d_v^2)$ [147], which in the case of a dense graph are eventually $O(n^3)$. For sparse graphs, these methods are significant improvements over the naive algorithm. In [147] the $forward$ algorithm is proposed, which is an improvement of the *Edge Iterator* algorithm, with running time $\Theta(m^{\frac{3}{2}})$. In [98], a further improvement of the $forward$ algorithm is proposed, called the $compact$-$forward$ algorithm.

The algorithms with the lowest time complexity for counting triangles rely on fast matrix multiplication. The asymptotically fastest matrix multiplication algorithm to date is $O(n^{2.376})$ [38]. In [13] an algorithm of $O(m^{\frac{2\omega}{\omega+1}}) \subset O(m^{1.41})$ time complexity and of $\Theta(n^2)$ space complexity is proposed to find and count triangles in a graph. In practice, listing methods [147] are preferred against matrix-based methods because of the prohibitive memory requirements of the latter.

Approximate Counting: In many applications such as the ones mentioned in Section 3.1 the exact number of triangles is not crucial. Thus approximating algorithms which are faster and output a high quality estimate are desirable. Most of the approximate triangle counting algorithms have been developed in the streaming setting. In this scenario, the graph is represented as a stream. Two main representations of a graph as a stream are the edge stream and the incidence stream. In the former, edges are arriving one at a time. In the latter scenario all edges incident to the same vertex appear successively in the stream. The ordering of the vertices is assumed to be arbitrary. A streaming algorithm produces a relative ϵ-approximation of the number of triangles with high probability, making a constant number of passes over the stream. However, sampling algorithms developed in the streaming literature can be applied in the setting where the graph fits in the memory as well.

Monte Carlo sampling techniques have been proposed to give a fast estimate of the number of triangles. According to such an approach, a.k.a. naive sampling, we choose three nodes at random repetitively and check if they form a triangle or not. If one makes

$$r = \log(\frac{1}{\delta})\frac{1}{\epsilon^2}(1 + \frac{T_0 + T_1 + T_2}{T_3})$$

independent trials where $T_i = \#$triples with i edges and outputs as the estimate of triangles the random variable $T_3' = \binom{n}{3}\frac{\sum_{i=1}^{r} X_i}{r}$ then

$$(1-\epsilon)T_3 < T_3' < (1+\epsilon)T_3$$

with probability at least $1-\delta$. For graphs that have $T_3 = o(n^2)$ triangles this approach is not suitable. This is the typical case, when dealing with real-world networks. This sampling approach is presented in [133].

In the seminal paper [17] the authors reduce the problem of triangle counting efficiently to estimating moments for a stream of node triples. Then they use the Alon-Matias-Szegedy algorithms [11] (a.k.a. AMS algorithms) to proceed. Along the same lines, Buriol et al. in [27] proposed two space-bounded sampling algorithms to estimate the number of triangles. Again, the underlying sampling procedures are simple. E.g., for the case of the edge stream representation, they sample randomly an edge and a node in the stream and check if they form a triangle. Their algorithms are the state-of-the-art algorithms to our knowledge. In their three-pass algorithm, in the first pass they count the number of edges, in the second pass they sample uniformly at random an edge (i,j) and a node $k \in V - \{i,j\}$ and in the third pass they test whether the edges $(i,k),(k,j)$ are present in the stream. The number of draws that have to be done in order to get concentration (of course these draws are done in parallel), is of the order

$$r = \log(\frac{1}{\delta})\frac{2}{\epsilon^2}(3 + \frac{T_1 + 2T_2}{T_3})$$

Even if the term T_0 is missing compared to the naive sampling, the graph still has to be fairly dense with respect to the number of triangles in order to get an ϵ approximation with high probability. In [18] the semi-streaming model for counting triangles is introduced. The authors observed that since counting triangles reduces to computing the intersection of two sets, namely the induced neighborhoods of two adjacent nodes, ideas from the locality sensitivity hashing [62] are applicable to the problem of counting triangles. They relax the constraint of a constant number of passes over the edges, by allowing $\log n$ passes.

Doulion [154] proposed a new sampling procedure which is used in the Peta-Scale graph mining project. The approach of Doulion is the combinatorial perspective of the sparsification procedure proposed by [7] and by [150] in the multilinear setting, which has been used to speed up spectral counting approach of

[148] in [152]. The algorithm tosses a coin independently for each edge with probability p to keep the edge and probability $q = 1 - p$ to throw it away. In case the edge "survives", it gets reweighed with weight equal to $\frac{1}{p}$. Then, any triangle counting algorithm, such as the node- or edge- iterator, is used to count the number of triangles t' in G'. The estimate of the algorithm is the random variable $T = \frac{t'}{p^3}$. The following facts -among others- were shown in [154]:a) The estimator T is unbiased, i.e., $E[T] = t$ and the expected speedup when a simple exact counting algorithm as the node iterator is used, is $1/p^2$. The authors however did not answer the critical question, of how small can p be? Therefore [154] provides constant factor speedups leaving the question as a research topic. The answer concerning p was given recently in [155].

3.2.2 Singular Value Decomposition (SVD)

The Singular Value Decomposition (SVD) [139] is a powerful matrix decomposition frequently used for dimensionality reduction. SVD is widely used in problems involving least squares problems, linear systems and finding a low rank representation of a matrix. Furthermore, a wide range of applications uses SVD as its main algorithmic tool. Notable applications of the SVD are the HITS algorithm [87], Latent Semantic Indexing [118], and image compression [78].

The SVD theorem states that any matrix $A \in \mathbb{R}^{m \times n}$ can be written as a sum of rank one matrices, i.e., $A = \sum_{i=1}^{r} \sigma_i u_i v_i^T$, where $u_i, i = 1 \dots r$ (left singular vectors) and $v_i, i = 1 \dots r$ (right singular vectors) are orthonormal and the singular values are ordered in decreasing order $\sigma_1 \geq \dots \geq \sigma_r > 0$. Here r is the rank of A. We denote with A_k the k-rank approximation of A, i.e., $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$. Among all matrices $C \in \mathbb{R}^{m \times n}$ of rank at most k, A_k is the one that minimizes $||A - C||_F$.

An exhaustive listing of the work related to the SVD is impossible. We report here briefly the main result of [51], since it is related to our work. Therein, a fast randomized algorithm is presented to approximate the SVD of a given matrix A. Specifically, the authors approximate the left singular vectors and the singular values of the SVD using an appropriately sampled set of columns of the matrix. Similarly, the right singular vectors can be approximated via a row sampling procedure. The probability of choosing a specific column $A^{(i)}$ is equal to $p_i = \frac{||A^{(i)}||^2}{||A||_F^2}$. They prove that their k-rank approximation $\hat{A}_k$ satisfies the following form of inequality with probability at least 1-δ when the sampling procedure picks c columns of A: $||A - \hat{A}_k||_F^2 \leq ||A - A_k||_F^2 + f(\delta, k, c)||A||_F^2$, where $f(\cdot)$

Sym.	Definition
G	Undirected graph (no self-edges)
d_{max}	maximum node degree
Δ	total number of triangles
Δ'	EIGENTRIANGLE's estimation of Δ
$\boldsymbol{\Delta}(G) = [\Delta_i]_{i=1..n}$	Δ_i number of triangles node i participates
$\boldsymbol{\Delta}'(G) = [\Delta'_i]_{i=1..n}$	Δ'_i EIGENTRIANGLELOCAL's estimation of Δ_i
m, n	Number of edges and nodes.
$[n] = (1..n)$	Node ids
A	Adjacency matrix
$A^{(i)}$	i-th column of A
λ_i	top-i-th eigenvalue (absolute value)
u_i	top-i-th eigenvector corresponding to eigenvalue λ_i
$\Lambda_k = [\lambda_i]_{i=1..k}$	vector containing k top eigenvalues
$U_k = [u_1 \| \dots \| u_k]$	matrix containing the k top eigenvectors as its columns
$u_{i,j}$	the i-th entry of the j-th eigenvector

Table 3.1: Definitions of symbols used.

is a function of the three parameters k, c, δ as described in [51].

3.3 Proposed Method

In this section we present the proposed algorithms for the triangle counting problem and explain why they are efficient when applied to a real-world network. Table 3.1 gives a list of symbols and their definitions.

3.3.1 Theorems and proofs

The following theorem connects the number of triangles in which node i participates with the eigenvalues and eigenvectors of the adjacency matrix.

Theorem 6 *Let G be an undirected, simple graph and A is adjacency matrix representation. The number of triangles Δ_i that node i participates in satisfies the following equation:*

$$\Delta_i = \frac{\sum_j \lambda_j^3 u_{i,j}^2}{2} \tag{3.1}$$

where $u_{i,j}$ is the i-th entry of the j-th eigenvector and λ_j is the j-th eigenvalue of the adjacency matrix.

Proof 5 *Since G is undirected, A is a real, symmetric matrix. Thus, by the spectral theorem we can diagonalize A using its eigenvalues and eigenvectors. Therefore $A = U\Lambda U^T$, where Λ is a diagonal matrix containing the eigenvalues of A and $U = [u_1|\ldots|u_n]$ is the orthonormal matrix containing in its i-th column the eigenvector u_i corresponding to the i-th eigenvalue λ_i, $i = 1, \ldots, n$. By the orthonormality of U, it follows that $A^3 = U\Lambda^3 U^T$ ($\diamond$).*

Consider now α_{ii} the i-th diagonal element of A^3. α_{ii} is equal to twice (each triangle ijk is counted twice as $i \rightarrow j \rightarrow k \rightarrow i$ and $i \rightarrow k \rightarrow j \rightarrow i$) the number of closed walks of length three, i.e., the number of triangles in which node i participates. From equation ($\diamond$) follows that $\alpha_{ii} = \sum_j \lambda_j^3 u_{i,j}^2$. Combining these two facts we obtain for equation 3.1.

The following lemma holds, see [67, 148]:

Lemma 1 *The total number of triangles $\Delta(G)$ in the graph is given by the sum of the cubes of the eigenvalues of the adjacency matrix divided by six, i.e.,:*

$$\Delta(G) = \frac{1}{6}\sum_{i=1}^{n} \lambda_i^3 \tag{3.2}$$

3.3.2 Proposed algorithms

We propose algorithms 1 and 2, the EIGENTRIANGLE and EIGENTRIANGLE-LOCAL algorithms respectively. The former is based on Lemma 3.2, whereas the latter on Theorem 3.1. Both take as input the $n \times n$ adjacency matrix A and a tolerance parameter tol. EIGENTRIANGLE keeps computing eigenvalues until the contribution of the cube of the current eigenvalue is considered to be significantly smaller than the sum of the cubes of the previously computed eigenvalues. The tolerance parameter determines when the algorithm will stop looping, i.e., when we consider that the currently computed eigenvalue contributes little to the total number of triangles. The idea behind them is that due to the special spectral

Algorithm 4 The EIGENTRIANGLE algorithm

```
Require: Adjacency matrix $A$ $(n \times n)$
Require: Tolerance $tol$
Output: $\Delta'(G)$ global triangle estimation
  $\lambda_1 \leftarrow LanczosMethod(A, 1)$
  $\mathbf{\Lambda} \leftarrow [\lambda_1]$
  $i \leftarrow 1$ {initialize $i$, $\mathbf{\Lambda}$}
  repeat
    $i \leftarrow i + 1$
    $\lambda_i \leftarrow LanczosMethod(A, i)$
    $\mathbf{\Lambda} \leftarrow [\mathbf{\Lambda}\ \lambda_i]$
  until $0 \leq \frac{|\lambda_i^3|}{\sum_{j=1}^{i} \lambda_j^3} \leq tol$
  $\Delta'(G) \leftarrow \frac{1}{6} \sum_{j=1}^{i} \lambda_j^3$
  return $\Delta'(G)$
```

properties of real-world networks few iterations suffice to output a good approximation.

Specifically, EIGENTRIANGLE starts by computing the first eigenvalue λ_1. It then computes the second eigenvalue λ_2, and checks using the condition in the *repeat* loop if λ_2 contributes significantly or not to the current estimate of triangles, i.e., $\sum_{j=1}^{2} \lambda_j^3$. In the former case, the algorithm keeps iterating and computing eigenvalues until the stopping criterion is satisfied. Then, it outputs the estimate of the total number of triangles $\Delta'(G)$ using the computed eigenvalues and equation 3.2. EIGENTRIANGLELOCAL additionally stores the eigenvectors corresponding to the top eigenvalues in order to make an estimate of Δ_i using equation 3.1. The *repeat* loop as in EIGENTRIANGLE computes eigenvalue-eigenvector pairs until the stopping criterion is met and the *for* loop computes the estimates Δ_i' of Δ_i, $i = 1, \ldots, n$.

Both algorithms use the subroutine $LanczosMethod$ [78] as a black box[1] to compute a low-rank eigendecomposition of the adjacency matrix. Lanczos method is a well studied projection based method for solving the symmetric eigenvalue problem using Krylov subspaces. It is based on simple matrix-vector multiplications. Furthermore, high quality software implementing Lanczos method is publicly available (ARPACK, Parallel ARPACK, MATLAB etc.). It is worth

[1]For simplifying the presentation, depending on the number of output arguments, Lanczos returns either λ_i only or $\boldsymbol{u_i}$ too. The required time is (almost) the same in both cases.

Algorithm 5 The EIGENTRIANGLELOCAL algorithm

```
Require: Adjacency matrix $A$ $(n \times n)$
Require: Tolerance $tol$
Output: $\mathbf{\Delta}'(G)$ per node triangle estimation
  $\langle \lambda_1, \boldsymbol{u_1} \rangle \leftarrow LanczosMethod(A, 1)$
  $\mathbf{\Lambda} \leftarrow [\lambda_1]$
  $\mathbf{U} \leftarrow [\boldsymbol{u_1}]$
  $i \leftarrow 1$
  {initialize $i$, $\mathbf{\Lambda}, \mathbf{U}$}
  repeat
    $i \leftarrow i + 1$
    $\langle \lambda_i, \boldsymbol{u_i} \rangle \leftarrow LanczosMethod(A, i)$
    $\mathbf{\Lambda} \leftarrow [\mathbf{\Lambda}\ \lambda_i]$
    $\mathbf{U} \leftarrow [\mathbf{U}\ \boldsymbol{u_i}]$
  until $0 \leq \frac{|\lambda_i^3|}{\sum_{j=1}^{i} \lambda_j^3} \leq tol$
  for $j = 1$ to $n$ do
    $\Delta'_j = \frac{\sum_{k=1}^{i} u_{jk}^2 \lambda_k^3}{2}$
  end for
  $\mathbf{\Delta}'(G) \leftarrow [\Delta'_1, .., \Delta'_n]$
  return $\mathbf{\Delta}'(G)$
```

noting how easy it is to implement our algorithm in a programming language that offers routines for eigenvalue computation. For example, assuming that a k-rank approximation of the adjacency matrix gives good results, the piece of MATLAB code described in algorithm 3 will output an accurate estimate of the number of triangles. This function takes two input arguments, A and k which are the adjacency matrix representation of the graph and the desired rank of the low rank approximation respectively.

3.3.3 Why is EIGENTRIANGLE successful?

Real-world networks have several special properties, such as small-worldness, scale-freeness and self-similarity characteristics. For our work, the special spectral properties are crucial. Figure 1(a) and Figure 1(b) show the spectra of two real-world networks. Both are representative of the typical spectrum of a real-world network. These figures plot the value of the eigenvalue vs. its rank. The

Algorithm 6 MATLAB implementation, k-rank approximation

```
function $\Delta'$ = EigenTriangleLocal(A,k) {A is the adjacency matrix, k is the required rank approximation}
n = size(A,1);
$\Delta'$ = zeros(n,1); {Preallocate space for $\Delta'$}
opts.isreal=1; opts.issym=1; {Specify that the matrix is real and symmetric}
[u l] = eigs(A,k,'LM',opts); {Compute top k eigenvalues and eigenvectors of A}
l = diag(l)';
for j=1:n do
    $\Delta'(j)$ = sum( l.^3.*u(j,:).^2)/2
end for
```

spectrum of Figure 1(a) corresponds to the Political Blogs network [**?**], a small network with approximately 1,2K nodes and 17K edges. The spectrum of Figure 1(b) corresponds to an anonymous social network with approximately 404K nodes and 2,1M edges. Notice that in the latter network, only the 800 top eigenvalues out of the approximately 404K eigenvalues are plotted.

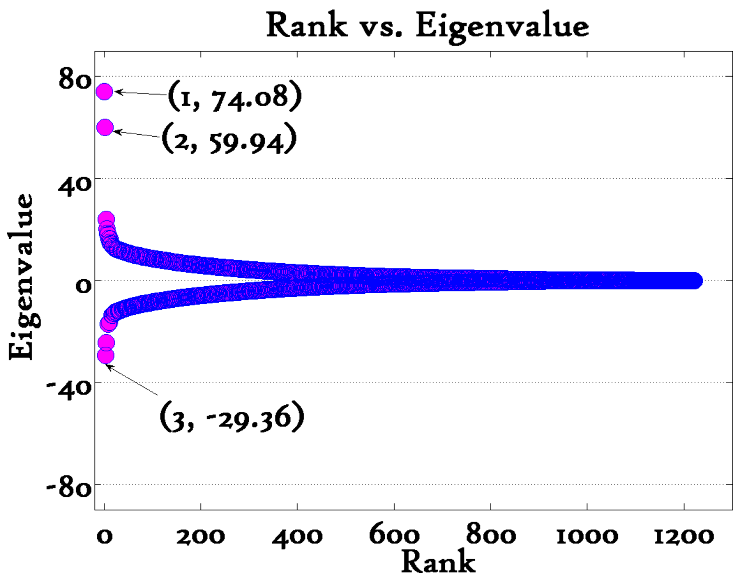
The following two facts which are apparent in the two figures, play a crucial role in the effectiveness of our proposed algorithms:

1. The absolute values of the few top eigenvalues are skewed, typically following a power law [58][2],[109],[36].

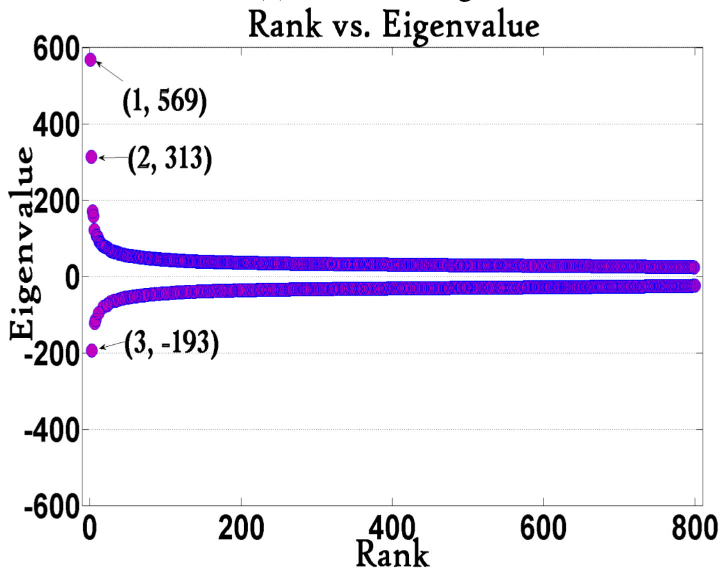
2. Moreover, the signs of the eigenvalues tend to alternate [59] and thus their cubes roughly cancel out.

In other words, the contribution of the bulk of the eigenvalues is negligible compared to the contribution of the few top eigenvalues to the total number of triangles. This fact allows us to discard the largest part of the spectrum. Therefore we can keep just a handful of eigenvalues and approximate fast and well the number of triangles. Experimentally 1 to 25 eigenvalues, see Figure2(a), lead to a satisfactory approximation. The time complexity of our proposed algorithms is $O(c\text{nnz})$ where nnz is the number of non zeros in the adjacency matrix, i.e., twice the number of edges, and c is the total number of matrix vector multiplications

[2]Even if the least squares fitting used in [58] has been questioned as a methodology of fitting power laws and better methodologies have been developed [37], the key property is the skewness observed in the values of the top eigenvalues rather than the exact distribution that they follow.



(a) Political Blogs



(b) Anonymous Social Network

Figure 3.1: Spectra of two real-world networks, representative of the typical spectrum of networks with skewed degree distributions. Both figures (a) and (b) plot the value λ_i versus the rank i. Political blogs is a small network with ≈17K edges and ≈1,2K nodes. The Anonymous Social Network has ≈404K nodes and ≈2,1M edges. Figure (b) plots only the 800 top eigenvalues. Notice that (1) the first few eigenvalues are significantly larger than the rest, (2) which are almost symmetric around zero and (3) cubing amplifies these effects.

Lanczos method performs. As we explain in the next subsection, the computation of a handful of the top eigenvalues results in a small number of iterations c and therefore the performance of our methods is fast.

3.3.4 Lanczos method and Real-World Networks

First we give a brief description of Lanczos method for computing the eigenvalues of a symmetric matrix and then we explain why it converges fast in the case of real-world networks.

Short Description of Lanczos Method: Consider a symmetric $n \times n$ matrix A whose eigenvalues and eigenvectors are sought and let $u \in \mathbb{R}^n$ be a given unit vector. Lanczos method is based on the subspace spanned by the vectors $u, Au, \ldots, A^{k-1}u$, also known as the Krylov subspace. Let K be the $n \times k$ matrix $K = [u|Au|\ldots|A^{k-1}u]$. For $k \leq m \leq n$, where m is the order of the minimal polynomial of u with respect to A, matrix K has full column rank. However, since the successive multiplications of matrix A lead the terms $A^j u$ for large j to being almost equal to the first eigenvector, it is necessary to get a numerically better base for this subspace. Using the Gram-Schmidt orthogonalization procedure we produce an orthonormal sequence of vectors $u = q_1, \ldots, q_k$ such that the following three term recurrence equation holds:

$$Aq_j = b_{j-1}q_{j-1} + a_j q_j + b_j q_{j+1} \tag{3.3}$$

The coefficients a_j, b_j can be found by using the orthogonality properties of the q_j vectors. Let Q be the matrix $Q = [q_1|\ldots|q_k]$. The matrix $Q^T AQ$ is a small $k \times k$, tridiagonal matrix (containing the coefficients $a_1, \ldots, a_k$ in its main diagonal, and the coefficients $b_1, \ldots, b_{k-1}$ in the first diagonal above and below the main one) whose eigenvalues typically approximate well the top k eigenvalues of A. It is also worth noting that Lanczos method performs only matrix-vector multiplications making it a good option for a low rank approximation of a sparse matrix A. For more details see [78].

Convergence of Lanczos method: As we know, the eigenvalues of matrix A are the roots of its characteristic polynomial. The latter is also known as the secular function. When the roots of the secular function are very close, Lanczos needs several iterations to find them. Even if there exist sophisticated methods for finding the roots of the secular function, they run into similar problems with Newton's method when the two roots we are trying to find are very close.

Since real-world networks tend to have skewed degree distributions which imply a skewed eigenvalue distribution too, Lanczos converges fast to the top eigenvalues because they correspond to roots of the secular function which are well separated. Therefore, assuming that the top eigenvalues provide us a satisfactory approximation to the total number of triangles implies that we can find fast a good estimate of the total number of triangles.

3.4 Experimental Results

We conduct numerous experiments in order to answer the following question: for at least 95% accuracy what are the speedups we can achieve for the triangle counting problem using EIGENTRIANGLE? First, we describe the experimental setup, and then we provide the experimental results.

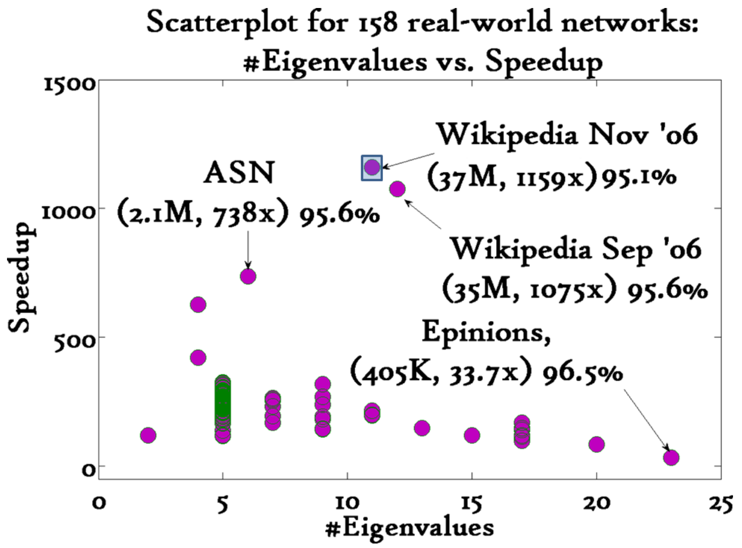
3.4.1 Experimental set up

Each directed graph was converted into an undirected graph by ignoring the direction of the edges. Multiple edges and self-loops were removed. The number of nodes and edges of the networks used after the preprocessing are summarized in table 3.2. [3] As the competitor for our method we chose the *Node Iterator* (see section 3.2), a basic, non-trivial exact listing algorithm which allows us to directly evaluate the quality of EIGENTRIANGLE and EIGENTRIANGLELOCALby comparing the outputs. We ran the experiments in a machine with a quad-processor Intel Xeon 3GHz with 16GB of RAM. We express the experimental results as the ratio of the clock-work times of the *Node Iterator* to the EIGENTRIANGLE (speedup). All algorithms were implemented in MATLAB. For the eigenvalue computation, we used the command *eigs* to which we passed a struct *opts*, specifying that our matrices are symmetric and real, as shown in Algorithm 3.

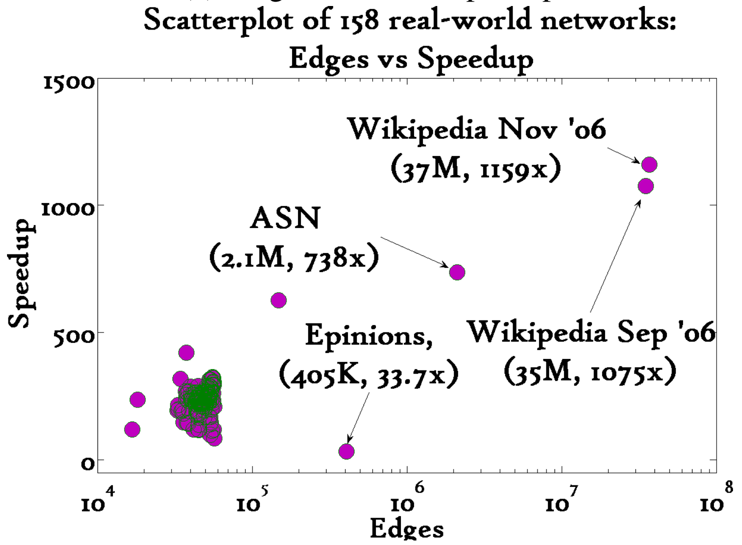
[3] Most of the datasets we used are publicly available. Indicative sources are : `http://arxiv.org`, `http://www.cise.ufl.edu/research/sparse/mat/`, `http://www-personal.umich.edu/~mejn/netdata/`

Nodes	Edges	Description
Social Networks		
75,877	405,740	Epinions network
404,733	2,110,078	Anonymous Social Network (ASN)
Co-authorship networks		
27,240	341,923	Arxiv Hep-Th
Information networks		
1,222	16,714	Political blogs
13,332	148,038	Reuters news, Sept 9-11,2001.
Web graphs		
2,983,494	35,048,116	Wikipedia 2006-Sep-25
3,148,440	37,043,458	Wikipedia 2006-Nov-04
Internet networks		
13,579	37,448	AS Oregon
23,389	47,448	CAIDA AS 2004 to 2008 (means over 151 timestamps)

Table 3.2: Summary of real-world networks used.



(a) #Eigenvalues vs. Speedup



(b) Edges vs. Speedup

Figure 3.2: Scatterplots of the results for 158 graphs. (a) Speedup vs. Eigenvalues: The mean required approximation rank for $\geq$ 95% accuracy is 6.2. Speedups are between 33.7x and 1159x, with mean 250.(b) Speedup vs. Edges: Notice the trend of increasing speedup as the network size grows (#edges).

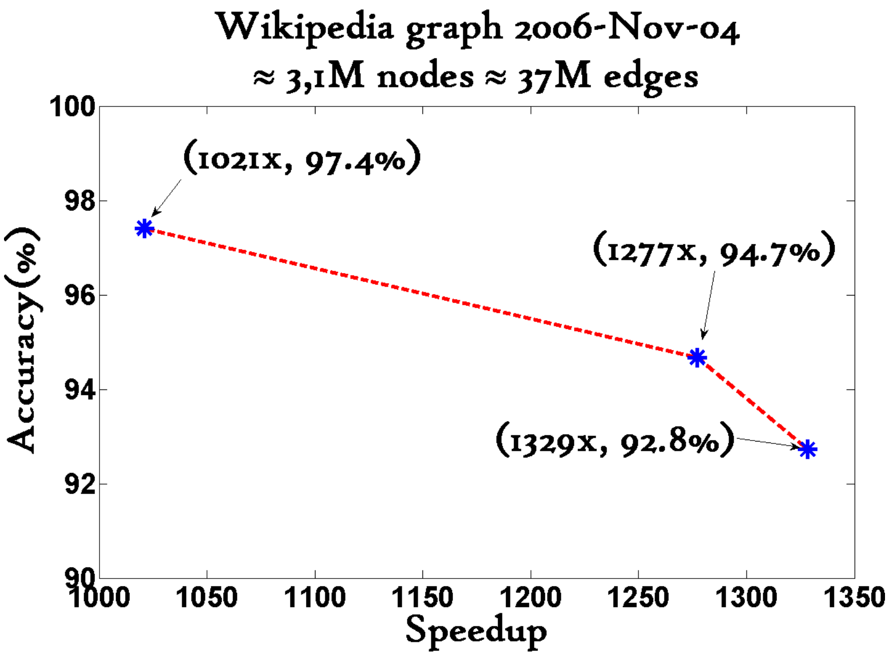


Figure 3.3: Zooming in the point enclosed by a rectange of figure 3.2(a). This figure plots the accuracy obtained versus the speed-up ratio for the Wikipedia web graph ($\approx 3,1M$ nodes, $\approx 37M$ edges). Proposed method achieves 1021x faster time, for 97.4% accuracy, compared to a typical competitor, the *Node Iterator* method.

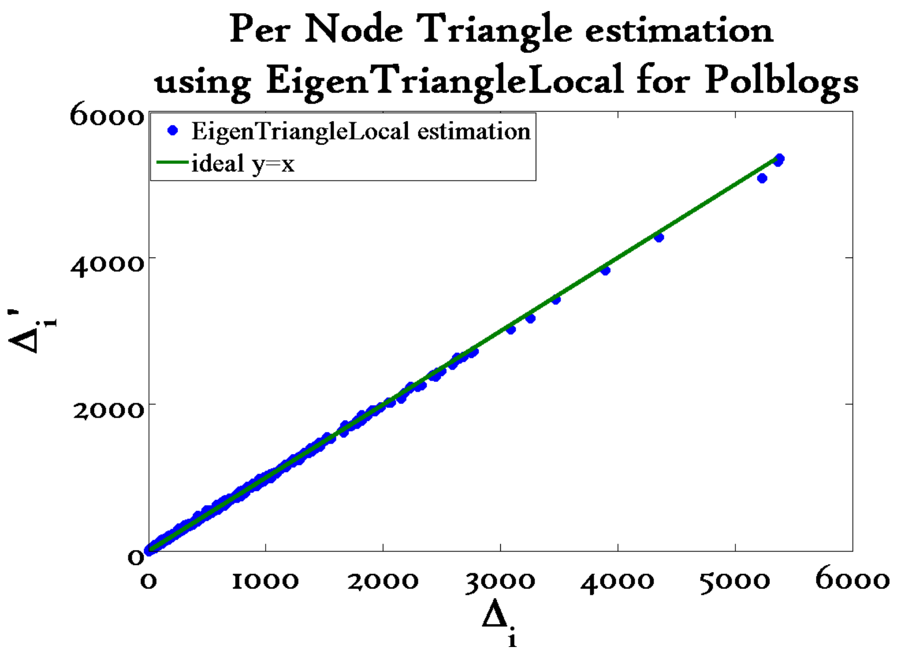


Figure 3.4: Scatterplot of Δ_i' (estimated #triangles of node i) vs. Δ_i (actual number) for Polblogs using a rank 10 approximation. Relative reconstruction error is $7 * 10^{-4}$ and the Pearson's correlation coefficient is 99.97%.

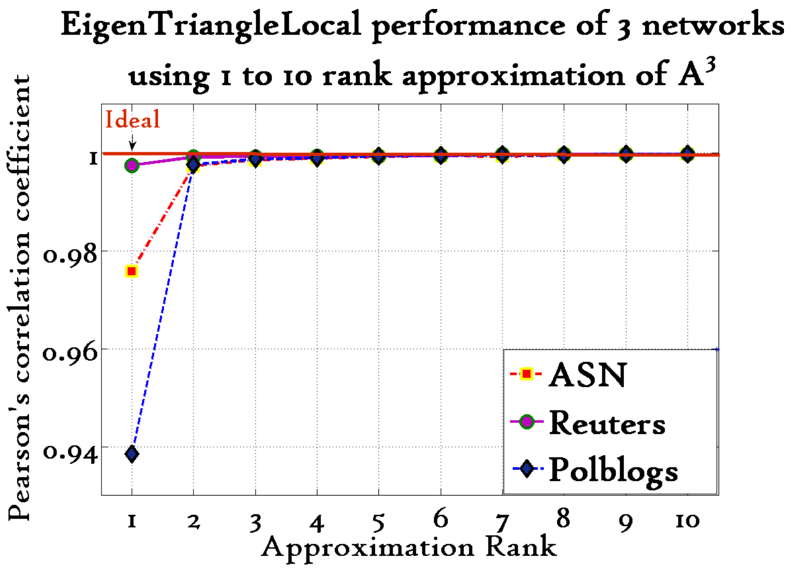


Figure 3.5: Local triangle reconstruction for three real-world networks using rank 1 to 10 approximation of the diagonal of $\mathbf{A}^3$. Pearson's correlation coefficient ρ vs. approximation rank.Notice that after rank 2 ρ is greater than 99.9% for all three networks.

3.4.2 Total Triangle Counting

Figures 3.2(a), 3.2(b) summarize the results of the EIGENTRIANGLE algorithm when applied to 158 real world networks. Specifically, Figure 3.2(a) plots the achieved speedup versus the number of eigenvalues required to get at least 95% accuracy. Figure 3.2(b) plots the speedup versus the number of edges in the graph. The following facts are worth noting:

1. The mean number of eigenvalues required to achieve more than 95% is 6.2 with standard deviation equal to 3.2. The mean speedup is $250\times$ with the standard deviation equal to 123. The maximum speedup is $1159\times$ whereas the minimum speedup is $33.7\times$.

2. The speedup appears to increase as the size of the network grows. A possible explanation for this, assuming that our degree distribution follows approximately a power law, could be that as the network grows, the maximum degrees are getting more detached from the rest. According to [109], the top eigenvalues exhibit the same behavior, i.e., get more detached from the bulk. Therefore, with a handful of eigenvalues, we get high accuracy, since their cubes dominate the total sum of the cubes of the eigenvalues. Furthermore, due to the fast convergence of Lanczos method, EIGENTRIANGLE ouputs fast its estimate.

   An exception to the observation above is the performance of our method on the Epinions graph. EIGENTRIANGLE needs to compute more than 20 eigenvalues to ouput a high quality estimate, due to the specific spectrum of this graph. This fact has as a consequence the smallest speedup observed ($33.7\times$) which is still significant.

3. An important issue in EIGENTRIANGLE and EIGENTRIANGLELOCAL is the choice of the tolerance parameter tol. Clearly, if the parameter is set to $\epsilon \rightarrow 0$, both algorithms will have to compute many eigenvalues slowing down significantly their performance. An extremely small value for the parameter tol is likely to turn the proposed algorithms into slower than other exact counting algorithms, since computing the whole spectrum of a square $n \times n$ matrix has time complexity $O(n^3)$ with potential convergence and numerical problems. On the other hand, if the tolerance parameter is set to a high value, then the accuracy of the estimate can be unsatisfactory. It is not clear how to decide the tol parameter a priori. However, this does not render EIGENTRIANGLE useless. A useful "rule of thumb" for practitioners

based on Figure 3.2(a) is to compute 5-15 eigenvalues and see how well does the sum S_i of the cubes of the eigenvalues from 1 to i compare to S_{i+1}. This is essentially the same criterion with the stopping criterion of the algorithms we propose. However, using this “rule of thumb” is a practical way of running the algorithms without depending on the parameter tol.If one wants to run the algorithm as is, a choice of tol that was satisfactory in many experiments was 0.05.

4. Figure 3.3 is zooming in the point enclosed with a rectangle of Figure 3.2(a). This point corresponds to the Wikipedia Web graph (4 Nov. 2006 with approximately 3,1M nodes, and 37M edges). We observe that with a single eigenvalue we get 92.8$\%$ accuracy and 1329$\times$ speedup. When the algorithm terminates, the accuracy is 97.4%, the speedup 1021$\times$ and the rank of the required approximation equal to 7.

3.4.3 Local Triangle Counting

To measure the performance of the EigenTriangleLocal algorithm, we use Pearson’s correlation coefficient ρ and the relative reconstruction error, as in [18].

$$RRE = \frac{1}{n}\sum_{i=1}^{n} \frac{|\Delta_i - \Delta_i'|}{\Delta_i} \tag{3.4}$$

In figure 3.4 we see how well $\mathbf{\Delta}'(G)$, i.e., the vector which contains in its i-th coordinate our estimate of the number of triangles in which node i participates in, approximates $\mathbf{\Delta}(G)$ using the top 10 eigenvalues and eigenvectors for the Political blogs dataset. The RRE we obtain is $7 * 10^{-4}$ and ρ is equal to 0.9997, close to the ideal value 1. Figure 3.5 explains why our proposed methods work well in practice. It plots ρ versus the rank of the approximation. We observe that after the two rank approximation, for all three networks the approximation is excellent: ρ is greater than 99.9$\%$ whereas the RRE has always order of magnitude between 10^{-7} and 10^{-4}. Similar results hold for the rest of the datasets we experimented with. Finally, it is worth noting that figure 3.5 suggests that the rank-10 approximation of the adjacency matrix used to produce Figure 3.4 is significantly larger than the minimum one needed to obtain satisfactory results.

3.5 Theoretical Ramifications

In this section we extend our theoretical results in the following three ways. First, we show a simple sampling procedure allows us to apply the core idea of EIGENTRIANGLE on large graphs which do not fit into the main memory. The resulting algorithm is the FastSVD and is based on the seminal work of [51]. Secondly, using the spectral counting idea, we prove a theorem which provides a closed formula for the number of triangles in Kronecker graphs. Finally, we discuss about cases where the EIGENTRIANGLE algorithm still works, even if the graph is not a "real-world" network.

3.5.1 Counting Triangles via Fast SVD

We consider the following simple randomized procedure to speedup further the performance of our proposed algorithms: Given our $n \times n$ adjacency matrix A, integers c, k such that $c \leq n$, $k \leq c$, we sample c integers from 1 to n, with the probability of choosing integer i equal to $Pr(i) = p_i = \frac{d_i}{2m}$, where d_i is the degree of node i and m is the total number of edges in the graph. Let $\{i_1, \ldots, i_c\}$ be the indices sampled. We create a $n \times c$ matrix $A' = [\frac{A^{(i_1)}}{\sqrt{cp_{i_1}}}|\frac{A^{(i_2)}}{\sqrt{cp_{i_2}}}|\ldots|\frac{A^{(i_c)}}{\sqrt{cp_{i_c}}}]$. We use A' to approximate the k top eigenvalues and eigenvectors of A, where k is assumed to be the required rank of the approximation of the adjacency matrix which gives us a good estimate of the number of triangles in the graph. The top k left singular vectors $\hat{u}^{(i)}_{i=1\ldots k}$ of A' define a subspace which is close to the optimal k dimensional subspace spanned by the top k left singular vectors $u^{(i)}_{i=1\ldots k}$ of A. In order to approximate the right singular vectors as suggested by [51] one should sample rows of A. Instead, we choose to approximate the right singular vectors using the equation $\hat{V}^T = \hat{\Sigma}^{-1}\hat{U}^T A$ assuming that $\hat{\Sigma}^{-1}\hat{U}^T A \approx \Sigma U^T A$. The signs of the eigenvalue λ_i can be recovered by multiplying the corresponding left and right singular vectors. For example if we had the exact SVD of A we could determine the i-th eigenvalue by $\lambda_i = \sigma_i (v^{(i)})^T u^{(i)}$. We approximate λ_i by $\hat{\lambda}_i$ where $\hat{\lambda}_i \leftarrow \hat{\sigma}_i \text{sgn}((\hat{v}^{(i)})^T \hat{u}^{(i)})$. The reason that the sign function appears[4] is that the ideal situation where the inner product $(v^{(i)})^T u^{(i)}$ should equal either +1 or -1 does not occur in practice. This procedure results in algorithm 4. The reason that this procedure is theoretically sound is the seminal work of [51]. Specifically, since our matrix is a square, symmetric matrix containing only zeros and ones, the

[4]The sign function $\text{sgn}(\cdot)$ returns the sign of its argument.

Algorithm 7 The FastSVD Triangle Counting algorithm

```
Require: Adjacency matrix $A$ $(n\text{x}n)$
Require: c, $c \leq n$
Require: k, $k \leq c$
Output: $\Delta'(G)$ global triangle estimation
  for $j = 1$ to $c$ do
    Pick an integer from $\{1, \ldots, n\}$, where $p_i = \frac{d_i}{2m}$
    Include $\frac{A^{(i)}}{\sqrt{cp_i}}$ as a column of $A'$
  end for
  Compute the top k left singular vectors $\hat{u}^{(1)}, \ldots, \hat{u}^{(k)}$ and the top k singular
  values $\hat{\sigma}_1 > \ldots > \hat{\sigma_k} > 0$ of $A'$
  $\hat{U} \leftarrow [\hat{u}^{(1)}| \ldots |\hat{u}^{(k)}]$
  $\hat{\Sigma} \leftarrow \text{diag}(\hat{\sigma}_1, \ldots, \hat{\sigma_k})$
  $\hat{V}^T \leftarrow \hat{\Sigma}^{-1}\hat{U}^T A$
  for $j = 1$ to $k$ do
    $\hat{\lambda}_j \leftarrow \hat{\sigma}_j \text{sgn}((\hat{v}^{(j)})^T \hat{u}^{(j)})$
  end for
  $\Delta'(G) \leftarrow \frac{1}{6}\sum_{i=1}^{k} \hat{\lambda}_i^3$
  return $\Delta'(G)$
```

probabilities $p_i = \frac{||A^{(i)}||^2}{||A||_F^2}$ defined in [51] are simplified to the expression $\frac{d_i}{2m}$. Intuitively, by favoring nodes of high degree we can recover the number of triangles approximately.

We apply Algorithm 4 on the anonymous social network, for which with 6 eigenvalues we obtain a 95.6% accuracy using Lanczos method. The obtained accuracy using Algorithm 4 is 95.46% using k equal to 6 and c equal to 100. With both algorithms we are able to compute with high accuracy an estimate of the 38036823 total triangles which exist in the graph. The speedup is not apparent due to the overhead of the sampling procedure and the necessary multiplications we make to find the signs of the singular values. Combined with the overall small amount of time needed to compute the top six eigenvalues (less than 4 seconds) the performance of EIGENTRIANGLE and Algorithm 4 are comparable. Nonetheless, algorithm 4 is useful, allowing us to apply the core idea of EIGENTRIANGLE on graphs which do not fit into the main memory.

3.5.2 Kronecker graphs

Kronecker graphs [101] have attracted recent interest, because they can be made to mimic real graphs well. In the following we give a closed formula that estimates the number of triangles for a Kronecker graph. Some definitions first:

Let A be the $n \times n$ adjacency matrix of an n-node graph G_A with $\Delta(G_A)$ triangles, and let $B = A^{[k]}$ be the k-th Kronecker power of it, that is, an $n^k \times n^k$ adjacency matrix (see [101] for the exact definition of the deterministic Kronecker graph). Let G_B denote the corresponding graph. Let $\boldsymbol{\lambda} = (\lambda_1, .., \lambda_n)$ be the eigenvalues of matrix A. The following theorem holds:

Theorem 7 (KRONECKERTRC) *The number of triangles $\Delta(G_B)$ of G_B can be computed from the n eigenvalues of A:*

$$\Delta(G_B) = 6^k \Delta(G_A)^{k+1} \;\; k \geq 0. \tag{3.5}$$

Proof 6 *We use induction on the depth of the recursion k. For $k = 0$,* KRONECKERTRC *trivially holds. So the base case is true. Let* KRONECKERTRC *hold for some $r \geq 1$. For notation simplicity, let $C = A^{[r]}$ with eigenvalues $[\mu_i]_{i=1..s}$ and $D = \mathbf{A}^{[r+1]}$. According to the induction assumption:*

$$\Delta(G_C) = 6^r \Delta(G_A)^{r+1}$$

The eigenvalues of D are given by the Kronecker product $\boldsymbol{\lambda} \otimes \boldsymbol{\mu}$. Using these two facts, we will now show that KRONECKERTRC *holds for $r + 1$. By Lemma 3.2, we get that the number of triangles in G_D is given by the following equation:* $\Delta(G_D) = \frac{\sum_{i=1}^{s} \sum_{j=1}^{n} \mu_i^3 \lambda_j^3}{6} = \frac{\sum_{i=1}^{s} \mu_i^3 \sum_{j=1}^{n} \lambda_j^3}{6} = \frac{\sum_{i=1}^{s} \mu_i^3 6\Delta(G_A)}{6} = 6\Delta(G_A)\frac{\sum_{i=1}^{s} \mu_i^3}{6} = 6\Delta(G_A) 6^r \Delta(G_A)^{r+1} = 6^{r+1}\Delta(G_A)^{r+2}$

Therefore KRONECKERTRC *holds for all $k \geq 0$.*

Timing results, and stochastic Kronecker graphs The above theorem results in tremendous time savings and perfect accuracy for deterministic Kronecker graphs. For example, experimenting on a small deterministic Kronecker graph with 6,561 nodes and 839,808 edges coming from the 3-clique initiator with depth of recursion equal to 7, we get 10^6 faster performance. As the size of the Kronecker graph increases, we obtain arbitrarily large speedups.

It is interesting that the KRONECKERTRC theorem also leads to a fast estimation of triangles, even for stochastic Kronecker graphs [101]. Stochastic Kronecker graphs have been shown to mimic real graphs very well. Intuitively, a

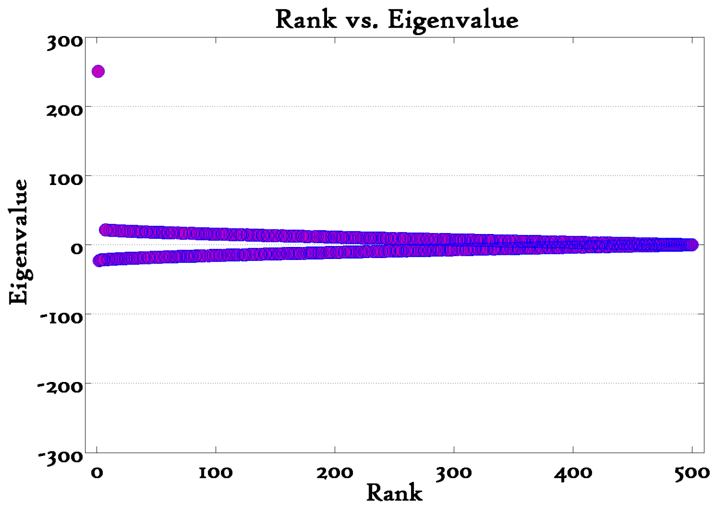


Figure 3.6: Eigenvalue vs. rank plot of a random Erdős-Rényi graph $G_{n,p}$, with n =500 and $p = \frac{1}{2}$.

stochastic Kronecker graph is like a deterministic one, with a few random edge deletions and additions. Our experiments with a stochastic Kronecker graph show that these random edge manipulations have little effect on the accuracy. Specifically, our experiments with n=6,561 and m=2,202,808[5], show that we obtain $1.5 * 10^6\times$ faster execution, while maintaining 99.34% accuracy. Similar results hold for other experiments we conducted as well. Proving bounds for the accuracy for stochastic Kronecker graphs is an interesting research direction.

3.5.3 Erdős-Rényi graphs

It is interesting to notice that our algorithm is guaranteed to give high accuracy and speedup performance for random Erdős-Rényi graphs [23]. This is due to Wigner's semi-circle law for all but the first eigenvalue [64]. In figure 3.6 we see the eigenvalue-rank plot for an Erdős-Rényi graph with n =500 and $p = \frac{1}{2}$, i.e., p constant.

For example, for a graph with $n = 20,000$ and $p = 0.6$, using EIGENTRIANGLELOCAL with 0.05 tolerance parameter, we get 1600 faster performance compared to the *Node Iterator* with relative error $5 * 10^{-5}$ and Pearson's correlation

[5]Seed matrix (using MATLAB notation): [.99 .9 .9;.9 .99 .1;.9 .1 .99], depth of recursion: 7

coefficient almost equal to 1[6].

3.6 Conclusions

In this work, we propose the EIGENTRIANGLE and EIGENTRIANGLELOCAL algorithms [148] to estimate the total number of triangles and the number of triangles per node respectively in an undirected, unweighted graph. The special spectral properties which real-world networks frequently possess make both algorithms efficient for the triangle counting problem. We showed experimentally that our method outperforms a straight-forward, exact triangle counting algorithm using different types of real-world networks. To our knowledge, the knowledge for the bulk of the spectrum is limited in contrast to the few, top eigenvalues [109, 36]. An interesting theoretical problem is to find the distribution of the bulk of the eigenvalues of a random graph generated by a model which mimics real-world networks. As the underlying eigendecomposition algorithm we use Lanczos method, which converges fast as we explain in Section 3.3. In practice, EIGENTRIANGLE using in average a rank six approximation of the adjacency matrix results in at least 95% accuracy, for speedups ranging from $30\times$ to $1000\times$ compared to the *Node Iterator* algorithm. However, this behavior is empirical and requires further theoretical justification and understanding. More experiments is another future direction, in order to establish to what extent real-world networks share similar spectral properties.

We also provide a simple randomized algorithm which allows us to use the core idea of EIGENTRIANGLE on graphs which do not fit in the main memory. The key idea behind this lies in the seminal work of [51] and the fact that we can find the eigendecomposition of the adjacency matrix through its Singular Value Decomposition. Furthermore, we give a closed formula for the number of triangles in deterministic Kronecker graphs and show that the same formula can be used to approximate satisfactorily the number of triangles in a stochastic Kronecker graph as well.

It is worth noting that since [148] other combinatorial triangle counting algorithms have been developed [154] with strong theoretical guarantees [155]. These algorithms are independent of any special spectral properties. Giving guarantees for the performance EIGENTRIANGLE algorithm under some random graph

[6]It makes no sense to apply EIGENTRIANGLE on Erdős-Rényi since we can approximate well the total number of triangles, i.e., $\binom{n}{3}p^3$.

model, e.g., [36] is another research direction as already mentioned. Nonetheless, EIGENTRIANGLE is a viable option for computing triangles in real-world networks which also shows that restricting our input graphs to possess special properties like those possessed empirically by real-world networks can lead us in developing efficient algorithms. Investigating further properties of real-world networks and developing such algorithms is another broad research direction.

Chapter 4

Fast Radius Plot and Diameter Computation for Terabyte Scale Graphs

4.1 Introduction

How does a real, Terabyte-scale graph look like? How do we compute the diameter and node radii in graphs of such size? Graphs appear in numerous settings, such as social networks (Facebook, LinkedIn), computer network intrusion logs, who-calls-whom phone networks, search engine clickstreams (term-URL bipartite graphs), and many more. 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 4.1, 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 4.1, which also shows the effective diameter and the center node of the Web(‘google.com’).

The rest of the chapter is organized as follows: Section 4.2 defines related terms and a sequential algorithm for the Radius Plot. Section 4.3 describes large

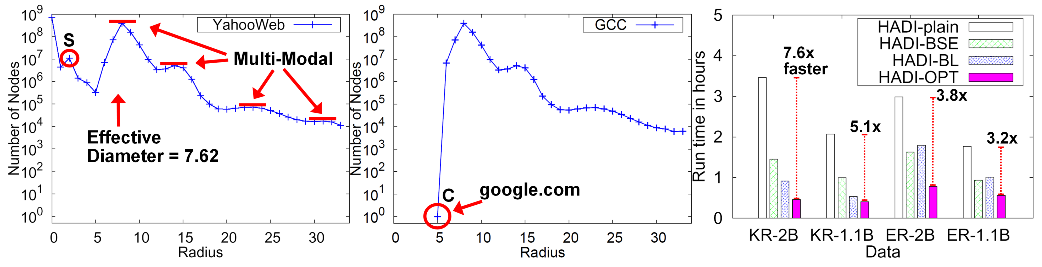


Figure 4.1: **(Left)** Radius Plot(Count versus Radius) of the YahooWeb graph. Notice the effective diameter is surprisingly small. Also notice the peak(marked 'S') at radius 2, due to star-structured disconnected components.
(Middle) Radius Plot of GCC(Giant Connected Component) of YahooWeb graph. The *only* node with radius 5 (marked 'C') is `google.com`.
(Right) 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**$\times$ faster than HADI-plain.

scale algorithms for the Radius Plot, and Section 4.4 analyzes the complexity of the algorithms and provides a possible extension. In Section 4.5 we present timing results. After describing background knowledge in Section 4.6, we conclude in Section 4.7.

4.2 Preliminaries; Sequential Radii Calculation

4.2.1 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* $r(v)$ of 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, $d(G) = \max_v r(v)$.

Since the radius and the diameter are susceptible to outliers (e.g., long chains), we follow the literature 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 distances from v.*

Definition 2 (Effective Diameter) *The effective diameter $d_{eff}(G)$ of a graph G*

Symbol	Definition
G	a graph
n	number of nodes in a graph
m	number of edges in a graph
d	diameter of 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 4.1: Table of symbols

is the minimum number of hops in which 90% of all connected pairs of nodes can reach each other.

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, as shown in Figure 4.1.
2. **Temporal Radius Plot** shows the distributions of effective radius of nodes at several times.
3. **Radius-Degree Plot** shows the scatter-plot of the effective radius $r_{eff}(v)$ versus the degree d_v for each node v.

Table 4.1 lists the symbols used in this paper.

4.2.2 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 4.6, 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 HADOOP /MAPREDUCE (the algorithm can also run in a parallel RDBMS).

To approximate the effective radius and the effective diameter, we use the Flajolet-Martin algorithm [61][115] for counting the number of distinct elements in a multiset. While many other applicable algorithms exist (e.g., [22], [31], [65]), 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 [12].

The main idea is that 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:

$$b(h,i) = b(h-1,i) \text{ BIT-OR } \{b(h-1,j) | (i,j) \in E\} \tag{4.1}$$

where "BIT-OR" denotes bitwise OR. 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

$$N(h,i) = \frac{1}{0.77351} 2^{\frac{1}{K}\sum_{l=1}^{K} b_l(i)} \tag{4.2}$$

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 can 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 $N(h) = \sum_i N(h,i) \geq 0.9 \cdot N(h_{max})$.

Algorithm 4.2.2 shows the summary of the algorithm described above.

The parameter K is typically set to 32[61], and $MaxIter$ is set to 256 since real graphs have relatively small effective diameter. The NewFMBitstring() function in line 2 generates K FM bitstrings [61]. The effective radius $r_{eff}(i)$ is determined at line 21, and the effective diameter $d_{eff}(G)$ is determined at line 23.

Algorithm 1: Computing Radii and Diameter

```
Input: Input graph G and integers $MaxIter$ and $K$
Output: $r_{eff}(i)$ of every node $i$, and $d_{eff}(G)$
for $i = 1$ to $n$ do
    $b(0, i) \leftarrow$ NewFMBitstring($n$);
end
for $h = 1$ to $MaxIter$ do
    $Changed \leftarrow 0$;
    for $i = 1$ to $n$ do
        for $l = 1$ to $K$ do
            $b_l(h, i) \leftarrow b_l(h-1, i)$BIT-OR$\{b_l(h-1, j) | \forall j$ adjacent from $i\}$;
        end
        if $\exists l$ s.t. $b_l(h, i) \neq b_l(h-1, i)$ then
            increase $Changed$ by 1;
        end
    end
    $N(h) \leftarrow \sum_i N(h, i)$;
    if $Changed$ equals to 0 then
        $h_{max} \leftarrow h$, and break for loop;
    end
end
for $i = 1$ to $n$ do
    // estimate eff. radii
    $r_{eff}(i) \leftarrow$ smallest $h'$ where $N(h', i) \geq 0.9 \cdot N(h_{max}, i)$;
end
$d_{eff}(G) \leftarrow$ smallest $h'$ where $N(h') \geq 0.9 \cdot N(h_{max})$;
```

Algorithm 4.2.2 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 4.2.2 runs faster than previous approaches (see Section 4.6 for discussion). However, it 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 can not fit into a single machine. In the next sections we present efficient parallel algorithms.

4.3 Proposed Method

In the next two sections we describe HADI, a parallel radius and diameter estimation algorithm. As mentioned in Section 4.2, 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 parallel SQL DBMS is sketched in Section 4.4.

4.3.1 HADI Overview

HADI follows the flow of Algorithm 4.2.2; that is, it uses the FM bitstrings and iteratively updates them using the bitstrings of its neighbors. The most expensive operation in Algorithm 4.2.2 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 optimization 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 it 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. We use HADOOP to describe the MAPREDUCE version of HADI.

4.3.2 HADI-naive in MAPREDUCE

HADI-naive is inefficient, but we present for illustration purposes.

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 *don't 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 (4.1) 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 such 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 machine 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 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 4.2.

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.

4.3.3 HADI-plain in MAPREDUCE

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

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 4.2.2, can be performed in completely parallel way. The $flag$ of each node records the following information:

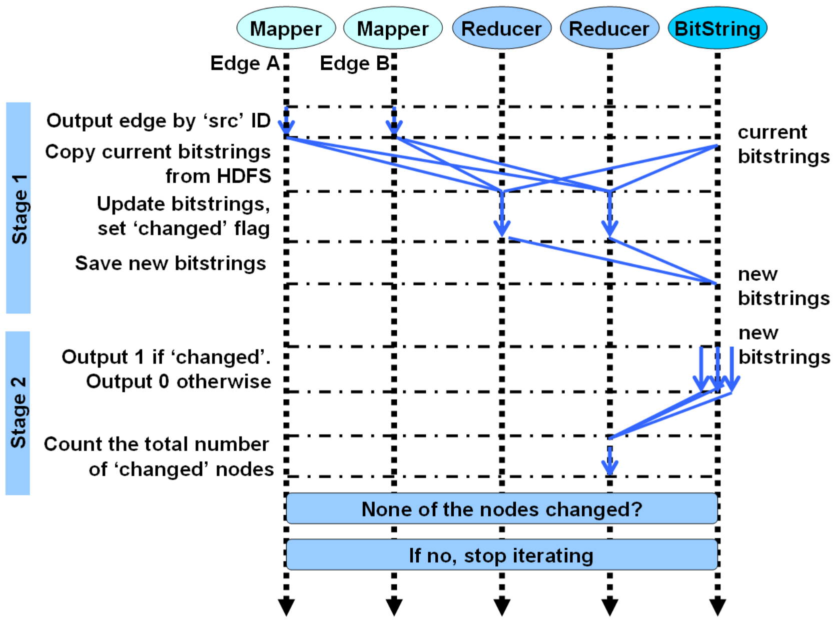

Mapper
Mapper
Reducer
Reducer
BitString
Edge A
Edge B
Stage 1
Output edge by 'src' ID
Copy current bitstrings from HDFS
Update bitstrings, set 'changed' flag
Save new bitstrings
current bitstrings
new bitstrings
Stage 2
Output 1 if 'changed'. Output 0 otherwise
Count the total number of 'changed' nodes
new bitstrings
None of the nodes changed?
If no, stop iterating

Figure 4.2: One iteration of HADI-naive. First stage: Bitstrings of all nodes are sent to every reducer. Second stage: sums up the count of changed nodes. The multiple arrows at the beginning of Stage 2 mean that there may be many machines containing bitstrings.

- **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 graph. HADI-plain iteratively runs three-stage MAPREDUCE jobs until all bitstrings of all nodes stop changing. Algorithm 4.3.3, 4.3.3, 4.3.3 shows HADI-plain. We use h for the current iteration number, starting 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 a 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 4.3.3 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 bitstring changed or not, and (b) the current iteration number h and the neighborhood value $N(h, i)$ (line 11). This h and $N(h, i)$ are used to calculate the effective radius of nodes after all bitstrings converge, i.e., don't 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 3 can be done by checking the $flag$ field and using Equation (4.2) in Section 4.2. 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

Algorithm 2: HADI Stage 1

```
Input: Edge data $E = \{(i,j)\}$,
Current bitstring $B = \{(i, b(h-1,i))\}$ or
Bitstring Creation Command $BC = \{(i, cmd)\}$
Output: Partial bitstring $B' = \{(i, b(h-1,j))\}$
Stage1-Map(key $k$, value $v$)
begin
    if $(k,v)$ is of type $B$ or $BC$ then
        Output$(k,v)$;
    else if $(k,v)$ is of type $E$ then
        Output$(v,k)$;
end
Stage1-Reduce(key $k$, values $V[]$)
begin
    SRC $\leftarrow$ [];
    for $v \in V$ do
        if $(k,v)$ is of type $BC$ then
            $\hat{b}(h-1,k) \leftarrow$NewFMBitstring();
        else if $(k,v)$ is of type $B$ then
            $\hat{b}(h-1,k) \leftarrow v$;
        else if $(k,v)$ is of type $E$ then
            Add $v$ to $SRC$;
    end
    for $src \in SRC$ do
        Output$(src, \hat{b}(h-1,k))$;
    end
    Output$(k, \hat{b}(h-1,k))$;
end
```

Algorithm 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
begin
    Output($k, v$);
end
Stage2-Reduce(key $k$, values $V$[])
begin
    $b(h, k) \leftarrow 0$;
    for $v \in V$ do
        $b(h, k) \leftarrow b(h, k)$ BIT-OR $v$;
    end
    Update $flag$ of $b(h, k)$;
    Output($k, b(h, k)$);
end
```

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.

4.3.4 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

Algorithm 4: HADI Stage 3

```
Input: Full bitstring $B = \{(i, b(h, i))\}$
Output: Number of changed nodes, Neighborhood $N(h)$
Stage3-Map(key $k$, value $v$)
begin
    Analyze $v$ to get ($changed$, $N(h, i)$);
    Output($key\_for\_changed$,$changed$);
    Output($key\_for\_neighborhood$, $N(h, i)$);
end
Stage3-Reduce(key $k$, values $V[]$)
begin
    $Changed \leftarrow 0$;
    $N(h) \leftarrow 0$;
    for $v \in V$ do
        if $k$ is key_for_changed then
            $Changed \leftarrow Changed + v$;
        else if $k$ is key_for_neighborhood then
            $N(h) \leftarrow N(h) + v$;
    end
    Output($key\_for\_changed$,$Changed$);
    Output($key\_for\_neighborhood$, $N(h)$);
end
```

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 is necessary at each stage. Figure 4.3 shows an example of converting data to block.

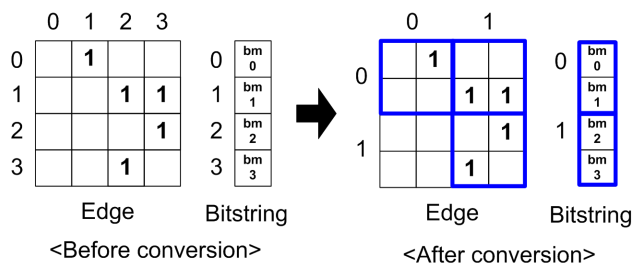


Figure 4.3: Converting the original edge and bitstring to blocks. The 4-by-4 edge and length-4 bitstring are converted to 2-by-2 super-elements and length-2 super-bitstrings. Notice the lower-left super-element of the edge is not produced since there is no nonzero element inside it.

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* takes 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 4.5, where we show results from block-structured graphs (‘Kronecker graphs’ [100]) and from random graphs (‘Erdős-Rényi graphs’ [56]).

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 $K \cdot ((n+m)\log n)$ space, the amount of data increases when K is large. For example, the YahooWeb graph in Section **??** 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 $K * |bitstring|$, 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).

4.4 Analysis and Discussion

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

4.4.1 Time and Space Analysis

We analyze the algorithm complexity of HADI with M machines for a graph G with n nodes and m edges with diameter d. We are interested in the time complexity, as well as the space complexity.

Lemma 2 (Time Complexity of HADI) *HADI takes $O(\frac{d(n+m)}{M} log \frac{n+m}{M})$ time.*

Proof 7 *(Sketch) The Shuffle steps after Stage1 takes $O(\frac{n+m}{M} log \frac{n+m}{M})$ time which dominates the time complexity.*

Notice that the time complexity of HADI is less than previous approaches in Section 4.6($O(n^2 + nm)$, at best). Similarly, for space we have:

Lemma 3 (Space Complexity of HADI) *HADI requires $O((n+m)\log n)$ space.*

Proof 8 *(Sketch) The maximum space* $k \cdot ((n+m)\log n)$ *is required at the output of* `Stage1`*-Reduce. Since k is a constant, the space complexity is* $O((n+m)\log n)$.

4.4.2 HADI in parallel DBMSs

Using relational database management systems (RDBMS) for graph mining is a promising research direction, especially given the findings of [123]. We mention that HADI can be implemented on top of an Object-Relational DBMS (parallel or serial): it needs repeated joins of the edge file with the appropriate file of bit-strings, and a user-defined function for bit-OR-ing. See [81] for details.

4.5 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?

4.5.1 Experimental Setup

We use both real and synthetic graphs in Table 4.2 for our experiments.

- YahooWeb: web pages and their hypertext links indexed by Yahoo! Altavista search engine in 2002.
- Patents: U.S. patents, citing each other (from 1975 to 1999).
- LinkedIn: people connected to other people (from 2003 to 2006).
- Kronecker: Synthetic Kronecker graphs [100] using a chain of length two as the seed graph.

For the performance experiments, we use synthetic Kronecker and Erdős-Rényi graphs. The reason of this choice is that we can generate any size of these two types of graphs, and Kronecker graph 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 values of the corresponding Kronecker graphs. The main difference of Kronecker compared to Erdős-Rényi graphs is the emergence of a block-wise

Graph	Nodes	Edges	File	Description
YahooWeb	1.4 B	6.6 B	116G	page-page
LinkedIn	7.5 M	58 M	1G	person-person
Patents	6 M	16 M	264M	patent-patent
Kronecker	177 K	1,977 M	25G	synthetic
	120 K	1,145M	13.9G	
	59 K	282 M	3.3G	
Erdős-Rényi	177 K	1,977 M	25G	random $G_{n,p}$
	120 K	1,145 M	13.9G	
	59 K	282 M	3.3G	

Table 4.2: Datasets. B: Billion, M: Million, K: Thousand, G: Gigabytes

structure of the adjacency matrix, from its construction [100]. 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 3Tb aggregate RAM, and over 1.5 Peta-byte 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 bit shuffle encoding and block operation.

4.5.2 Running Time and Scale-up

Figure 4.4 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 4.5 gives the throughput $1/T_M$ of HADI-OPT. We also tried HADI with one machine; however it didn't complete, since the machine would take so long that it would often fail in the meanwhile. For this reason, we do not report

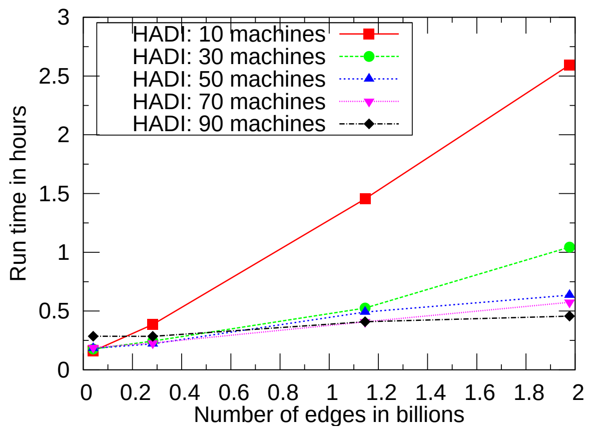


Figure 4.4: Running time versus number of edges with HADI-OPT on Kronecker graphs for three iterations. Notice the excellent scalability: linear on the graph size (number of edges).

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.

4.5.3 Effect of Optimizations

Among the optimizations that we mentioned earlier, which one helps the most, and by how much? Figure 4.1 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**$\times$ 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, as Kronecker graphs do. Also notice that HADI-BLK and HADI-OPT run faster on Kronecker graphs than on Erdős-Rényi graphs of the same size. Again, the reason is that Kronecker graphs have fewer nonzero blocks (i.e., "communities") by their construction, and the "block" operation yields more savings.

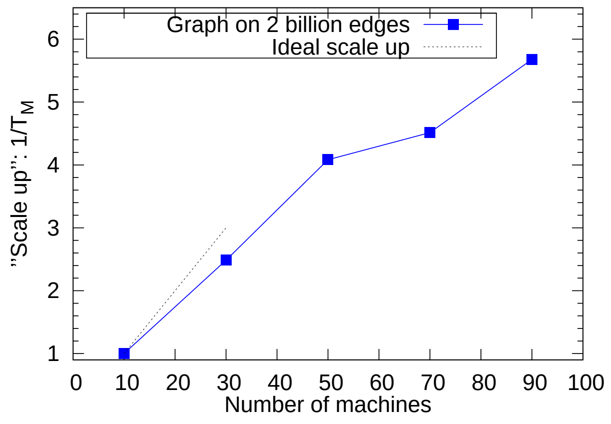


Figure 4.5: "Scale-up" (throughput $1/T_M$) versus number of machines M, for the Kronecker graph (2B edges). Notice the near-linear growth in the beginning, close to the ideal(dotted line).

4.6 Background

We briefly present related works on algorithms for radius and diameter computation, as well as on large graph mining.

Computing Radius and Diameter The typical algorithms to compute the radius and the diameter of a graph include Breadth First Search (BFS) and Floyd's algorithm ([39]). Both approaches are prohibitively slow for large graphs, requiring $O(n^2 + nm)$ and $O(n^3)$ time, where n and m are the number of nodes and edges, respectively. For the same reason, related BFS or all-pair shortest-path based algorithms like [60], [16], [104], [136] can not handle large graphs.

A sampling approach starts BFS from a subset of nodes, typically chosen at random as in [25]. Despite its practicality, this approach has no obvious solution for choosing the representative sample for BFS.

Large Graph Mining There are numerous papers on large graph mining and indexing, mining subgraphs([83], [168], ADI[161], gSpan[166]), graph clustering([130], Graclus [49], METIS [82]), partitioning([40], [30], [50]), tensors([150]), triangle counting([19], [154]), minimum cut([8]), to name a few. However, none of the above computes the diameter of the graph or radii of the nodes.

Large scale data processing using scalable and parallel algorithms has attracted increasing attention due to the needs to process web-scale data. Due to the volume of the data, platforms for this type of processing choose “shared-nothing” architecture. Two promising platforms for such large scale data analysis are (a) MAPREDUCE and (b) parallel RDBMS.

The MAPREDUCE programming framework processes huge amounts of data in a massively parallel way, using thousands or millions commodity machines. It has advantages of (a) fault-tolerance, (b) familiar concepts from functional programming, and (c) low cost of building the cluster. HADOOP, the open source version of MAPREDUCE, is a very promising tool for massive parallel graph mining applications, (e.g., cross-associations [121], connected components [81]). Other advanced MAPREDUCE-like systems include [71], [29], and [124].

Parallel RDBMS systems, including Vertica and Aster Data, are based on traditional database systems and provide high performance using distributed processing and query optimization. They have strength in processing structured data. For detailed comparison of these two systems, see [123]. Again, none of the above articles shows how to use such platforms to efficiently compute the diameter of a graph.

4.7 Conclusions

Our main goal is to develop an open-source package to mine Giga-byte, Tera-byte and eventually Peta-byte networks. We designed HADI, an algorithm for computing radii and diameter of Tera-byte scale graphs, and analyzed large networks to observe important patterns. The contributions of this paper are the following:

- *Design:* We developed 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 ran on the M45 HADOOP cluster of Yahoo, one of the 50 largest supercomputers in the world.
- *Observations:* Thanks to HADI, we could study the diameter and radii distribution of one of the largest public web graphs ever analyzed (over 6 *billion* edges); we also observed the “Small Web” phenomenon, multi-modal/bi-modal radius distributions, and palindrome motions of radius distributions over time in real networks.

Future work includes algorithms for additional graph mining tasks like computing eigenvalues, and outlier detection, for graphs that span Tera- and Petabytes.

Chapter 5

PEGASUS: Mining Peta-Scale Graphs

5.1 Introduction

Graphs are ubiquitous: computer networks, social networks, mobile call networks, the World Wide Web [25], 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.

A promising tool is parallelism, and specifically MAPREDUCE [46] and its open source version, HADOOP. Based on HADOOP, here we describe PEGASUS, a graph mining package for handling graphs with *billions* of nodes and edges. The PEGASUS code and several dataset are at **http://www.cs.cmu.edu/∼pegasus**. 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 5.2 presents the related work. Section 5.3 describes our framework and explains several graph mining algorithms. Section 5.4 discusses optimizations that allow us to achieve significantly faster performance in practice. In Section 5.5 we present timing results and Section 5.6 our findings in real world, large scale graphs. We conclude in Section 5.7.

5.2 Background and Related Work

The related work forms two groups, graph mining, and HADOOP.

Large-Scale Graph Mining. There are a huge number of graph mining algorithms, computing communities (eg., [33], DENGRAPH [57], METIS [82]), subgraph discovery(e.g., GraphSig [127], [83], [73], [34], gPrune [169], gApprox [32], gSpan [166], Subdue [84], HSIGRAM/VSIGRAM [96], ADI [161], CSV [162]), finding important nodes (e.g., PageRank [24] and HITS [87]), computing the number of triangles [154, 155], topic detection [126], attack detection [135], with too-many-to-list alternatives for each of the above tasks. Most of the previous algorithms do not scale, at least directly, to several millions and billions of nodes and edges.

For connected components, there are several algorithms, using Breadth-First Search, Depth-First-Search, "propagation" ([134], [14], [74]), or "contraction" [70] . These works rely on a shared memory model which limits their ability to handle large, disk-resident graphs.

MapReduce and Hadoop. MAPREDUCE is a programming framework [46] [9] 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. Briefly, the programmer needs to provide only two functions, a *map* and a *reduce*. The typical framework is as follows [97]: (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, (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) [1] and PIG, a high level language for data analysis [114]. Due to its power, simplicity and the fact that building a small cluster is relatively cheap, HADOOP is a very promising tool for large scale graph mining applications, something already reflected in academia, see [121]. In addition to PIG, there are several high-level language and environments for advanced MAPREDUCE-like systems, including SCOPE [29], Sawzall [124], and Sphere [71].

5.3 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.

5.3.1 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

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

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`: sum n multiplication results for node i.

3. `assign`: overwrite previous value of v_i with new result to make v'_i.

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

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

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

1. $\texttt{combine2}(m_{i,j}, v_j)$: combine $m_{i,j}$ and v_j.

2. $\texttt{combineAll}_i(x_1, ..., x_n)$: combine all the results from `combine2`() for node i.

3. $\texttt{assign}(v_i, v_{new})$: decide how to update v_i with v_{new}.

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 $\texttt{combineAll}_i$() 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 following SQL statement - we assume that we have (built-in or user-defined) functions $\texttt{combineAll}_i$() 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.

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

5.3.2 GIM-V and PageRank

Our first application of GIM-V is PageRank, a famous algorithm that was used by Google to calculate relative importance of web pages [24]. The PageRank vector p of n web pages satisfies the following eigenvector equation:

$$p = (cE^T + (1-c)U)p$$

where c is a damping factor (usually set to 0.85), E is the row-normalized adjacency matrix (source, destination), and U is a 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 p^{next} is calculated by $p^{next} = (cE^T + (1-c)U)p^{cur}$. 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 by column-normalize E^T such that every column of M sum to 1. 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}

5.3.3 GIM-V and Random Walk with Restart

Random Walk with Restart(RWR) is an algorithm to measure the proximity of nodes in graph [116]. In RWR, the proximity vector r_k from node k satisfies the equation:

$$r_k = cMr_k + (1-c)e_k$$

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 [116]. M is a column-normalized and transposed adjacency matrix, as in Section 5.3.2. In GIM-V, RWR is formulated by $r_k^{next} = M \times_G r_k^{cur}$ where 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}

5.3.4 GIM-V and Diameter Estimation

HADI [80] is an algorithm to estimate the diameter and radius of large graphs. The diameter of a graph is the maximum of the length of the shortest path between every pair of nodes. The radius of a node v_i is the number of hops that we need to reach the farthest-away node from v_i. The main idea of HADI is as follows. For each node v_i in the graph, we maintain the number of neighbors reachable from v_i within h hops. As h increases, the number of neighbors increases until h reaches it maximum value. The diameter is h where the number of neighbors within $h+1$ does not increase for every node. For further details and optimizations, see [80].

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 BITWISE-OR $\{b_k^h \mid (i,k) \in E\}$

In GIM-V, the bitstring update of HADI is represented by

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

where M is an 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.

5.3.5 `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 an 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 8 (Upper bound of iterations in HCC) HCC *requires maximum d iterations where d is the diameter of the graph.*

Proof 9 *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 a small number of iterations.

5.4 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`. After that we propose several faster methods for `GIM-V`.

5.4.1 `GIM-V` BASE: Naive Multiplication

`GIM-V` BASE is a two-stage algorithm whose pseudo code is in Algorithm 5.4.1 and 5.4.1. 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 16 of `Stage2`, where the "self" and "others" tags in line 16 and line 22 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 5.4.1 and 5.4.1, Output(k, v) means to output data with the key k and the value v.

5.4.2 `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

Algorithm 1 GIM-V BASE Stage 1.

Input: Matrix $M = \{(id_{src}, (id_{dst}, mval))\}$, Vector $V = \{(id, vval)\}$
Output: Partial vector $V' = \{(id_{src}$, combine2$(mval, vval)\}$

```
1: Stage1-Map(Key k, Value v):
2: if $(k, v)$ is of type V then
3:     Output$(k, v)$;                                   // (k: $id$, v: $vval$)
4: else if $(k, v)$ is of type M then
5:     $(id_{dst}, mval) \leftarrow v$;
6:     Output$(id_{dst}, (k, mval))$;                    // (k: $id_{src}$)
7: end if
8:
9: Stage1-Reduce(Key k, Value v[1..m]):
10: $saved\_kv \leftarrow [\ ]$;
11: $saved\_v \leftarrow [\ ]$;
12: for $v \in v[1..m]$ do
13:     if $(k, v)$ is of type V then
14:         $saved\_v \leftarrow v$;
15:         Output$(k, (\text{"}self\text{"}, saved\_v))$;
16:     else if $(k, v)$ is of type M then
17:         Add $v$ to $saved\_kv$;                      // (v: $(id_{src}, mval)$)
18:     end if
19: end for
20: for $(id'_{src}, mval') \in saved\_kv$ do
21:     Output$(id'_{src}$, ("$others$",combine2$(mval', saved\_v)))$;
22: end for
```

Algorithm 2 GIM-V BASE Stage 2.

Input: Partial vector $V' = \{(id_{src}, vval')\}$
Output: Result Vector $V = \{(id_{src}, vval)\}$

```
1: Stage2-Map(Key k, Value v):
2: Output$(k, v)$;
3:
4: Stage2-Reduce(Key k, Value v[1..m]):
5: $others\_v \leftarrow [\ ]$;
6: $self\_v \leftarrow [\ ]$;
7: for $v \in v[1..m]$ do
8:     $(tag, v') \leftarrow v$;
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$_k(others\_v)))$;
```

grouping, GIM-V is performed on blocks, not on individual elements. GIM-V BL is illustrated in Figure 5.1.

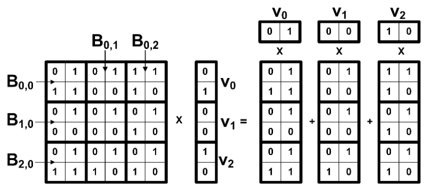


Figure 5.1: GIM-V BL using 2 x 2 blocks. $B_{i,j}$ represents a matrix block, and v_i represents a vector block. The matrix and vector are joined block-wise, not element-wise.

In our experiment at Section 5.5, 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 decrease 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 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 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 $2 \times logb$ 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 5.5).

5.4.3 GIM-V CL: Clustered Edges

When we use block multiplication, another advantage is that we can benefit from clustered edges. As can be seen from Figure 5.2, 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 [121]). 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 is only useful when combined with block encoding. If every element is treated separately, then clustered edges don't help anything for the performance of GIM-V.

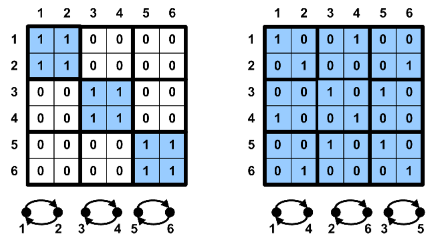


Figure 5.2: Clustered vs. non-clustered adjacency matrices for two isomorphic graphs. The edges are grouped into 2 by 2 blocks. The left graph uses only 3 blocks while the right graph uses 9 blocks.

5.4.4 GIM-V DI: Diagonal Block Iteration

As mentioned in Section 5.4.2, the main bottleneck of GIM-V is its shuffling and disk I/O steps. Since GIM-V iteratively runs Algorithm 5.4.1 and 5.4.1, 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 much 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 5.3.

We see that in Figure 5.3 (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

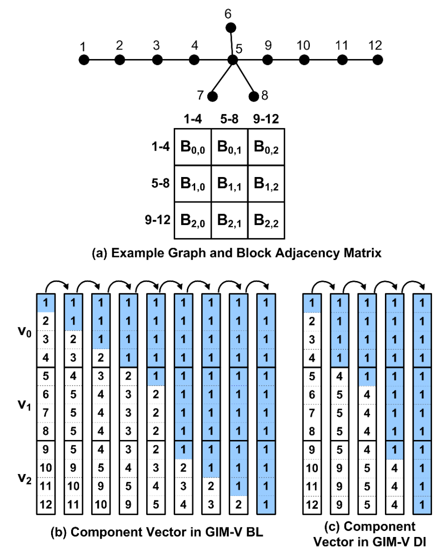

(a) Example Graph and Block Adjacency Matrix
(b) Component Vector in GIM-V BL
(c) Component Vector in GIM-V DI

Figure 5.3: 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.

{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 are determined by the following theorem.

Theorem 9 (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 10 *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.*

Algorithm 3 Renumbering the minimum node

```
Input: Edge $E = \{(id_{src}, id_{dst})\}$,
    current minimum node id $minid_{cur}$,
    new minimum node id $minid_{new}$
Output: Renumbered Edge $V = \{(id'_{src}, id'_{dst})\}$
 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$(src, dst)$;
```

5.4.5 GIM-V NR: Node Renumbering

In HCC, the minimum node id is propagated to the other parts of the network within at most d steps, where d is the diameter of the network. If the node with the minimum id(which we call 'minimum node') is located at the center of the network, 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 [80] 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 highest-degree nodes. This heuristic is based on the fact that nodes with large degree have small radii [80]. Moreover, computing the degree of very large graphs is trivial in MAPREDUCE and could be performed quickly with one job.

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 5.4.5. Since the renumbering requires only filtering, it can be done with a Map-only job.

5.4.6 Analysis

We analyze the time and space complexity of GIM-V. In the theorems below, M is the number of machines.

Theorem 10 (Time Complexity of GIM-V) *One iteration of* GIM-V *takes* $O(\frac{V+E}{M} log \frac{V+E}{M})$ *time.*

Proof 11 *The running time is dominated by the sorting time for* $\frac{V+E}{M}$ *records, which is* $O(\frac{V+E}{M} log \frac{V+E}{M})$.

Theorem 11 (Space Complexity of GIM-V) GIM-V *requires* $O(V+E)$ *space.*

Proof 12 *We assume the value of the elements of the input vector* v *is constant. Then the theorem 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.*

5.5 Performance and Scalability

We do experiments to answer following questions:

Q1 How does GIM-V scale up?

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

The graphs we used in our experiments at Section 5.5 and 5.6 are described in Table 5.1 [1].

We run PEGASUS in M45 HADOOP cluster by Yahoo! and our own cluster composed of 9 machines. M45 is one of the top 50 supercomputers in the world with 1.5 Pb total storage and 3.5 Tb memory. For the performance and scalability experiments, we used synthetic Kronecker graphs [100] since we can generate them with any size, and they are one of the most realistic graphs among synthetic graphs.

[1] Wikipedia: http://www.cise.ufl.edu/research/sparse/matrices/
Kronecker, DBLP: http://www.cs.cmu.edu/∼pegasus
YahooWeb, LinkedIn: released under NDA.
Flickr, Epinions, patent: not public data.

Name	Nodes	Edges	Description
YahooWeb	1,413 M	6,636 M	WWW pages in 2002
LinkedIn	7.5 M	58 M	person-person in 2006
	4.4 M	27 M	person-person in 2005
	1.6 M	6.8 M	person-person in 2004
	85 K	230 K	person-person in 2003
Wikipedia	3.5 M	42 M	doc-doc in 2007/02
	3 M	35 M	doc-doc in 2006/09
	1.6 M	18.5 M	doc-doc in 2005/11
Kronecker	177 K	1,977 M	synthetic
	120 K	1,145 M	synthetic
	59 K	282 M	synthetic
	19 K	40 M	synthetic
WWW-Barabasi	325 K	1,497 K	WWW pages in nd.edu
DBLP	471 K	112 K	document-document
flickr	404 K	2.1 M	person-person
Epinions	75 K	508 K	who trusts whom

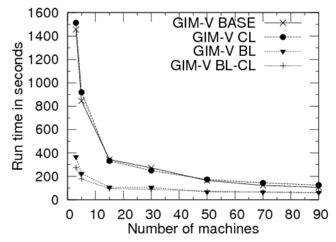
Table 5.1: Order and size of networks.

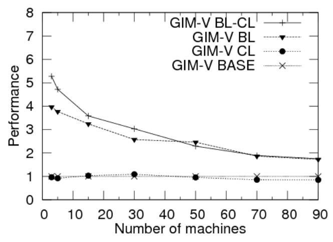
5.5.1 Results

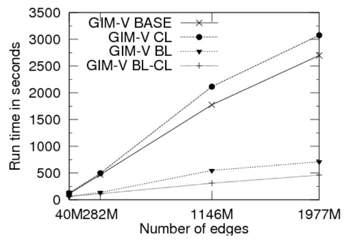
We first show how the performance of our method changes as we add more machines. Figure 5.4 shows the running time and performance of GIM-V for PageRank with Kronecker graph of 282 million edges, and size 32 blocks if necessary.

In Figure 5.4 (a), for all of the methods the running time decreases as we add more machines. Note that clustered edges(GIM-V CL) didn't help performance unless it is combined with block encoding. When it is combined, however, it showed the best performance (GIM-V BL-CL).

In Figure 5.4 (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.







(a) Running time vs. Machines(b) Performance vs. Machines(c) Running time vs. Edges

Figure 5.4: 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.

Next we show how the performance of our methods changes as the input size grows. Figure 5.4 (c) shows the running time of GIM-V with different number of edges under 10 machines. As we can see, all of the methods 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 new 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 5.5, 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 same. Therefore, the diagonal block iteration method decrease 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 5.6 shows the degree distribution of LinkedIn. Without renumbering, the minimum node has degree 1, which is not surprising 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 5.7. 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 5.8 and 5.9. The original minimum node has degree 1, and the number of iterations decreased to 83 % of the original after renumbering.

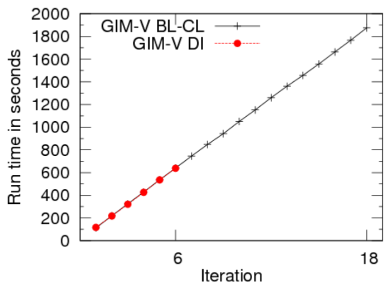


Figure 5.5: Comparison of GIM−V DI and GIM−V BL-CL for HCC. GIM−V DI finishes in 6 iterations while GIM−V BL-CL finishes in 18 iterations due to long chains.

5.6 GIM−V At Work

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

5.6.1 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 5.10 show the evolution of connected components of LinkedIn and Wikipedia data. Figure 5.11 show the distribution of connected components in the YahooWeb graph. We have following observations.

Power Law Tails in Connected Components Distributions We observed power law relation of count and size of small connected components in Figure 5.10(a),(b) and Figure 5.11. This reflects that the connected components in real networks are formed by processes similar to Chinese Restaurant Process and Yule distribution [113].

Stable Connected Components After Gelling Point In Figure 5.10(a), the distribution of connected components remain stable after a 'gelling' point[107] at year 2003.We can see that the slope of tail distribution do not change after year 2003. We observed the same phenomenon in Wikipedia graph in Figure 5.10 (b). The graph show stable tail slopes from the beginning, since the network were

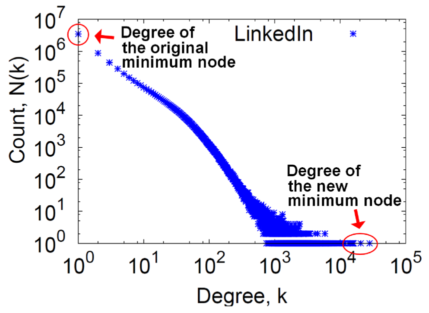


Figure 5.6: Degree distribution of LinkedIn. Notice that the original minimum node has degree 1, which is highly probable given the power-law behavior of the degree distribution. After the renumbering, the minimum node is replaced with a highest-degree node.

already mature in year 2005.

Absorbed Connected Components and Dunbar's number In Figure 5.10(a), we find two large connected components in year 2003. However it became 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 5.10 (a) and (b). This agrees with the observation[107] 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[54], which says that the maximum community size in social networks is roughly 150.

Anomalous Connected Components In Figure 5.11, 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 porn 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.

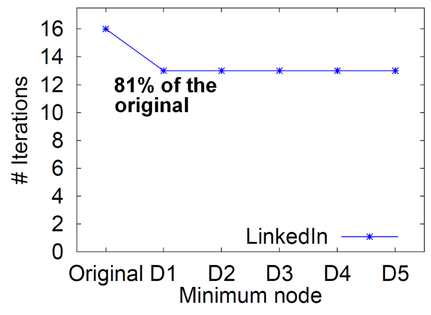


Figure 5.7: Number of iterations vs. the minimum node of LinkedIn, for connected components. Di represents the node with i-th largest degree. Notice that the number of iterations decreased by 19 % after renumbering.

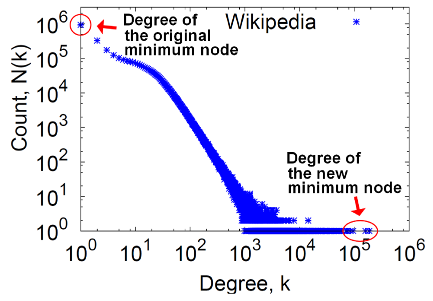


Figure 5.8: Degree distribution of Wikipedia. Notice that the original minimum node has degree 1, as in LinkedIn. After the renumbering, the minimum node is replaced with a highest-degree node.

5.6.2 PageRank scores of Real Networks

We analyzed the PageRank scores of the nodes of real graphs, using PEGASUS. Figure 5.12 and 5.13 show the distribution of the PageRank scores for the Web graphs, and Figure 5.14 shows the evolution of PageRank scores of the LinkedIn and Wikipedia graphs. We have the following observations.

Power Laws in PageRank Distributions In Figure 5.12, 5.13, and 5.14, we observe power-law relations between the PageRank score and the number of nodes with such PageRank. Pandurangan et. al.[117] 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

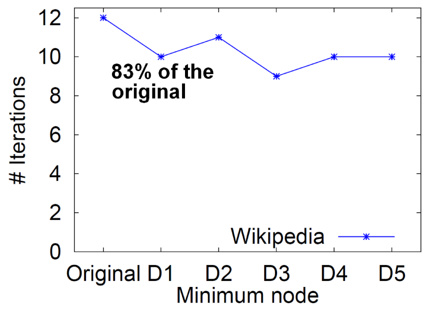


Figure 5.9: Number of iterations vs. the minimum node of Wikipedia, for connected components. Di represents the node with i-th largest degree. Notice that the number of iterations decreased by 17 % after renumbering.

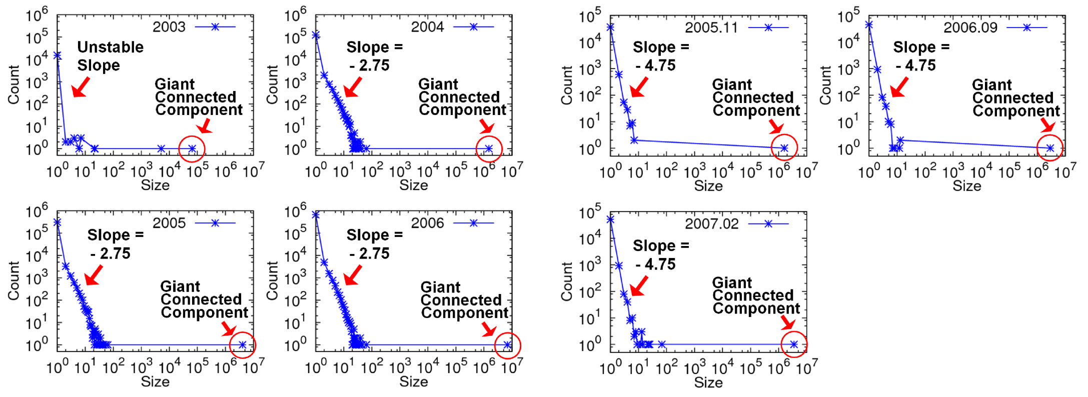
snapshot of the Internet. The top 3 highest PageRank sites for the year 2002 are `www.careerbank.com`, `access.adobe.com`, and `top100.rambler.ru`. As expected, they have huge in- degrees (from $\approx$70K to $\approx$70M).

PageRank and the Gelling Point In the LinkedIn network (see Figure 5.14 (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 5.14 (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.

5.6.3 Diameter of Real Network

We analyzed the diameter and radius of real networks with PEGASUS. Figure 5.15 shows the radius plot of real networks. We have following observations:

Small Diameter For all the graphs in Figure 5.15, the average diameter was less than 6.09. This means that the real world graphs are well connected.



(a) Connected Components of LinkedIn (b) Connected Components of Wikipedia

Figure 5.10: 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($\approx$ 150) and the slope of the tail remains constant after the gelling point at year 2003. (b) . As in LinkedIn, notice the growth of giant connected component and the constant slope for tails.

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 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 power law. Therefore, the first mode is at radius zero which comes from one-node components; second mode(e.g., at radius 6 in Epinion) comes from the giant connected component.

5.7 Conclusions

In this paper we proposed PEGASUS, a graph mining package for very large graphs using the HADOOP architecture. The main contributions are followings:

- We identified the common, underlying primitive of several graph mining operations, and we showed that it is a generalized form of a matrix-vector mul-

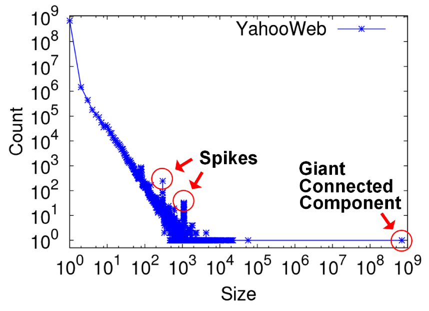


Figure 5.11: Connected Components of YahooWeb. Notice the two anomalous spikes which are far from the constant-slope tail.

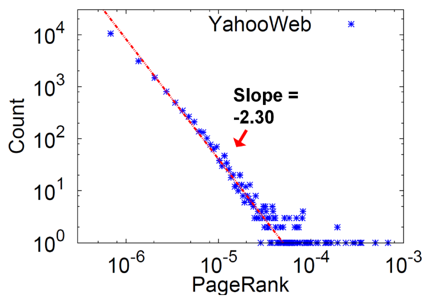


Figure 5.12: PageRank distribution of YahooWeb. The distribution follows power law with exponent 2.30.

tiplication. 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 proposed several optimizations (block-multiplication, diagonal block iteration, node renumbering etc) and reported the winning combination, which achieves more than *5 times* faster performance to the naive implementation.

- We implemented PEGASUS and ran it on M45, one of the 50 largest supercomputers in the world (3.5 Tb memory, 1.5Pb disk storage). Using PEGASUS and our optimized Generalized Iterative Matrix-Vector multipli-

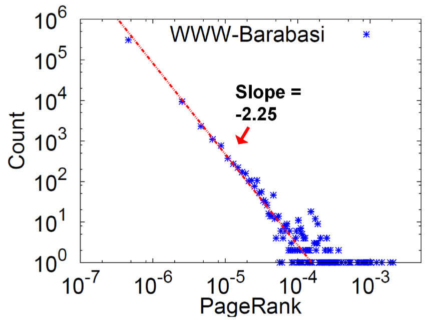
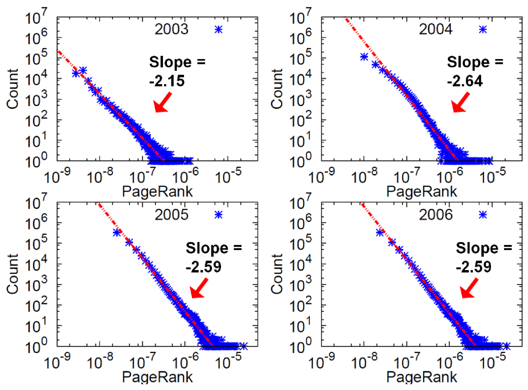


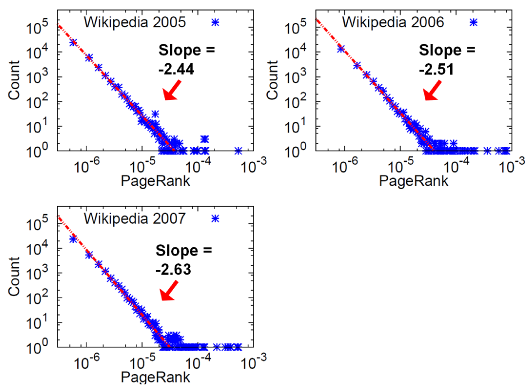
Figure 5.13: PageRank distribution of WWW-Barabasi. The distribution follows power law with exponent 2.25.

> cation variants, we analyzed 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.

Other open source libraries such as HAMA (Hadoop Matrix Algebra) [2] can benefit significantly from PEGASUS. One major research direction is to add to PEGASUS an eigensolver, which will compute the top k eigenvectors and eigenvalues of a matrix. Another directions includes tensor analysis on HADOOP ([93]), and inferences of graphical models in large scale.



(a) PageRanks of LinkedIn



(b) PageRanks of Wikipedia

Figure 5.14: The evolution of PageRanks.(a) The distributions of PageRanks follows 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 power-law. Since the gelling point is before year 2005, the three plots shows similar characteristics: the maximum PageRanks and the slopes are similar.

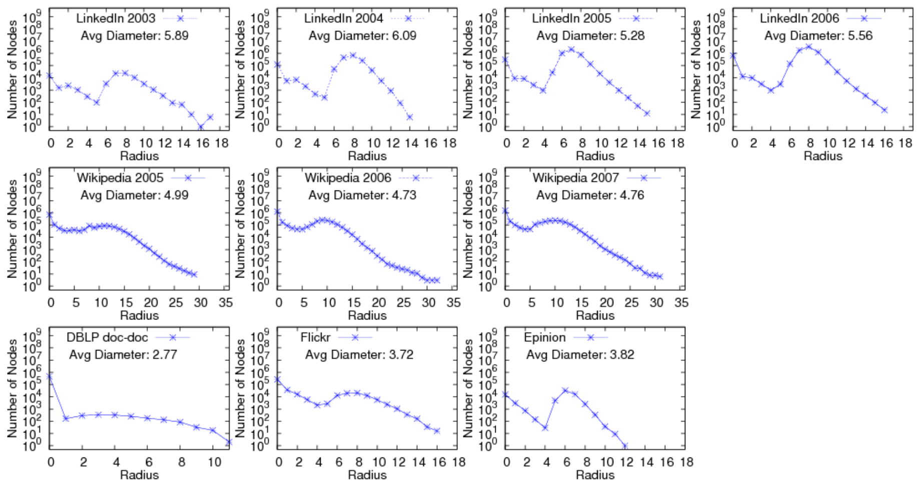


Figure 5.15: Radius of real graphs. X axis: radius. Y axis: number of nodes. (Row 1) LinkedIn from 2003 to 2006. (Row 2) Wikipedia from 2005 to 2007. (Row 3) DBLP, flickr, Epinion. Notice that all the radius plots have the bimodal structure due to many smaller connected components(first mode) and the giant connected component(second mode).

Chapter 6

Two heads better than one: Pattern Discovery in Time-evolving Multi-Aspect Data

6.1 Introduction

Data streams have received attention in different communities. In the standard stream model, each value is associated with a $(\text{timestamp}, \text{stream-ID})$ pair. However, the stream-ID itself may have some additional structure. For example, it may be decomposed into $(\text{location-ID}, \text{type}) \equiv$ stream-ID. We call each such component of the stream model an *aspect*. The number of discrete values each aspect may take is called its *dimensionality*, e.g., a location aspect has dimensionality 2, where the individual dimensions are longitude and latitude. Figure 6.1 illustrates the structure within a stream model. This additional structure should not be ignored in data exploration tasks since it may provide additional insights. Motivated by the idea that the typical "flat-world" view may not be sufficient. How should we summarize such high dimensional and multi-aspect streams? Some of the recent developments are along these lines such as Dynamic Tensor Analysis [142] and Window-based Tensor Analysis [141], which incrementalize the standard offline tensor decompositions such as Tensor PCA (Tucker 2) and Tucker. However, existing literature adopts the same model for all aspects. Specifically, PCA-like operation is performed on each aspect to project data onto maximum variance subspaces. Yet, different aspects have different characteristics, which often require different models. For example, maximum variance summarization is good for

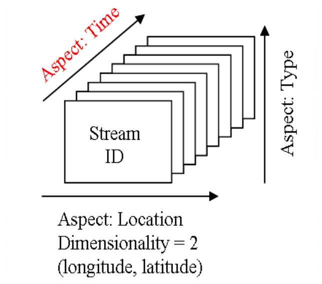


Figure 6.1: A stream model contains multiple aspects such as time, location, and type. Each aspect has a dimensionality, which indicates the number of discrete values it may take.

correlated streams such as correlated readings on sensors in a vicinity; time and frequency based summarizations such as Fourier and wavelet analysis are good for the time aspect due to the temporal dependency and seasonality. In this paper, we propose a *2-heads Tensor Analysis* (2Heads) to allow more than one model or summarization scheme on dynamic tensors. In particular, 2Heads adopts a time-frequency based summarization, namely wavelet transform, on the time aspect and a maximum variance summarization on all other aspects. As shown in experiments, this hybrid approach provides a powerful mining tool to study dynamic tensors, and also outperforms all the others in both space and speed.

Contributions Our proposed approach, 2Heads, provides a general framework of mining and compression for multi-aspect streams. 2Heads has the following key properties:

- *Multi-model summarization:* It engages multiple summarization schemes on various aspects of dynamic tensors.
- *Streaming scalability:* It is fast, incremental and scalable for the streaming environment.
- *Error Guarantee:* It can efficiently compute the approximation error based on the orthogonality property of the models.
- *Space efficiency:* It provides an accurate approximation which achieves very high compression ratios (over 20:1), on all real-world data in our experiments.

We demonstrate the efficiency and effectiveness of our approach in discovering and tracking the key patterns and compressing dynamic tensors on real environmental sensor data.

6.2 Related Work

Tensor Mining: Vasilescu and Terzopoulos [157] introduced the tensor singular value decomposition for face recognition. Xu et al. [165] formally presented the tensor representation for principal component analysis and applied it for face recognition. Kolda et al. [91] apply PARAFAC on Web graphs to generalize the hub and authority scores for Web ranking through term information. Acar et al. [3] applied different tensor decompositions, including Tucker, to the problem of discussion in online chatrooms. Chew et al [35] uses PARAFAC2 to study the multi-language translation probem. J.-T. Sun et al. [144] used Tucker to analyze Web site click-through data. J. Sun et al. [142, 141] have written a pair of papers on dynamically updating a Tucker approximation, with applications ranging from text analysis to environmental and network modeling. All the aforementioned methods share a common characteristic: they assume one type of model for all modes/aspects.

Wavelet: The discrete wavelet transform (DWT) [41] has been proved to be a powerful tool for signal processing, like time series analysis and image analysis [125].

Wavelets have an important advantage over the Discrete Fourier transform (DFT): they can provide information from signals with both periodicities and occasional spikes (where DFT fails). Moreover, wavelets can be easily extended to operate in a streaming, incremental setting [66] as well as for stream mining [120]. However, none of them work on high-order data as we do.

6.3 Background

Principal Component Analysis:

PCA finds the best linear projections of a set of high dimensional points to minimize least-squares cost. More formally, given n points represented as row vectors $\mathbf{x}_i|_{i=1}^n \in \mathbb{R}^N$ in an N dimensional space, PCA computes n points $\mathbf{y}_i|_{i=1}^n \in \mathbb{R}^r$ ($r \ll N$) in a lower dimensional space and the factor matrix $\mathbf{U} \in \mathbb{R}^{N \times r}$ such that the least-squares cost $e = \sum_{i=1}^n \|\mathbf{x}_i - \mathbf{U}\mathbf{y}_i\|_2^2$ is minimized.[1]

Discrete Wavelet Transform: The key idea of wavelets is to separate the input sequence into low frequency part and high frequency part and to do that recursively in different scales. In particular, the *discrete wavelet transform* (DWT)

[1]Both $\mathbf{x}$ and $\mathbf{y}$ are row vectors.

over a sequence $\mathbf{x} \in \mathbb{R}^N$ gives a N wavelet coefficients which encode the averages (low frequency parts) and differences (high frequency parts) at all $\lg N+1$ levels.

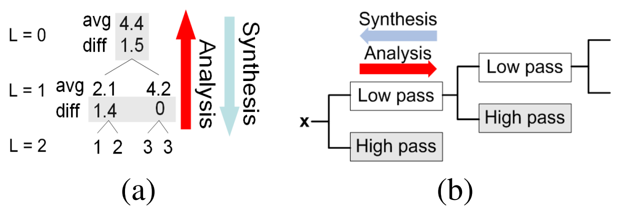


Figure 6.2: Example: (a) Haar wavelet transform on $\mathbf{x} = (1, 2, 3, 3)^\mathsf{T}$. The wavelet coefficients are highlighted in the shaded area. (b) the same process can be viewed as passing $\mathbf{x}$ through two filter banks.

In the matrix presentation, the *analysis* step is

$$\mathbf{b} = \mathbf{A}\mathbf{x} \tag{6.1}$$

where $\mathbf{x} \in \mathbb{R}^N$ is the input vector, $\mathbf{b} \in \mathbb{R}^N$ consists of the wavelet coefficients. At i-th level, the pair of low- and high-pass filters, formally called filter banks, can be represented as a matrix, say $\mathbf{A}_i$. For the Haar wavelet example in Figure 6.2, the first and second level filter banks $\mathbf{A}_1$ and $\mathbf{A}_0$ are the following, where $r = \frac{1}{\sqrt{2}}$:

$$\mathbf{A}_1 = \begin{bmatrix} r & r & & \\ & & r & r \\ r & -r & & \\ & & r & -r \end{bmatrix} \quad \mathbf{A}_0 = \begin{bmatrix} r & r & & \\ r & -r & & \\ & & 1 & \\ & & & 1 \end{bmatrix}$$

The final analysis matrix $\mathbf{A}$ is a sequence of filter banks applied on the input signal, i.e., $\mathbf{A} = \mathbf{A}_0\mathbf{A}_1$. Conversely, the *synthesis* step is $\mathbf{x} = \mathbf{S}\mathbf{b}$. Note that synthesis is the inverse of analysis, $\mathbf{S} = \mathbf{A}^{-1}$. When the wavelet is orthonormal like Haar wavelet, Daubechies, the synthesis is simply the transpose of analysis, i.e., $\mathbf{S} = \mathbf{A}^\mathsf{T}$.

Multilinear Analysis: A tensor of order M closely resembles a data cube with M dimensions. Formally, we write an M-th order tensor as $\mathcal{X} \in \mathbb{R}^{N_1 \times N_2 \times \cdots \times N_M}$ where N_i $(1 \le i \le M)$ is the *dimensionality* of the i-th mode ("dimension" in OLAP terminology).

Matricization The mode-d matricization of an M-th order tensor $\mathcal{X} \in \mathbb{R}^{N_1 \times N_2 \times \cdots \times N_M}$ is the rearrangement of a tensor into a matrix by keeping index d fixed and flatten the other indices. Therefore, the mode-d matricization $\mathbf{X}_{(d)}$ is in $\mathbb{R}^{N_d \times (\prod_{i \neq d} N_i)}$. The mode-$d$ matricization $\mathcal{X}$ is denoted as $\mathrm{unfold}(\mathcal{X}, d)$ or $\mathbf{X}_{(d)}$. Similarly, the inverse operation is denoted as $\mathrm{fold}(\mathbf{X}_{(d)})$. In particular, we have $\mathcal{X} = \mathrm{fold}(\mathrm{unfold}(\mathcal{X}, d))$.

Mode Product The d-mode product of a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_M}$ with a matrix $\mathbf{A} \in \mathbb{R}^{r \times n_d}$ is denoted as $\mathcal{X} \times_d \mathbf{A}$ which is defined element-wise as

$$(\mathcal{X} \times_d \mathbf{A})_{i_1 \ldots i_{d-1} j i_{d+1} \ldots i_M} = \sum_{i_d=1}^{n_d} x_{i_1 \times i_2 \times \cdots \times i_M} a_{j i_d}$$

The process is equivalent to a three step procedure: first we matricize $\mathcal{X}$ along mode-1, then we multiply it with $\mathbf{U}$, and finally we fold the result back as a tensor. In general, a tensor $\mathcal{Y} \in \mathbb{R}^{r_1 \times \cdots \times r_M}$ can multiply a sequence of matrices $\mathbf{U}^{(i)}|_{i=1}^{M} \in \mathbb{R}^{n_i \times r_i}$ as: $\mathcal{Y} \times_1 \mathbf{U}_1 \cdots \times_M \mathbf{U}_M \in \mathbb{R}^{n_1 \times \cdots \times n_M}$, which can be compactly written as $\mathcal{Y} \prod_{i=1}^{M} \times_i \mathbf{U}_i$. Furthermore, the notation for $\mathcal{Y} \times_1 \mathbf{U}_1 \cdots \times_{i-1} \mathbf{U}_{i-1} \times_{i+1} \mathbf{U}_{i+1} \cdots \times_M \mathbf{U}_M$ (i.e. multiplication of all $\mathbf{U}_j$s except the i-th) is simplified as $\mathcal{Y} \prod_{j \neq i} \times_j \mathbf{U}_j$.

Tucker Decomposition Given a tensor $\mathcal{X} \in \mathbb{R}^{N_1 \times N_2 \times \cdots \times N_M}$, we can perform a higher-order principal component analysis so that the tensor is decomposed into a core tensor and a set of factor matrices. Formally, we can reconstruct $\mathcal{X}$ using a sequence of mode products between the core tensor $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times \cdots \times R_M}$ and the matrices $\mathbf{U}^{(i)} \in \mathbb{R}^{N_i \times R_i}|_{i=1}^{M}$. We use the following standard notation for Tucker decomposition:

$$\mathcal{X} = \mathcal{G} \prod_{i=1}^{M} \times_i \mathbf{U}^{(i)} \equiv [\![\mathcal{G} \,; \mathbf{U}^{(i)}|_{i=1}^{M}]\!]$$

We will refer to the decomposed tensor $[\![\mathcal{G} \,; \mathbf{U}^{(i)}|_{i=1}^{M}]\!]$ as a Tucker Tensor. If a tensor $\mathcal{X} \in \mathbb{R}^{N_1 \times N_2 \times \cdots \times N_M}$ can be decomposed (even approximately), the storage space can be reduced from $\prod N_{i=1}^{M}$ to $\prod R_{i=1}^{M} + \sum_{i=1}^{M} (N_i \times R_i)$, see Figure 6.3.

6.4 Problem Formulation

In this section, two problems addressed in this paper are formally defined: *Static 2-heads tensor mining* and *Dynamic 2-heads tensor mining*. To facilitate the discussion, we refer all aspects except for the time aspect as "spatial aspects."

6.4.1 Static 2-heads tensor mining

In the static case, the data are represented as a single tensor $\mathcal{D} \in \mathbb{R}^{W \times N_1 \times N_2 \times \cdots \times N_M}$. Notice the first mode corresponds to the time aspect which is qualitatively different from the spatial aspects. The mining goal is to compress the tensor $\mathcal{D}$ while revealing the underlying patterns on both temporal and spatial aspects. More specifically, we define the problem as the follows:

Problem 1 (Static tensor mining) *Given a tensor $\mathcal{D} \in \mathbb{R}^{W \times N_1 \times N_2 \times \cdots \times N_M}$, find the Tucker tensor $\hat{\mathcal{D}} \equiv [\![\mathcal{G} \; ; \mathbf{U}^{(i)}|_{i=0}^{M}]\!]$ such that 1) both the space requirement of $\hat{\mathcal{D}}$ and the reconstruction error $e = \left\| \mathcal{D} - \hat{\mathcal{D}} \right\|_F / \| \mathcal{D} \|_F$ are small; 2) both spatial and temporal patterns are reviewed through the process.*

The central question is how to construct the suitable Tucker tensor, more specifically, what model to be used on each mode. As we will show shortly, different models on time and spatial aspects can serve much better for time-evolving applications.

The intuition of Problem 1 is illustrated in Figure 6.3. The mining operation aims at compressing the original tensor $\mathcal{D}$ and revealing patterns. Both goals are achieved through the specialized Tucker decomposition, *2-heads Tensor Analysis* (2Heads) as presented shortly in Section 6.5.1.

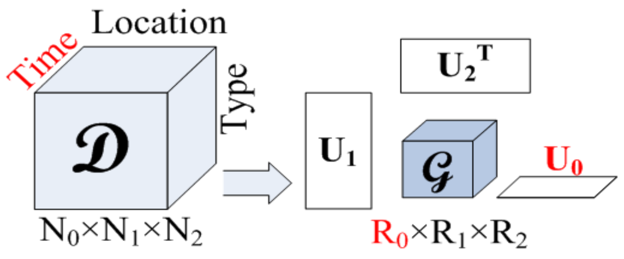


Figure 6.3: The input tensor $\mathcal{D} \in \mathbb{R}^{W \times N_1 \times N_2 \times \cdots \times N_M}$ (time-by-location-by-type) is approximated by a Tucker tensor $[\![\mathcal{G} \; ; \mathbf{U}^{(i)}|_{i=0}^{2}]\!]$. Note that the time mode will be treated differently to the rest as shown later.

6.4.2 Dynamic 2-heads tensor mining

In the dynamic case, the input data are evolving along time aspect. More specifically, given a dynamic tensor $\mathcal{D} \in \mathbb{R}^{n \times N_1 \times N_2 \times \cdots \times N_M}$, the size of time aspect (first mode) n is increasing over time $n \to \infty$. In particular, n is the current time. In another words, new slices along time mode are continuously arriving. To mine the time-evolving data, a time-evolving model is required for space and running time efficiency. In this paper, we adopt a sliding window model which is popular in data stream processing.

Before we formally introduce the problem, two terms have to be defined:

Definition 3 (Time slice) *A time slice $\mathfrak{D}_i$ of $\mathfrak{D} \in \mathbb{R}^{n \times N_1 \times N_2 \times \cdots \times N_M}$ is the i-th slice along the time mode (first mode). In particular, $\mathfrak{D}_n$ is the current slice.*

Note that given a tensor $\mathfrak{D} \in \mathbb{R}^{n \times N_1 \times N_2 \times \cdots \times N_M}$, every time slice is actually a tensor with one less mode, i.e., $\mathfrak{D}_i \in \mathbb{R}^{N_1 \times N_2 \times \cdots \times N_M}$.

Definition 4 (Tensor window) *A tensor window $\mathfrak{D}_{(n,W)}$ consists of a set of the tensor slices ending at time n with size W, or formally,*

$$\mathfrak{D}_{(n,W)} \equiv \{\mathfrak{D}_{n-W+1}, \ldots, \mathfrak{D}_n\} \in \mathbb{R}^{W \times N_1 \times N_2 \times \cdots \times N_M}. \tag{6.2}$$

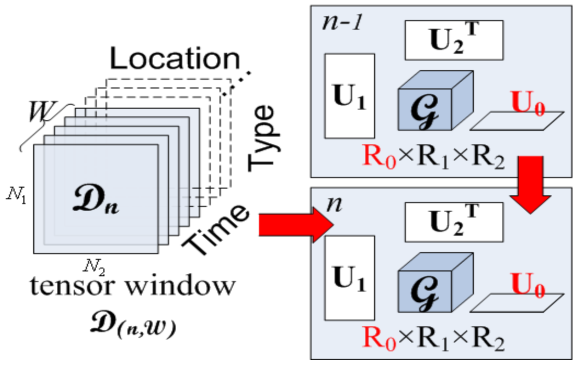


Figure 6.4: Tensor window $\mathfrak{D}_{(n,W)}$ consists of the most recent W time slices in $\mathfrak{D}$. Dynamic tensor mining utilizes the old model for $\mathfrak{D}_{(n-1,W)}$ to facilitate the model construction for new window $\mathfrak{D}_{(n,W)}$.

Figure 6.4 shows an example of tensor window. We now formalize the core problem, *Dynamic 2-heads Tensor Mining*. The goal is to incrementally compress the dynamic tensor while extracting spatial and temporal patterns and their correlations. More specifically, we aim at incrementally maintaining a Tucker model for approximating tensor windows.

Problem 2 (Dynamic 2-heads Tensor Mining) *Given the current tensor window $\mathcal{D}_{(n,W)} \in \mathbb{R}^{W\times N_1 \times N_2 \times \cdots \times N_M}$ and the old Tucker model for $\mathcal{D}(n-1, W)$, find the new Tucker model $\hat{\mathcal{D}}_{(n,W)} \equiv [\![\mathcal{G}\ ; \mathbf{U}^{(i)}|_{i=0}^{M}]\!]$ such that 1)the space requirement of $\hat{\mathcal{D}}$ is small 2)the reconstruction error $e = \left\| \mathcal{D} - \hat{\mathcal{D}} \right\|_F$ is small (see Figure 6.4). 3)both spatial and temporal patterns are reviewed through the process.*

6.5 Multi-model Tensor Analysis

6.5.1 Static 2 Heads Tensor Mining

Many applications as listed before exhibit strong spatio-temporal correlations in the input tensor $\mathcal{D}$. Strong spatial correlation is usually reflected in great benefit on dimensionality reduction. For example, if all temperature measurements in a building exhibit the same trend, PCA can compress the data into one principal component (a single trend is enough to summarize the data). Strong temporal correlation means periodic pattern and long-term dependency. This is better viewed in the frequency domain through Fourier or wavelet transform.

Hence, we propose 2-Heads Tensor Analysis (2Heads), which combines both PCA and wavelet approaches to compress the multi-aspect data by exploiting the spatio-temporal correlation. The algorithm involves three steps:

- *Spatial compression:* we perform alternating minimization on all modes except for time mode.
- *Temporal compression:* we perform discrete wavelet transform on the result of spatial compression.

Spatial compression uses the idea of alternating least square (ALS) method on all projection matrices except for time mode. More specifically, it initializes all projection matrices to be identity matrix $\mathbf{I}$; then it iteratively update the projection matrices of every spatial mode until convergence. The results are the *spatial core tensor* $\mathcal{X} \equiv \mathcal{D} \prod_{i\neq 0} \times_i \mathbf{U}^{(i)}$ and the projection matrices $\mathbf{U}^{(i)}|_{i=1}^{M}$.

Temporal compression perform frequency-based compression (e.g., wavelet transform) on the spatial core tensor $\mathcal{X}$. More specifically, we obtain the *spatio-temporal core tensor* $\mathcal{G} \equiv \mathcal{X} \times_0 \mathbf{U}^{(0)}$ where $\mathbf{U}^{(0)}$ is the DWT matrix such as the one shown in Figure 6.2[2]. The entries in the core tensor $\mathcal{G}$ are the wavelet

[2]$\mathbf{U}^{(0)}$ is never materialized but recursively computed on the fly.

coefficients. We then drop the small entries(coefficients) in G, result denoted as $\hat{\mathcal{G}}$, such that the reconstruction error is just below the error threshold θ. Finally, we obtain Tucker approximation $\hat{\mathcal{D}} \equiv [\![\hat{\mathcal{G}}\ ; \mathbf{U}^{(i)}|_{i=0}^{M}]\!]$. The pseudo-code is listed in Algorithm 6.5.1.

By definition, the error $e = \left\| \mathcal{D} - \hat{\mathcal{D}} \right\|_F^2 / \| \mathcal{D} \|_F^2$. It seems that we need to construct the tensor $\hat{\mathcal{D}}$ and compute the difference between $\mathcal{D}$ and $\hat{\mathcal{D}}$ in order to calculate the error e. Actually, the error e can be computed efficiently based on the following theorem. Computational cost of Algorithm 6.5.1 comes from the mode product and diagonalization, which is $O(\sum_{i=1}^{M}(W \prod_{j<i} R_j \prod_{k\geq i} N_k + N_i^3))$. The dominating factor is usually the mode product. Therefore, the complexity of Algorithm 6.5.1 can be simplified as $O(WM \prod_{i=1}^{M} N_i)$.

Theorem 12 (Error estimation) *Given a tensor $\mathcal{D}$ and its Tucker approximation described in 6.5.1, $\hat{\mathcal{D}} \equiv [\![\mathcal{G}\ ; \mathbf{U}^{(i)}|_{i=0}^{M}]\!]$, we have*

$$e = \sqrt{1 - \left\| \hat{\mathcal{G}} \right\|_F^2 / \| \mathcal{D} \|_F^2} \qquad (6.3)$$

where $\hat{\mathcal{G}}$ is the core tensor after zero-out the small entries and the error estimation $e \equiv \left\| \mathcal{D} - \hat{\mathcal{D}} \right\|_F / \| \mathcal{D} \|_F$.

Proof 13 *See appendix.*

6.5.2 Dynamic 2 Heads Tensor Mining

For the time-evolving case, the idea is to explicitly utilize the overlapping information of the two consecutive tensor windows to update the co-variance matrices $\mathbf{C}^{(i)}|_{i=1}^{M}$. More specifically, given a tensor window $\mathcal{D}_{(n,W)} \in \mathbb{R}^{W\times N_1 \times N_2 \times \cdots \times N_M}$, we aim at removing the effect of the old slice $\mathcal{D}_{n-W}$ and adding in that of the new slice $\mathcal{D}_n$.

This is hard to do because of the iterative optimization. Recall that the ALS algorithm searches for the projection matrices. We approximate the ALS search by updating projection matrices independently on each mode. This is similar to high-order SVD [43]. This process can be efficiently updated by maintaining the co-variance matrices on each mode.

```
Algorithm 1: STATIC 2-HEADS
Input  : a tensor $\mathcal{D} \in \mathbb{R}^{W\times N_1\times N_2\times\cdots\times N_M}$, accuracy $\theta$
Output: a tucker tensor $\hat{\mathcal{D}} \equiv [\![\hat{\mathcal{G}}\ ; \mathbf{U}^{(i)}|_{i=0}^{M}]\!]$
   // search for factor matrices
1  initialize all $\mathbf{U}^{(i)}|_{i=0}^{M} = \mathbf{I}$
2  repeat
3      for $i \leftarrow 1$ to $M$ do
               // project $\mathcal{D}$ to all modes but $i$
4              $\mathcal{X} = \mathcal{D} \prod_{j\neq i} \times_j \mathbf{U}^{(j)}$
               // find co-variance matrix
5              $\mathbf{C}^{(i)} = \mathbf{X}_{(i)}^{\mathsf{T}}\mathbf{X}_{(i)}$
               // diagonalization
6              $\mathbf{U}^{(i)}$ as the eigenvectors of $\mathbf{C}^{(i)}$
       // any changes on factor matrices?
7      if $Tr(||\mathbf{U}^{(i)\mathsf{T}}\mathbf{U}^{(i)}| - \mathbf{I}|) \leq \epsilon$ for $1 \leq i \leq M$ then converged
8  until converged;
   // spatial compression
9  for $i \leftarrow 1$ to $M$ do $\mathcal{X} = \mathcal{D} \prod_{i=1}^{M} \times_i \mathbf{U}^{(i)}$
   // temporal compression
10 $\mathcal{G} = \mathcal{X} \times_0 \mathbf{U}^{(0)}$ where $\mathbf{U}^{(0)}$ is the DWT matrix.
11 $\hat{\mathcal{G}} \approx \mathcal{G}$ by setting all small entries (in absolute value) to zero, s.t.
   $\sqrt{1 - \left\| \hat{\mathcal{G}} \right\|_F^2 / \| \mathcal{D} \|_F^2} \leq \theta$
```

More formally, the co-variance matrix along the ith mode is as follows:

$$\mathbf{C}_{old}^{(i)} = \begin{bmatrix}\mathbf{X}\\ \mathbf{D}\end{bmatrix}^T \begin{bmatrix}\mathbf{X}\\ \mathbf{D}\end{bmatrix} = \mathbf{X}^{\mathsf{T}}\mathbf{X} + \mathbf{D}^{\mathsf{T}}\mathbf{D}$$

where $\mathbf{X}$ is the matricization of the old tensor slice $\mathcal{D}_{n-W}$ and $\mathbf{D}$ is the matricization of tensor window $\mathcal{D}_{(n-1,W-1)}$ (i.e., the overlapping part of tensor window $\mathcal{D}_{(n-1,W)}$ and $\mathcal{D}_{(n,W)}$). Similarly, $\mathbf{C}_{new}^{(i)} = \mathbf{D}^{\mathsf{T}}\mathbf{D} + \mathbf{Y}^{\mathsf{T}}\mathbf{Y}$, where $\mathbf{Y}$ is the matricization of the new tensor $\mathcal{D}_n$. As a result, the update can be easily achieved as follows:

$$\mathbf{C}^{(i)} \leftarrow \mathbf{C}^{(i)} - \mathcal{D}_{\mathbf{N-W}}{}_{(i)}^{\mathsf{T}}\mathcal{D}_{\mathbf{N-W}(i)} + \mathcal{D}_{\mathbf{N}}{}_{(i)}^{\mathsf{T}}\mathcal{D}_{\mathbf{N-W}(i)}$$

where $\mathcal{D}_{\mathbf{N-W}(i)}(\mathcal{D}_{\mathbf{N}(i)})$ is the mode-i matricization of tensor slice $\mathcal{D}_{n-W}(\mathcal{D}_n)$.

The updated projection matrices are just the eigenvectors of the new co-variance matrices. Once the projection matrices are updated, the spatio-temporal compression remains the same. One observation is that Algorithm 6.5.2 can be performed in batches. The only change is to update co-variance matrices involving multiple tensor slices. The batch update can significantly lower the amortized cost for updating projection matrices as well as spatial and temporal compression.

Algorithm 2: DYNAMIC 2-HEADS

```
  Input  : a new tensor window
           $\mathfrak{D}_{(n,W)} = \{\mathfrak{D}_{n-W+1}, \ldots, \mathfrak{D}_n\} \in \mathbb{R}^{W \times N_1 \times N_2 \times \cdots \times N_M}$, old co-variance
           matrices $\mathbf{C}^{(i)}|_{i=1}^{M}$, accuracy $\theta$
  Output: a tucker tensor $\hat{\mathfrak{D}} \equiv [\![\hat{\mathfrak{G}} \,; \mathbf{U}^{(i)}|_{i=0}^{M}]\!]$
1 for $i \leftarrow 1$ to $M$ do
      // update co-variance matrix
2     $\mathbf{C}^{(i)} \leftarrow \mathbf{C}^{(i)} - \mathfrak{D}_{\mathbf{N-W}}{}_{(i)}^{\top} \mathfrak{D}_{\mathbf{N-W}}{}_{(i)} + \mathfrak{D}_{\mathbf{N}}{}_{(i)}^{\top} \mathfrak{D}_{\mathbf{N-W}}{}_{(i)}$
      // diagonalization
3     $\mathbf{U}^{(i)}$ as the eigen-vectors of $\mathbf{C}^{(i)}$
  // spatial compression
4 for $i \leftarrow 1$ to $M$ do $\mathfrak{X} = \mathfrak{D} \prod_{i=1}^{M} \times_i \mathbf{U}^{(i)}$
  // temporal compression
5 $\mathfrak{G} = \mathfrak{X} \times_0 \mathbf{U}^{(0)}$ where $\mathbf{U}^{(0)}$ is the DWT matrix.
6 $\hat{\mathfrak{G}} \approx \mathfrak{G}$ with setting all small entries (in absolute value) to zero, s.t.
  $\sqrt{1 - \left\| \hat{\mathfrak{G}} \right\|_F^2 / \| \mathfrak{D} \|_F^2} \leq \theta$
```

6.5.3 Mining Guide

We now illustrate practical aspects concerning our proposed methods.

The goal of 2Heads is to find high correlated dimensions within the same aspect and across different aspects, and monitor them over time.

Spatial correlation A projection matrix gives the correlation information among dimensions for a single aspect. More specifically, the dimensions of the i-th aspect can be grouped based on their values in the columns of $\mathbf{U}^{(i)}$. The entries with high absolute values in a column of $\mathbf{U}^{(i)}$ correspond to the important dimensions in the same concept. The SENSOR type example shown in Table 6.1 correspond to two concepts in the sensor type aspect — see Section 6.6.1 for details.

Temporal correlation Unlike spatial correlations that reside in the projection matrices, temporal correlation is reflected in the core tensor. After spatial compression, the original tensor is transformed into the spatial core tensor $\mathfrak{X}$ — line 4 of Algorithm 2. Then temporal compression applies on $\mathfrak{X}$ to obtain the (spatio-temporal) core tensor $\mathfrak{G}$ which consists of dominant wavelet coefficients of the spatial core. By focusing the largest entry (wavelet coefficient) in the core tensor, we can easily identify the dominant frequency components in time and space — see Figure 6.6 for more discussion.

Correlations across aspects The interesting aspect of 2Heads is that the core tensor Y provides indications on the correlations of different dimensions across both spatial and temporal aspects. More specifically, a large entry in the core means a high correlation between the corresponding columns in the spatial aspects at specific time and frequency. For example, the combination of Figure 6.5(b), the first concept of Table 6.1 and Figure 6.6(a) gives us the main trend in the data, which is the daily periodic trend of the environmental sensors in a lab.

6.6 Experiment Evaluation

In this section, we will evaluate both mining and compression aspects of 2Heads on real environment sensor data. We first describe the dataset, then illustrate our mining observation on the real data in Section 6.6.1 and finally show some quantitative evaluation in Section 6.6.2.

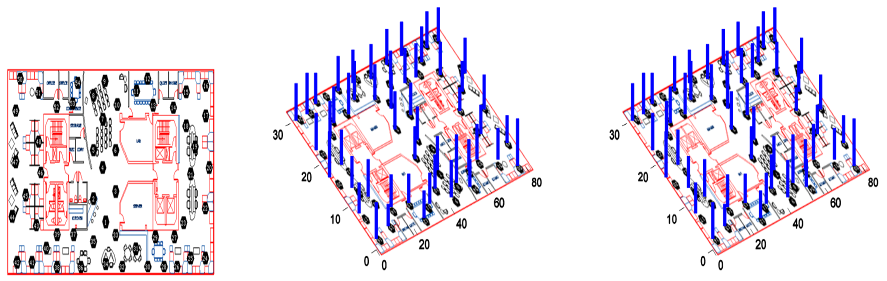
The sensor data consists of voltage, humidity, temperature, and light intensity at the 54 different locations in the Intel Berkeley Lab (see Figure 6.5(a)). It has 1093 timestamps, one for each 30 minutes. The dataset is a $1093 \times 54 \times 4$ tensor corresponding to 22 days of data.

6.6.1 Mining Case-studies

Here, we illustrate how 2Heads can reveal interesting spatial and temporal correlations in sensor data.

Spatial Correlations The SENSOR dataset consists of two spatial aspects, namely, the location and sensor types. Interesting patterns are revealed on both aspects.

For the location aspect, the most dominant trend is scattered uniformly across all locations. As shown in Figure 6.5(b), the weights (the blue bars) on all locations have about the same height. For sensor type aspect, the dominant trend is shown as the 1st concept in Table 6.1. It indicates that 1) the positive correlation among temperature, light intensity and voltage level and 2) negative correlation between humidity and the rest. This corresponds to the regular daily periodic pattern: During the day, temperature and light intensity go up but humidity drops because the A/C is on. During the night, temperature and light intensity drop but humidity increases because A/C is off. The voltage is always positively correlated with the temperature due to the design of MICA2 sensors.



(a) Lab floor map (b) SENSOR Concept 1 (c) SENSOR Concept 2

Figure 6.5: Spatial correlation: Blue bars indicate that positive weights of the corresponding sensors and red bars indicate negative weights. (a) shows the floor plan of the lab, where the numbers indicate the sensor locations. (b) shows the distribution of the most dominant trend, which is more or less uniform. This suggests that all the sensors follow the same pattern over time, which is the daily periodic trend (see Figure 6.6 for more discussion) (c) shows the second most dominant trend, which gives the negative weights to the bottom left corner and positive weights to the rest. It indicates relatively low humidity and temperature measurements because of the vicinity to the A/C.

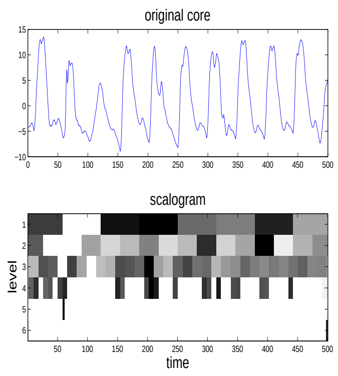
Sensor-Type	voltage	humidity	temperature	light-intensity
concept 1	.16	-.15	.28	.94
concept 2	.6	.79	.12	.01

Table 6.1: SENSOR type correlation

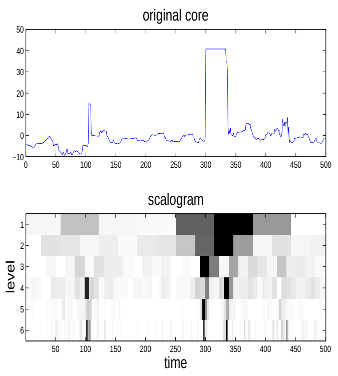
The second strongest trend is shown in Figure 6.5(c) and the 2nd concept in Table 6.1 for the location and type aspects, respectively. The red bars on Figure 6.5(c) indicate negative weights on a few locations close to A/C (mainly at the bottom and left part of the room). This affects the humidity and temperature patterns at those locations. In particular, the 2nd concept has a strong emphasis on humidity and temperature, see the 2nd concept in Table 6.1.

Temporal Correlations Temporal correlation can be best described by frequency-based methods such as wavelet. 2Heads provides a way to capture the global temporal correlation that traditional wavelet cannot capture.

Figure 6.6(a) shows the strongest tensor core stream of SENSOR dataset for the first 500 timestamps and its scalogram of the wavelet coefficients. Large wavelet coefficients (indicated by dark color) concentrate on low frequency part



(a) SENSOR 1st period: "normal"



(b) SENSOR last period: "low battery"

Figure 6.6: SENSOR time-frequency break-down on the dominant components: Notice that the scalogram of (a) only has the low-frequency components (dark color); but the scalogram of (b) has frequency penetration from 300 to 340 due to the sudden shift

(levels 1-3), which correspond to the daily periodic trend in the normal operation.

Figure 6.6(b) shows the strongest tensor core stream of SENSOR dataset for the last 500 timestamps and its corresponding scalogram. Notice large coefficients penetrate all frequency levels from 300 to 350 timestamps due to the erroneous sensor readings caused by low battery level of several sensors

Summary In general, 2Heads provides an effective and intuitive framework to identify both spatial and temporal correlation, which no traditional methods including Tucker and wavelet can do by themselves. Furthermore, 2Heads can track the correlations over time. All the above examples confirmed the great value of 2Heads for mining real-world, high-order data streams.

6.6.2 Quantitative evaluation

In this section, we quantitatively evaluate the proposed methods in both space and CPU cost.

Performance Metrics We use the following three metrics to quantify the mining performance:

Approximation accuracy: This is the key metric that we use to evaluate the quality of the approximation. It is defined as: accuracy $= 1 -$ relative SSE, where

relative SSE is defined as $\|\mathfrak{D} - \hat{\mathfrak{D}}\| / \|\mathfrak{D}\|$.
Space ratio: We use this metric to quantify the required space usage. It is defined as the ratio of the size of the approximation $\hat{\mathfrak{D}}$ and that of the original data $\mathfrak{D}$. Note that the approximation $\hat{\mathfrak{D}}$ is stored in the factorized forms, e.g., Tucker form including core and projection matrices.
CPU time: We use the CPU time spent in computation as the metric to quantify the computational expense. All the experiments are performed on the same dedicated server with four 2.4GHz Xeon CPUs and 48GB memory.

Method Parameters Two parameters affect the quantitative measurements of all the methods: **window size** is the scope of the model in time. For example, window size = 500 means that a model will be built and maintained for most recent 500 timestamps.
step size is the number of timestamps before a new model is constructed.

Methods for comparison The following four methods are compared in experiments: **Tucker**: It performs Tucker2 decomposition on spatial aspects only.

Wavelets: It performs Daubechies-4 compression on every stream. For example, 54×4 wavelet transforms are performed on `Motes` dataset since it has 54×4 stream pairs in `Motes`.

Static 2Heads: It is one of the proposed method in the paper. It uses PCA-like summarization on spatial aspects and wavelet on temporal aspect. The computational cost is similar to the sum of Tucker and wavelet methods.

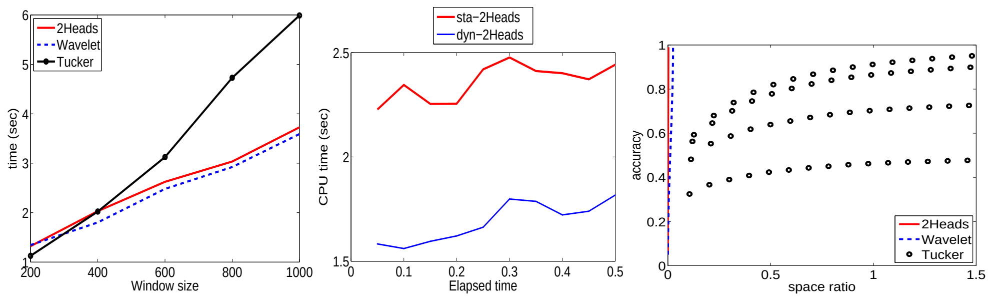
Dynamic 2Heads: It is the main practical contribution of this paper, due to handling efficiently the Dynamic 2Heads Tensor Mining Problem.

Computational efficiency The computation time can be affected by two parameters: window size and step size.

In general the CPU time increases linearly as the window size as shown in Figure 6.7(a).

Wavelets are faster than Tucker, because wavelets perform on individual streams, while Tucker operates on all streams simultaneously. The cost of Static 2Heads is roughly the sum of wavelets and Tucker decomposition, because it performs both spatial and temporal compression in a straightforward manner.

Dynamic 2Heads performs the same functionality as Static 2Heads . But, it is as fast as wavelets by exploiting the computational trick which avoids the computational penalty that static-2Heads has.



(a) Window size (b) Step size (c) Space vs. Accuracy

Figure 6.7: a): (step size is 20% window size): but 2Heads is faster than Tucker and is similar to wavelet. However, 2Heads reveals much more information than wavelet and Tucker without incurring computational penalty. b): Step Size vs. CPU time: (window size 500) Dynamic 2Heads requires much less computational time than Static 2Heads. c): Space vs. Accuracy: 2Heads and wavelet requires much smaller space to achieve high accuracy(e.g., 99%) than Tucker, which indicates the importance temporal aspect. 2Heads is slightly better than wavelet because it captures both spatial and temporal correlations. Wavelet and Tucker only provide partial view of the data.

The computational cost of Dynamic 2Heads increases as the step size, because the overlapping portion between two consecutive tensor windows decreases. Despite that, for all different step sizes, dynamic-2Heads requires much less CPU time as shown in Figure 6.7(b).

Space efficiency The space requirement can be affected by two parameters: approximation accuracy and window size. For all methods, Static and Dynamic 2Heads give comparable results, therefore, we omit Static 2Heads in the following figures.

Remember the fundamental trade-off between the space utilization and approximation accuracy. For all the methods, the more space, the better the approximation. However, the scope between space and accuracy varies across different methods. Figure 6.7(b) illustrates the accuracy as a function of space ratio for both datasets.

2Heads achieves very good compression ratio and it also reveals spatial and temporal patterns as shown in previous section.

Tucker captures spatial correlation but does not give a good compression since the redundancy is mainly in time aspect. Tucker method does not provide a smooth increasing curve as space ratio increases. First, the curve is not smooth because Tucker can only add or drop one component/column including multiple coefficients at a time unlike 2Heads and wavelets which allow to drop one coefficient at a time. Second, the curve is not strictly increasing because there are multiple aspects, different configurations with similar space requirement can lead to very different accuracy.

Wavelets give a good compression but do not reveal any spatial correlation. Furthermore, the summarization is done on each stream, which cannot easily reveal global patterns such as the ones shown in Figure 6.6.

Summary Dynamic 2Heads is efficient in both space utilization and CPU time compared to several all other methods including Tucker, wavelets and Static 2Heads . Dynamic 2Heads is a powerful mining tool combining only strong points from well studied methods while at the same time being computational efficient and applicable to real world situations where data arrive constantly. Therefore, it is as fast as wavelets, it reveals both spatial and temporal patterns as well as their correlations and subsequently achieves good compression.

6.7 Conclusions

We focus on mining of time-evolving streams, when they are associated with multiple *aspects*, like sensor-type (temperature, humidity), and sensor-location (indoor, on-the-window, outdoor). The main difference from old and recent tensor-proposed analysis is that the time aspect needs special treatment, which traditional "one size fit all" type of tensor analysis ignores. Our proposed approach, 2Heads, addresses exactly this problem, by applying the most suitable models to each aspect: wavelet-like for time, and PCA/tensor-like for the categorical-valued aspects.

2Heads has the following key properties:

- Mining patterns: By combining the advantages of existing methods is able to reveal interesting spatio-temporal patterns.
- *Multi-model summarization:* It engages multiple summarization schemes on multi-aspects streams, which gives us a more powerful view to study the high-order data, that traditional models cannot achieve.
- *Error Guarantees:* We proved that it can accurately (and quickly) measure the approximation error, using the orthogonality property of the models.

- *Streaming capability:* 2Heads is fast, incremental and scalable for the streaming environment.
- *Space efficiency:* It provides an accurate approximation which achieves very high compression ratios (over 20:1 ratio), on the real-world datasets we used in our experiments.

Finally, we illustrated the mining power of 2Heads through two case studies on real world datasets. We also performed its scalability through extensive quantitative experiments. Future work includes exploiting alternative methods for categorical aspects, such as Nonnegative Matrix Factorization.

6.8 Appendix

Theorem 13 (Error estimation)

Proof 14 *Let us denote $\mathcal{G}$ and $\hat{\mathcal{G}}$ as the core tensor before and after zero-outing the small entries ($\mathcal{G} = \mathcal{D} \prod \times_i \mathbf{U}^{(i)}$).*

$$
\begin{aligned}
e^2 &= \left\| \mathcal{D} - \hat{\mathcal{G}} \prod_{i=0}^{M} \times_i \mathbf{U}^{(i)\mathsf{T}} \right\|_F^2 / \parallel \mathcal{D} \parallel_F^2 && \textit{def. of } \hat{\mathcal{D}} \\
&= \left\| \mathcal{D} \prod_{i=0}^{M} \times_i \mathbf{U}^{(i)\mathsf{T}} - \hat{\mathcal{G}} \right\|_F^2 / \parallel \mathcal{D} \parallel_F^2 && \textit{unitary trans} \\
&= \left\| \mathcal{G} - \hat{\mathcal{G}} \right\|_F^2 / \parallel \mathcal{D} \parallel_F^2 && \textit{def. of } \mathcal{G} \\
&= \sum_x (g_x - \hat{g}_x)^2 / \parallel \mathcal{D} \parallel_F^2 && \textit{def. of F-norm} \\
&= (\sum_x g_x^2 - \sum_x \hat{g}_x^2) / \parallel \mathcal{D} \parallel_F^2 && \textit{def. of } \hat{\mathcal{G}} \\
&= 1 - \parallel \hat{g} \parallel_F^2 / \parallel \mathcal{D} \parallel_F^2 && \textit{def. of F-norm}
\end{aligned}
$$

Chapter 7

MACH: Fast Randomized Tensor Decompositions

7.1 Introduction

Numerous real-world problems involve multiple aspect data. For example fMRI (functional magnetic resonance imaging) scans, one of the most popular neuroimaging techniques, result in multi-aspect data: voxels $\times$ subjects $\times$ trials $\times$ task conditions $\times$ timeticks. Monitoring systems result in three-way data, machine id $\times$ type of measurement $\times$ timeticks. The machine depending on the setting can be for instance a sensor (sensor networks) or a computer (computer networks). Large data volumes generated by personalized web search, are frequently modeled as three way tensors, i.e., users $\times$ queries $\times$ web pages.

Ignoring the multi-aspect nature of the data by flattening them in a two-way matrix and applying an exploratory analysis algorithm, e.g., singular value decomposition (SVD) ([76]), is not optimal and typically hurts significantly the performance (e.g., [158]). The same holds in the case of applying e.g., SVD on different 2-way slices of the tensor as observed by [94]. On the contrary, multi-way data analysis techniques succeed in capturing the multilinear structures in the data, thus achieving better performance than the aforementioned ideas.

Tensor decompositions have found the last years many applications in different scientific disciplines. Indicatively, computer vision and signal processing (e.g., [158, 111]), neuroscience (e.g., [20]), time series anomaly detection ((e.g., [143]), psychometrics (e.g., [156]), chemometrics (e.g., [137]), graph analysis (e.g., [89, 140]), data mining (e.g., [146]). Two recent surveys of tensor decompositions and their applications are [90],[5], with a wealth of references on the topic.

Two broad families of decompositions are used in the multiway analysis, each with its own characteristics: the canonical decomposition (parallel factor analysis), a.k.a. CANDECOMP (PARAFAC) [28, 72], and the Tucker family of decompositions [156]. In this chapter, we focus on the latter. The Tucker decomposition can be thought of as the generalization of the Singular Value Decompositions (SVD) to the multiway case. Even if there exist algorithms which cast the Tucker decomposition as a nonlinear optimization problem (e.g., [131], [4]), currently in practice the approach followed is the Alternating Least Squares, which involves the computationally expensive SVD. To speed up tensor decompositions, randomized algorithms [53, 106] have appeared in the recent years. This family of randomized algorithms are generalizations of fast low rank approximation methods [48, 105, 52], adapted appropriately to the multiway case.

In this chapter we propose a simple randomized algorithm that speedups significantly the Tucker decomposition while at the same time with guarantees results in an accurate estimate of the tensor decomposition. MACH, the proposed method, can be applied both to "post-mortem" data analysis and to tensor streams to perform data mining tasks such as network anomaly detection, and in general the set of mining tasks which rely on the study of a low rank Tucker approximation. MACH is useful when the data does not fit into the memory and also in tensor streams, such as computer monitoring systems, which was also was also the main motivation behind this work. Specifically, one of the monitoring systems of Carnegie Mellon University, monitors and uses data mining techniques to detect failures. Currently, it monitors over 100 hosts in a prototype data center at CMU. It uses the SNMP protocol and it stores the monitoring data in an mySQL database. Mining anomalies in this system is performed using the SPIRIT method and its extension in the multiway case, i.e., the two heads method which uses a Tucker decomposition and treats the time aspect using wavelets [122, 75, 143]. Applying the aforementioned methods on the large volumes of data is a challenge.

It is worth outlining at this point that in many data mining applications preserving a constant number of principal components almost the same is of high practical value: a low rank approximation typically captures a significant proportion of the variance in many real world processes and outliers can be detected by examining their position relative to the subspace spanned by the PCs.

It is also worth noting that despite many cases where the formulated tensor is sparse, i.e., few non zero elements as observed in [92], there exist real world problems where the tensor is dense. As table 7.1 shows, for both monitoring system we use in the experimental section 7.4, the resulting tensors are very dense.

This is the typical case in a monitoring system, since at timetick k we receive a measurement j for machine i, resulting in a non zero in (i, j, k).

name	**Percentage of non-zeros**
`Sensor Network Data` [47]	85 %
`Computer Network Data ([75])`	81%

Table 7.1: Tensors from monitoring system are typically dense.

The main contributions of this chapter are summarized as in the following:

- MACH, a randomized algorithm to compute the Tucker decomposition of a tensor $\mathcal{X}$.
- The following theorem, which is our main theoretical result.

  Theorem 14 *Let* $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_d}$ *a* d*-mode tensor. Let* $I_n \geq 76$, $I_n^2 \leq \prod_{j=1}^{d} I_j$ *for* $n = 1 \ldots d$,

  $$\alpha = \max_j \frac{\prod_{m=1}^{d} I_m}{I_j}, b = \max_{i_1,\ldots,i_d} |\mathcal{X}_{i_1,\ldots,i_d}|$$

  For $p \geq \frac{(8 log\alpha)^4}{\alpha}$ *let* $\hat{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_d}$ *whose entries are independently distributed as:* $\hat{\mathcal{X}}_{i_1,\ldots,i_d} = \frac{\mathcal{X}_{i_1,\ldots,i_d}}{p}$ *with probability* p*, otherwise 0. Let* $\hat{\mathcal{G}}, A^{(1)}, \ldots, A^{(1)}$ *be the* $(R_1, \ldots, R_d)$ *Higher Order Singular Value Decomposition of* $\hat{\mathcal{X}}$*, where* $A^{(m)}$ *is a* $I_m \times R_m$ *matrix for* $m = 1 \ldots d$ *and* $\hat{\mathcal{G}}$ *the core tensor* $R_1 \times \ldots \times R_d$, $R_m \leq I_m$*. With probability at least* $\prod_{i=1}^{d}(1 - exp(-19 \sum_{k=1,k\neq i}^{d} \log I_k))$ *the following holds:*

  $$||\mathcal{X} - \hat{\mathcal{X}}|| \leq 4b \sum_{m=1}^{d} \sqrt{\frac{R_i}{p} \frac{\prod_{k=1}^{d} I_k}{I_j}} \tag{7.1}$$

- Experiments on monitoring systems, where we demonstrate the success of our proposed algorithm.

The outline of the chapter is the following: in Section 7.2 we briefly present the necessary theoretical background, in Section 7.3 we describe and analyze the proposed method and in Section 7.4 we present the experimental results. We conclude in Section 7.5.

7.2 Background

In this section we present briefly the background behind tensors and low rank approximations. Table 7.2 shows the symbols and the abbreviations we use and their explanation.

Symbol	Definition and Description
d	number of modes
I_j	dimensionality of j-th mode
$\mathcal{X}, \mathcal{Y}, \ldots \in \mathbb{R}^{I_1 \times \ldots \times I_d}$	d-mode tensor (calligraphic)
$A, U, \ldots \in \mathbb{R}^{m \times n}$	matrices (upper case)
$\alpha, \beta, a_{i,j}, x_{i_1,\ldots,i_d}$	scalars (lower case)
$\times_n$	mode-n product
HOOI	Higher Order Orthogonal Iteration [99]
HOSVD	Higher Order Singular Value Decomposition [42]

Table 7.2: Symbols

7.2.1 Tensors

Historical Remarks Tensors traditionally have been used in physics (e.g., stress and strain tensors). After Einstein presented the theory of general relativity tensor analysis became popular. Tucker introduced tensor analysis in psychometrics [156] (Tucker family). Harshman [72] and Carrol and Chang [28] independently proposed the canonical decomposition of a tensor (CANDECOMP family). These two families of decompositions come with different names, see [90]. The difference between them is visualized for a three way tensor in figure 7.2.1. In the following we will focus on Tucker decompositions.

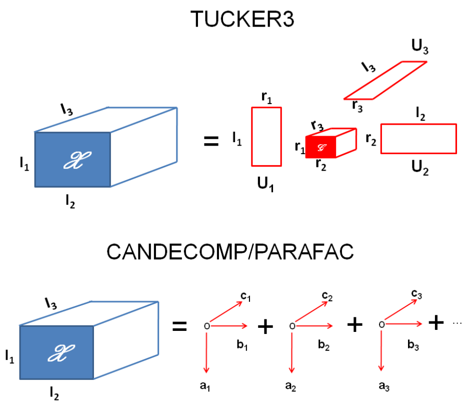


Figure 7.1: CANDECOMP/PARAFAC and Tucker tensor decompositions.

Tensor Concepts Let $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_d}$ be a multiway array. We will call $\mathcal{X}$ a tensor, i.e., we will use the terms multiway array and tensor interchangeably. The order of a tensor is the number of dimensions, also known as ways, modes or aspects and is equal to d for tensor $\mathcal{X}$. The dimensionality of the j-th mode is equal to I_j.

The norm of tensor $\mathcal{X}$ is defined to be the square root of the sum of all entries of the tensor squared, i.e.,

$$||\mathcal{X}|| = \sqrt{\sum_{j_1=1}^{I_1} \sum_{j_2=1}^{I_2} \ldots \sum_{j_d=1}^{I_d} x^2_{j_1,\ldots,j_d}} \tag{7.2}$$

As we see the norm of a tensor is the straight-forward generalization of the Frobenius norm of a matrix (2 modes) to N modes.

The inner product of two tensors with the same number of modes and equal dimensionality per mode, $\mathcal{X}, \mathcal{Y} \in \mathbb{R}^{I_1 \times I_2 \times \ldots \times I_d}$, is defined by the following equation:

$$\langle X, Y \rangle = \sum_{j_1=1}^{I_1} \sum_{j_2=1}^{I_2} \ldots \sum_{j_d=1}^{I_d} x_{j_1,\ldots,j_d} y_{j_1,\ldots,j_d} \tag{7.3}$$

Observe that equation 7.2 can equivalently be written as $||\mathcal{X}|| = \sqrt{\langle X, X \rangle}$ A tensor fiber (slice) is a one (two)-dimensional fragment of a tensor, obtained by fixing all indices but one (two). For more details on tensor fibers and slices, see [90].

Matricization along mode k, results in a $I_k \times \prod_{j=1, j\neq k}^{d} I_j$ matrix. The $(i_1, \ldots, i_d)$ element is mapped to (i_k, j) where $j = 1 + \sum_{q=1, q\neq k}^{d} (i_q - 1) J_q$ where $J_q = \prod_{m=1, m\neq k}^{q-1} I_m$. Figure 7.2.1 shows the concept of matricization for a three-way tensor. The operation of matricization naturally introduces the concept of a vector containing ranks $(R_1, \ldots, R_d)$: R_i is equal to the rank of the $X_{(i)}$, the matrix resulting by the matricization of the tensor $\mathcal{X}$ along the i-th mode.

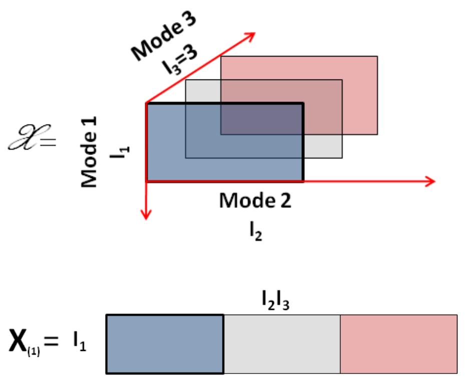


Figure 7.2: Matricization of a three-way $I_1 \times I_2 \times I_3$, $I_3 = 3$, tensor along the first mode. The three slices are denoted with different color.

The n-mode product of $\mathcal{X}$ with a matrix $M \in \mathbb{R}^{J\times I_n}$ is denoted by $\mathcal{X} \times_n M$ and is a tensor of size $I_1 \times I_2 \times \ldots I_{n-1} \times J \times I_{n+1} \times \ldots I_d$. Specifically,

$$(\mathcal{X} \times_n M)_{i_1 \ldots i_{n-1} j i_{n+1} \ldots i_d} = \sum_{i_n=1}^{I_n} x_{i_1 \ldots i_{n-1} j i_{n+1} \ldots i_d} m_{j i_n} \tag{7.4}$$

Some important facts concerning n-mode products, is the following:

$$\mathcal{X} \times_m A \times_n B = \mathcal{X} \times_n B \times_m A, m \neq n \tag{7.5}$$

The importance of this equation lies in the fact that the order of execution of the tensor matrix products does not play any role, as long as the multiplications are along different modes. When we multiply a tensor and two matrices along the same mode the following equation holds:

$$\mathcal{X} \times_m A \times_m B = \mathcal{X} \times_m (BA) \tag{7.6}$$

Furthermore, if $UU^T = I$ then $||A \times_n U|| = ||A||$.

The rank R of the d-way tensor $\mathcal{X}$ is the minimum number of d-linear components to fit $\mathcal{X}$ exactly, i.e.,:

$$\mathcal{X} = \sum_{m=1}^{R} c_m^{(1)} \circ c_m^{(2)} \circ \ldots \circ c_m^{(d)} \tag{7.7}$$

where $c_1^{(j)}, \ldots, c_R^{(j)}$ are the R components for the j-th mode and $\circ$ denotes the tensor product. Even if the above generalization is a straightforward generalization of the rank of a matrix, the concept of the tensor rank is special. For example, for a matrix $A \in \mathbb{R}^{2\times 2}$ the column rank R_c and the row rank R_r are equal $R_c = R_r = r$ to the matrix rank r. Furthermore, $r \leq 2$. However for a tensor $\mathcal{X} \in \mathbb{R}^{2\times 2\times 2}$ the rank can be 2 or 3 [95]. Therefore the word rank can have different meanings: a) The individual rank, i.e., for a specific instance of a tensor what is R? b) The typical rank is the rank that we almost surely observe. For example for $2 \times 2 \times 2$ tensors the typical rank is $\{2, 3\}$. c) Vector of dimensions $(R_1, \ldots, R_d)$. The value of R_i is equal to the rank of the matricized version $X_{(i)}$ of the tensor.

Consider figure 7.2.1, which depicts a three mode tensor $\mathcal{X} \in \mathbb{R}^{I_1\times I_2\times I_3}$. The PARAFAC/CANDECOMP model is given by equation 7.8, whereas the Tucker model is given by equation 7.9.

$$\mathcal{X}_{ijk} = \sum_{r=1}^{R} \alpha_{ir} b_{jr} c_{qr} \lambda_r + e_{ijk} \tag{7.8}$$

$$\mathcal{X}_{ijk} = \sum_{p=1}^{P} \sum_{q=1}^{Q} \sum_{r=1}^{R} \alpha_{ip} b_{jq} c_{kr} g_{pqr} + e_{ijk} \tag{7.9}$$

Few brief remarks on the above two models: a) In terms of the fit, the Tucker family is at least as good as the PARAFAC/CANDECOMP since as we see from the above equations, the PARAFAC model can be viewed as a restrictive Tucker model, where the core tensor $\mathcal{G}$ is superdiagonal, i.e., $g_{pqr} \neq 0$ only if $p = q = r$. However, it is worth noting that better fit is not necessarily optimal (see [137], Ch.7) b) The Tucker model does not result in unique solutions since it has rotational freedom. Typically one chooses a solution that satisfies a certain criterion, as the all-orthogonality core tensor: $\langle G(m, :, :), G(n, :, :)\rangle = \langle G(:, m, :), G(:, n, :)\rangle = \langle G(:, :, m), G(:, :, n)\rangle = 0$ when $m \neq n$ ([42]). c) Basic concepts as the uniqueness of the canonical tensor decomposition, degeneracy of the rank, border rank were not discussed here. A good reference is the [90] and the related references therein.

In the following we focus on the Tucker family. Compressing n out of the d modes of a tensor results in a Tucker-n decomposition ([85]). For example, for a three mode tensor we can have the Tucker1, Tucker2 and Tucker3 decomposition. In the following we discuss algorithms for the Tucker3 decomposition and briefly state some facts about Tucker2 and Tucker1 decompositions. Generalization to d modes is straightforward.

Tucker3 Algorithms The algorithm which should be used to compute the Tucker3 decomposition of a tensor depends on whether or not the data is noise free. In the former case, an exact, closed form solution exists, whereas in the latter case the alternating least squares algorithm (ALS) is frequently used. However, it is worth noting that even in cases where there is noise in the data, the closed form solution a.k.a. as HOSVD [90, 42] is satisfactory in practice [103].

Let $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ and (R_1, R_2, R_3) the vector containing the desired approximation ranks along each mode. In the case of noise-free data, the algorithm matricizes the tensor along each mode and computes the R_k top left singular vectors $k = 1, 2, 3$. Let A_k be the $I_k \times R_k$ matrix containing in its columns those vectors. The core tensor is computed with the following equation:

$$\mathcal{G} = \mathcal{X} \times_1 A_1^T \times_2 A_2^T \times_3 A_3^T \tag{7.10}$$

In the case of noise in the data, one performs the alternating least squares algorithm. To solve the nonlinear optimization problem that tries to optimize the fit of the low rank approximation with respect to the original tensor, one converts the problem into a linear one, by "fixing" all modes but one and optimizing along that mode. This method is also known as Higher Order Orthogonal Iteration (HOOI). This procedure is continued until some stopping criterion is met, i.e., ϵ improvement in terms of fit.

7.2.2 SVD and Fast Low Rank Approximation

Any matrix $A \in \mathbb{R}^{m \times n}$ can be written as a sum of rank 1 matrices, i.e., $A = \sum_{i=1}^{r} \sigma_i u_i v_i^T$, where $u_i, i = 1 \ldots r$ (left singular vectors) and $v_i, i = 1 \ldots r$ (right singular vectors) are orthonormal and the singular values are ordered in decreasing order $\sigma_1 \geq \ldots \geq \sigma_r > 0$. Here r is the rank of A. We denote with A_k the k-rank approximation of A, i.e., $A_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$. Among all matrices $C \in \mathbb{R}^{m \times n}$

of rank at most k, A_k is the one that minimizes $||A - C||_F$ ([76]). Since the computational cost of the SVD is high, $O(\min(m^2n, n^2m))$ for the full SVD approximation algorithms that give a close to the optimal solution A_k have been developed. Frieze, Kannan and Vempala showed in a breakthrough paper [63] that an approximate SVD can be computed by a randomly chosen submatrix of A. It is remarkable that the complexity does not depend at all on m, n. Their Monte-Carlo algorithm with probability at least $1 - \delta$ outputs a matrix $\hat{A}$ of rank at most k that satisfies the following equation:

$$||A - \hat{A}||_F^2 \leq ||A - A_k||_F^2 + \epsilon ||A||_F^2 \tag{7.11}$$

Drineas et al. in [51] showed how to find such a low rank approximation in $O(mk^2)$ time. A lot of work has followed on this problem. Here, we present the results of Achlioptas-McSherry [7] which are used in our work[1]. The main theorem that is of interest to us is theorem 15.

Theorem 15 (Achlioptas-McSherry [7]) *Let A be any $m \times n$ matrix where $76 \leq m \leq n$ and let $b = \max_{ij} |A_{ij}|$. For $p \geq (8 \log n)^4/n$. Let $\hat{A}$ be a random $m \times n$ matrix whose entries are independently distributed, with $\hat{A}_{ij} = A_{ij}/p$ with probability p and 0 with probability $1 - p$. Then with probability at least 1-exp$(19(\log n)^4)$, the matrix $N = A - \hat{A}$ satisfies the following two equations:*

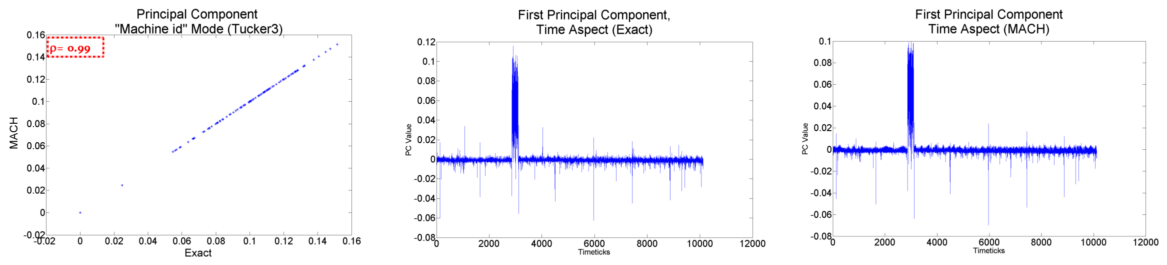
$$||N_k||_2 < 4b\sqrt{\frac{n}{p}} \tag{7.12}$$

$$||N_k||_F < 4b\sqrt{\frac{nk}{p}} \tag{7.13}$$

Randomized Tensor Algorithms As already discussed, the most computationally expensive step for the Tucker decomposition is the SVD part. To alleviate this cost, two randomized algorithms which select columns according to a biased probability distribution for tensor decompositions [53] have been proposed, extending the results of [48]and [52] to the multiway case and TensorCUR [106], the extension of the CUR method [105] in n-modes. Roughly speaking, the bounds proved are of the form 7.11.

[1]We call our proposed method MACH, to acknowledge the fact that it is based on the **Ach**lioptas-**M**cSherry work.

7.3 Proposed Method



(a) (b) (c)

Figure 7.3: (a) Top approximate Principal Component (PC) of the "machine-id" mode using the sampling MACH method vs. the exact PC. The PC was computed using a Tucker3 decomposition of the three-way tensor machine id *x* type of measurement *x* timeticks, formulated by data from the CMU monitoring system [75]. MACH used approximately 10% of the original data. Pearson's correlation coefficient is shown in the inset, and is almost equal to the ideal value 1. Such PCs are of high practical value since they are used in outlier detection algorithms [75, 122, 143]. (b) Exact PC for the time aspect (c) Approximate PC using MACH. Pearson's correlation coefficient for the two time series equals 0.9772, again close to the ideal value 1.

The proof of theorem 14 follows:

Proof 15 *Let $\mathcal{E} = \mathcal{X} - \hat{\mathcal{X}}$. By the definition of HOSVD and the properties of the n-mode product we have: $\hat{\mathcal{X}} = \mathcal{X} \times_1 A^{(1)}A^{(1)T} \dots \times_d A^{(d)}A^{(d)T}$. By using the triangular inequality for a norm and the fact that $P_i = A^{(i)}A^{(i)T}$ for $i = 1, \dots, d$ is a projection matrix $P_i = P_i^2$, we have the following:*

$||\mathcal{E}|| = ||\mathcal{X} - \mathcal{X} \times_1 A^{(1)}A^{(1)T} \dots \times_d A^{(d)}A^{(d)T}|| = ||\mathcal{X} - \mathcal{X} \times_d A^{(d)}A^{(d)T} + \mathcal{X} \times_d A^{(d)}A^{(d)T} - \mathcal{X} \times_1 A^{(1)}A^{(1)T} \dots \times_d A^{(d)}A^{(d)T}|| \leq ||\mathcal{X} - \mathcal{X} \times_d A^{(d)}A^{(d)T}|| + ||(\mathcal{X} - \mathcal{X} \times_1 A^{(1)}A^{(1)T} \dots \times_{d-1} A^{(d-1)}A^{(d-1)T}) \times_d A^{(d)}A^{(d)T}|| \leq ||\mathcal{X} - \mathcal{X} \times_d A^{(d)}A^{(d)T}|| + ||\mathcal{X} - \mathcal{X} \times_1 A^{(1)}A^{(1)T} \dots \times_{d-1} A^{(d-1)}A^{(d-1)T}||$

Applying for $i = 1 \dots (d-1)$ the same procedure the right hand term of the right hand side of the inequality, we get that the norm of the residuals tensor $\mathcal{E}$ satisfied the following equation:

$||\mathcal{E}|| \leq \sum_{i=1}^{d} ||\mathcal{X} - \mathcal{X} \times_d A^{(d)}A^{(d)T}|| = \sum_{i=1}^{d} ||X_{(i)} - \hat{X_{(i)}}||$

This follows from the fact that the norm of a tensor does not change by matricization and from the orthonormality properties of the left singular vectors. Now,

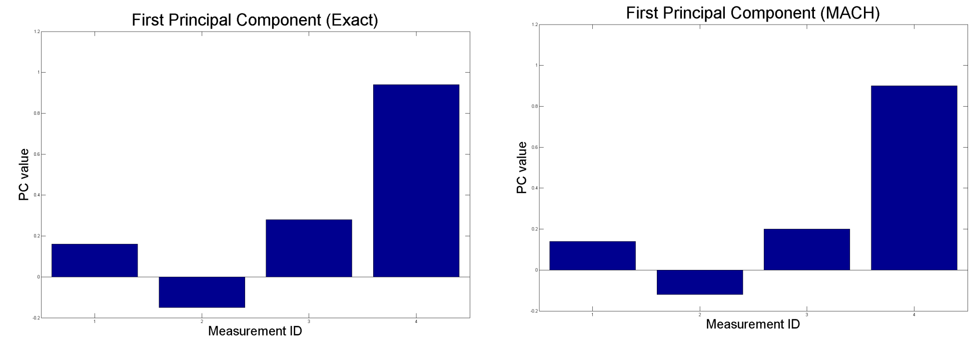


Figure 7.4: Principal component for the "type of measurement" aspect for the Intel Lab Berkeley sensor network [47]. Ids 1 to 4 correspond to voltage, humidity, temperature and light intensity. As we observe, the PC captures the correlations between those types and MACH succeeds with p=0.1 in preserving them accurately.

using theorem 15 we further bound each of the d terms in the summation. Specifically, given our conditions $I_n \geq 76$, $I_n^2 \leq \prod_{j=1}^{d} I_j$ and $\alpha = \max_j \frac{\prod_{m=1}^{d} I_m}{I_j}$, we assure that the assumptions of theorem 15. Furthermore, observe that $b = \max_{i_1,\ldots,i_d} |\mathcal{X}_{i_1,\ldots,i_d}|$ is the maximum for every matricized version of our tensor. Thus we obtain the following:

$$||\mathcal{E}|| \leq 4b \sum_{m=1}^{d} \sqrt{\frac{R_i}{p} \frac{\prod_{k=1}^{d} I_k}{I_m}}$$

Remarks **(1)** Theorem 14 suggests algorithm 1, MACH-HOSVD. The algorithm takes as input a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \ldots \times I_d}$ and a vector containing the desired ranks of approximation along each mode $(R_1, \ldots, R_d)$. MACH tosses a coin for each non-zero entry $\mathcal{X}_{i_1,\ldots,i_d}$ of the tensor with probability p of keeping it and $1-p$ for zeroing it. In case of keeping it, we reweigh it, i.e., $\mathcal{X}_{i_1,\ldots,i_d} \leftarrow \frac{\mathcal{X}_{i_1,\ldots,i_d}}{p}$. **(2)** Frequently small R_i's result in a satisfactory approximation of the original tensor. The sparsification process we propose due to its simplicity is easily parallelizable and can easily be adapted to the streaming case [75] by tossing a coin each time a new measurement arrives. **(3)** Picking the optimal p in a real world application can be hard, especially in the context we are interested in, i.e., monitoring systems, where data is constantly arriving. Another potential problem are the assumptions of the theorem which may be violated. Fortunately, this does not render MACH algorithm useless. On the contrary, picking a constant p even for small tensors which do not satisfy the conditions of the theorem result turns out to be accurate enough to perform data analysis. Furthermore, constant factor speedups and space savings are significant in many real-world applications. **(4)** The expected speedup depends on the under-the-hood method to find the top k singular vectors of a matrix. Lanczos method [69] is such a method. Recently, approximation algorithms approximate the k-rank approximation of a matrix in linear time [129]. Thus, if such a fast algorithm is used, the expected speedup is $\frac{1}{p}$. **(5)** Theorem 14 refers to the HOSVD of a tensor. We can apply the same idea to the HOOI. This results in algorithm 2. We do not analyze the performance of algorithm 2 here, since it would require the analysis of the convergence of the alternating least squares method which does not exist yet. As we will show in the experimental section 7.4, MACH-HOOI gives satisfactory results.

Algorithm 8 MACH-HOSVD

```
Require: $\mathcal{X} \in \mathbb{R}^{I_1 \times \ldots \times I_d}$
Require: $(R_1, \ldots, R_d)$
Require: $p$
  {MACH}
  for each $\mathcal{X}_{i_1,\ldots,i_d}$, $i_j = 1 \ldots I_j$ toss a coin with probability $p$ of keeping it.
  if success then
    $\mathcal{X}_{i_1,\ldots,i_d} \leftarrow \frac{\mathcal{X}_{i_1,\ldots,i_d}}{p}$
  else
    $\mathcal{X}_{i_1,\ldots,i_d} \leftarrow 0$
  end if{HOSVD}
  for $i = 1$ to $d$ do
    $A^{(i)} \leftarrow R_i$ leading left singular vectors of $X_{(i)}$
  end for
  $\mathcal{G} \leftarrow \mathcal{X} \times_1 A^{(1)T} \times_2 A^{(2)T} \ldots \times_d A^{(d)T}$
  return $\mathcal{G}, A^{(1)}, \ldots, A^{(d)}$
```

7.4 Experiments

Experimental Setup We used the Tensor Toolbox [15], which contains MATLAB implementations of the HOSVD and the HOOI. Our experiments ran in a 2GB RAM, Intel(R) Core(TM)2 Duo CPU at 2.4GHz Ubuntu Linux machine.

Algorithm 9 MACH-HOOI

```
Require: $\mathcal{X} \in \mathbb{R}^{I_1 \times \ldots \times I_d}$
Require: $(R_1, \ldots, R_d)$
Require: $p$
  {MACH}
  for each $\mathcal{X}_{i_1,\ldots,i_d}$, $i_j = 1 \ldots I_j$ toss a coin with probability $p$ of keeping it.
  if success then
    $\mathcal{X}_{i_1,\ldots,i_d} \leftarrow \frac{\mathcal{X}_{i_1,\ldots,i_d}}{p}$
  else
    $\mathcal{X}_{i_1,\ldots,i_d} \leftarrow 0$
  end if
  {HOOI}
  initialize $A^{(k)} \in \mathbb{R}^{I_k \times R_k}$ for $k = 1 \ldots d$ using HOSVD
  repeat
    for $i = 1$ to $d$ do
      $\mathcal{Y} \leftarrow \mathcal{X} \times_1 A^{(1)T} \ldots \times_{i-1} A^{(i-1)T} \times_{i+1} A^{(i+1)T} \ldots \times_d A^{(d)T}$
      $A^{(i)} \leftarrow R_i$ leading left singular vectors of $Y_{(i)}$
    end for
  until fit stops improving or maximum number of iterations is reached
  $\mathcal{G} \leftarrow \mathcal{X} \times_1 A^{(1)T} \times_2 A^{(2)T} \ldots \times_d A^{(d)T}$
  return $\mathcal{G}, A^{(1)}, \ldots, A^{(d)}$
```

Table 7.3 describes the datasets we use. The motivation of our method as already mentioned, is to provide a practical algorithm for tensor decompositions which involve streams, such as monitoring systems. It is also worth noting that the assumptions of theorem 14 do not hold. Nonetheless, results are close to ideal. Finally, in this section we report experimental results for the MACH-HOOI. The reason is that Tucker decompositions using alternating least squares are used in practice more than the HOSVD and also, they have already been successfully applied to the real world problems we consider in the following [143]. The results for HOSVD are consistently same or better than the results we report in this section.

7.4.1 Monitoring computer networks

As already mentioned in Section 7.1, a prototype monitoring system in Carnegie Mellon University uses data mining techniques successfully [122, 75, 143] to spot

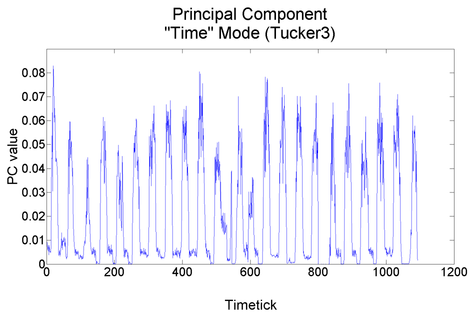
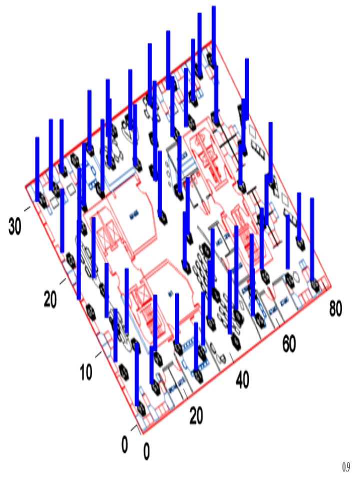


Figure 7.5: Principal component for the time aspect using MACH with p=0.1. Daily periodicity appears to be the dominant latent factor for the time aspect.

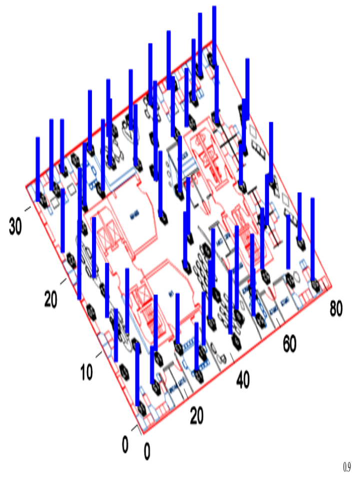
name	$I_1 \times I_2 \times I_3$
`Sensor Network Data ([47])`	54-by-4-by 5385
`Intemon Data ([75])`	100-by-12-by-10080

Table 7.3: Dataset summary. The third aspect is the time aspect.

anomalies and detect correlations among different types of measurements and machines. Analyzing and applying these techniques on large amounts of data however is a challenge. A natural way to model this type of data is a three-way tensor, i.e., machine id$\times$type of measurement$\times$time. The data on which we apply MACH is a tensor $\mathcal{X} \in \mathbb{R}^{100\times 12\times 10080}$. The first aspect is the "machine id" aspect and the second is the "type of measurement" aspect (bytes received, unicast packets received, bytes sent, unicast packets sent, unprivileged CPU utilization, other CPU utilization, privileged CPU utilization, CPU idle time, available memory, number of users, number of processes and disk usage). The third aspect is the time aspect. Figure 7.3(a) plots the Principal Component (PC) of the "machine id" aspect after performing a Tucker3 decomposition using MACH versus the exact PC. Our sampling approach thus kept approximately the 10% of the original data. As the figure shows, the results are close to ideal and similar results hold for the other few top PCs. Specifically, Pearson's correlation coefficient is 0.99, close to the ideal 1 which is the perfect linear correlation between the exact and the approximate top PC. This fact is important since these PCs can be used to find outlier machines, which ideally would be the machines that face a functionality problem.



(a) SENSOR Concept 1



(b) SENSOR Concept 1 using MACH

Figure 7.6: (a) shows the distribution of the most dominant trend, (b) shows the distribution of the most dominant trend, using MACH algorithm with p=0.1. Pearson's correlation coefficient equals 0.93, and thus the qualitative analysis of the dominant sensor/spatial correlations remains unaffected by the sparsification. Colored bars indicate positive weights of the corresponding sensors. As suggested in [143], e values assigned to the sensors are more or less uniform suggesting that the dominant trend is strongly affected by the daily periodicity.

Figures 7.3(b), 7.3(c) show the exact top and the MACH PC for the time aspect. Pearson's correlation coefficient is equal to 0.98. We observe that there is no clear periodic pattern in this time series. The important fact is that MACH using only 10% of the data, results in a good approximation. This is of significant practical value and can be used also in conjunction with DTA [142] to perform dynamic tensor analysis in larger time windows.

7.4.2 Environmental Monitoring

In this application we use data from the Intel Berkeley Research Lab sensor network [47]. The data is collected from 54 Mica2Dot sensors which measure at every timetick humidity, temperature, light and voltage.

It has been shown in [143] that tensor decompositions along with a wavelet analysis can efficiently capture anomalies in the network, i.e., battery outage as well as spatial and measurement correlations. In this section we show that a random subset about 10% of the initial data volume suffices to perform the same analysis as if we had used the whole tensor.

Figure 7.4 shows the correlations revealed by the the principal component for the “type of measurement” aspect. As we observe, voltage, temperature and light intensity are positively correlated, whereas at the same time the latter types of measurement are negatively correlated with humidity. This is because during the day, temperature and light intensity go up but humidity drops because the air conditioning system is on. Similarly , during the night, temperature and light intensity go down but humidity increases because the air conditioning system is off. Furthermore, the positive correlation between voltage and temperature is due to the design of MICA2 sensors. As we observe again, MACH gives the same qualitative analysis by examining the principal component. Pearson’s correlation coefficient is close to the ideal value 1. Figure 5 shows the principal component for the time aspect. A periodic pattern is apparent and corresponds to the daily periodicity. Performing a Tucker2 decomposition as suggested by [143] and plotting the fiber of the core tensor corresponding to the principal components of the tensor for the “sensor id” and “measurement type” mode, the results are again close to ideal. Figure 7.6(a) shows the principal component for the “sensor id” aspect using the exact Tucker decomposition and Figure 7.6(b) using MACH with p=0.1. The top component captures spatial correlations and MACH preserves them with a random subset of size approximately 10% of the original data. Pearson’s correlation coefficient is equal to 0.93.

7.4.3 Discussion

The above experiments show MACH results in a good approximation of the desired low rank Tucker approximation of a tensor. Similar result hold for the other few top principal components and for the HOSVD. As already mentioned, the above applications were selected since it has already been shown by previous work that Tucker decompositions and SVD can detect anomalies and correlations. Thus, the main goal of this section is -rather than introducing new applications- to show that keeping a small random subset of the tensor can give good results. Speedups due to the small size of the two datasets and the implementation was less than the expected 10$\times$ (running fact C code for the sparsification and then applying the tensor toolbox resulted in 2-3$\times$ faster performance). However, as the size of the tensor grows bigger the speedup should become apparent. Choosing the best possible p is an issue. We use a constant p, i.e., p=10% in our experiments[2]. Constant p's are of significant practical value in such settings where it is not clear how one should set p to sparsify the underlying tensor optimally. For "post-mortem" data analysis, one can try setting lower values for p according to theorem 14.

7.5 Conclusions

Tensor decompositions are useful in many real world problems. Here we focused on the Tucker decompositions. We proposed a simple randomized algorithm MACH which is easily parallelizable and adapted to online streaming systems since it simply tosses a coin for each entry of the tensor. Specifically, our contributions include:

- A new algorithm MACH, which keeps a small percentage of the entries of a tensor, and still produces an accurate low rank approximation of the tensor. We performed a theoretical analysis of the algorithm in Theorem 14 and of its speedup in Section 7.3.
- An experimental evaluation of MACH on two real world datasets, both generated from a monitoring system, where we showed that for constant values of p excellent performance.

This algorithm will be incorporated in the PEGASUS library, a graph and tensor mining system for handling large amounts of data using Hadoop, the open source version of MapReduce [44].

[2]For both applications that value of p, gives excellent results. If we set p=5% for the first application results get significantly worse whereas for the second results remain good.

Bibliography

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

[2] Hama website. http://incubator.apache.org/hama/.

[3] E. Acar, S. A. Çamtepe, M. S. Krishnamoorthy, and B. Yener. Modeling and multiway analysis of chatroom tensors. In *ISI*, pages 256–268, 2005.

[4] E. Acar, T. G. Kolda, and D. M. Dunlavy. An optimization approach for fitting canonical tensor decompositions. Technical Report SAND2009-0857, Sandia National Laboratories, 2009.

[5] E. Acar and B. Yener. Unsupervised multiway data analysis: A literature survey. *TKDE*, 21(1):6–20, 2009.

[6] D. Achlioptas and F. McSherry. Fast computation of low rank matrix approximation. In *STOC*, 2001.

[7] D. Achlioptas, F. McSherry, and F. M. Fast computation of low rank matrix approximations, 2001.

[8] C. C. Aggarwal, Y. Xie, and P. S. Yu. Gconnect: A connectivity index for massive disk-resident graphs. *PVLDB*, 2009.

[9] G. Aggarwal, M. Data, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. *Proceedings of FOCS*, 2004.

[10] N. Alon and S. Joel. *The Probabilistic Method.* Wiley Interscience, New York, second edition, 2000.

[11] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *STOC '96: Proceedings of the twenty-eighth*

annual ACM symposium on Theory of computing, pages 20–29, New York, NY, USA, 1996. ACM.

[12] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments, 1996.

[13] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

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

[15] B. W. Bader and T. G. Kolda. Efficient MATLAB computations with sparse and factored tensors. Technical Report SAND2006-7592, Sandia National Laboratories, Albuquerque, NM and Livermore, CA, 2006.

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

[17] Z. Bar-Yosseff, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 623–632, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics.

[18] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proceedings of ACM KDD*, Las Vegas, NV, USA, August 2008.

[19] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *KDD*, 2008.

[20] C. F. Beckmann and S. M. Smith. Tensorial extensions of independent component analysis for multisubject fmri analysis. *Neuroimage*, 2005.

[21] A. A. Benczúr and D. R. Karger. Randomized approximation schemes for cuts and flows in capacitated graphs. *CoRR*, cs.DS/0207078, 2002.

[22] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. SIGMOD, 2007.

[23] B. Bollobas. *Random Graphs*. Cambridge University Press, 2001.

[24] 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.

[25] 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.

[26] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting triangles in data streams. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 253–262, New York, NY, USA, 2006. ACM.

[27] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting triangles in data streams. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 253–262, New York, NY, USA, 2006. ACM.

[28] J. Carroll and J.-J. Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of eckart-young decomposition. *Psychometrika*, 1970.

[29] 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.

[30] D. Chakrabarti, S. Papadimitriou, D. S. Modha, and C. Faloutsos. Fully automatic cross-associations. In *KDD*, 2004.

[31] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. PODS, 2000.

[32] C. Chen, X. Yan, F. Zhu, and J. Han. gapprox: Mining frequent approximate patterns from a massive network. *ICDM*, 2007.

[33] J. Chen, O. R. Zaiane, and R. Goebel. Detecting communities in social networks using max-min modularity. *SDM*, 2009.

[34] J. Cheng, J. X. Yu, B. Ding, P. S. Yu, and H. Wang. Fast graph pattern matching. *ICDE*, 2008.

[35] P. A. Chew, B. W. Bader, T. G. Kolda, and A. Abdelali. Cross-language information retrieval using parafac2. In *KDD*, pages 143–152, New York, NY, USA, 2007. ACM Press.

[36] F. Chung, L. Lu, and V. Vu. Eigenvalues of random power law graphs. *Annals of Combinatorics*, 7(1):21–33, June 2003.

[37] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51:661, 2009.

[38] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 1–6, New York, NY, USA, 1987. ACM.

[39] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[40] S. Daruru, N. M. Marin, M. Walker, and J. Ghosh. Pervasive parallelism in data mining: dataflow solution to co-clustering large and sparse netflix data. In *KDD*, 2009.

[41] I. Daubechies. *Ten Lectures on Wavelets*. Capital City Press, Montpelier, Vermont, 1992. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA.

[42] L. De Lathauwer, B. De Moor, and J. Vandewalle. A multilinear singular value decomposition. *SIAM J. Matrix Anal. Appl*, 21:1253–1278, 2000.

[43] L. De Lathauwer, B. D. Moor, and J. Vandewalle. A multilinear singular value decomposition. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1253–1278, 2000.

[44] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters.

[45] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI '04*, pages 137–150, December 2004.

[46] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI*, 2004.

[47] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *VLDB '04*, pages 588–599, 2004.

[48] A. Deshpande, L. Rademacher, S. Vempala, and G. Wang. Matrix approximation and projective clustering via volume sampling. In *SODA*, 2006.

[49] I. S. Dhillon, Y. Guan, and B. Kulis. Weighted graph cuts without eigenvectors a multilevel approach. *IEEE TPAMT*, 2007.

[50] I. S. Dhillon, S. Mallela, and D. S. Modha. Information-theoretic co-clustering. In *The Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 03)*, Washington, DC, August 24-27 2003.

[51] P. Drineas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay. Clustering in large graphs and matrices. In *SODA '99*, pages 291–299, 1999.

[52] P. Drineas, R. Kannan, and M. W. Mahoney. Fast monte carlo algorithms for matrices ii: Computing a low-rank approximation to a matrix. *SIAM J. on Computing*, 2004.

[53] P. Drineas and M. W. Mahoney. A randomized algorithm for a tensor-based generalization of the singular value decomposition. *In Linear Algebra and its Applications*, 2005.

[54] R. Dunbar. Grooming, gossip, and the evolution of language. *Harvard Univ Press*, October 1998.

[55] J.-P. Eckmann and E. Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *PNAS*, 99(9):5825–5829, April 2002.

[56] P. Erdős and A. Rényi. On random graphs. *Publicationes Mathematicae*, 1959.

[57] T. Falkowski, A. Barth, and M. Spiliopoulou. Dengraph: A density-based community detection algorithm. *Web Intelligence*, 2007.

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

[59] I. J. Farkas, I. Derenyi, A.-L. Barabasi, and T. Vicsek. Spectra of ”real-world” graphs: Beyond the semi-circle law. *Physical Review E*, 64:1, 2001.

[60] J.-A. Ferrez, K. Fukuda, and T. Liebling. Parallel computation of the diameter of a graph. In *HPCSA*, 1998.

[61] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *Journal of Computer and System Sciences*, 1985.

[62] A. Frieze, R. Kannan, and S. Vempala. Fast monte-carlo algorithms for finding low-rank approximations. In *In Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science*, pages 370–378, 1998.

[63] A. Frieze, R. Kannan, and S. Vempala. Fast monte-carlo algorithms for finding low-rank approximations. *J. ACM*, 51(6):1025–1041, 2004.

[64] Z. Füredi and J. Komlós. The eigenvalues of random symmetric matrices. *Combinatorica*, 1(3):233–241, 1981.

[65] M. N. Garofalakis and P. B. Gibbon. Approximate query processing: Taming the terabytes. *VLDB*, 2001.

[66] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. One-pass wavelet decompositions of data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):541–554, 2003.

[67] R. G. Godsil C.D. *Algebraic Graph Theory*. Springer, 2001.

[68] G. Golub and C. Van Loan. *Matrix Computations*. JohnsHopkinsPress, Baltimore, MD, second edition, 1989.

[69] G. H. Golub and C. F. Van Loan. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA, 1996.

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

[71] R. L. Grossman and Y. Gu. Data mining using high performance data clouds: experimental studies using sector and sphere. *KDD*, 2008.

[72] R. Harshman. Foundations of the parafac procedure: Models and conditions for an "exploratory" multimodal factor analysis. *UCLA Working Papers in Phonetics*, 1970.

[73] P. Hintsanen and H. Toivonen. Finding reliable subgraphs from large probabilistic graphs. *PKDD*, 2008.

[74] D. Hirschberg, A. Chandra, and D. Sarwate. Computing connected components on parallel computers. *Communications of the ACM*, 22(8):461–464, 1979.

[75] E. Hoke, J. Sun, J. D. Strunk, G. R. Ganger, and C. Faloutsos. Intemon: Continuous mining of sensor data in large-scale self-* infrastructures. *ACM SIGOPS Operating Systems Review*, 40(3), 2003.

[76] R. Horn and C. R. Johnson. *Matrix Analysis*. pub-CAMBRIDGE, 1985.

[77] A. Itai and M. Rodeh. Finding a minimum circuit in a graph. In *STOC '77: Proceedings of the ninth annual ACM symposium on Theory of computing*, pages 1–10, New York, NY, USA, 1977. ACM.

[78] D. J. *Applied Numerical Linear Algebra*. SIAM, Philadelphia, PA, 1997.

[79] H. Jowhari and M. Ghodsi. New streaming algorithms for counting triangles in graphs. In *COCOON*, pages 710–716, 2005.

[80] U. Kang, C. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec. Radius plots for mining tera-byte scale graphs: Algorithms, patterns, and observations. *SIAM International Conference on Data Mining*, 2010.

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

[82] G. Karypis and V. Kumar. Parallel multilevel k-way partitioning for irregular graphs. *SIAM Review*, 41(2):278–300, 1999.

[83] Y. Ke, J. Cheng, and J. X. Yu. Top-k correlative graph mining. *SDM*, 2009.

[84] N. S. Ketkar, L. B. Holder, and D. J. Cook. Subdue: Compression-based frequent pattern discovery in graph data. *OSDM*, August 2005.

[85] H. Kiers. Some procedures for displaying results from three-way methods. *Journal of chemometrics*, 2000.

[86] J. H. Kim and V. H. Vu. Concentration of multivariate polynomials and its applications. *Combinatorica*, 20(3):417–434, 2000.

[87] J. Kleinberg. Authoritative sources in a hyperlinked environment. In *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms*, 1998. Also appears as IBM Research Report RJ 10076, May 1997.

[88] D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 10 Jan. 1981.

[89] T. Kolda and B. Bader. The TOPHITS model for higher-order web link analysis. In *SDM Workshops*, 2006.

[90] T. G. Kolda and B. W. Bader. Tensor decompositions and applications. *SIAM Review*, 51(3), September 2009. In press.

[91] T. G. Kolda, B. W. Bader, and J. P. Kenny. Higher-order web link analysis using multilinear algebra. In *ICDM*, 2005.

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

[93] T. G. Kolda and J. Sun. Scalable tensor decompsitions for multi-aspect data mining. *ICDM*, 2008.

[94] P. M. Kroonenberg. *Applied Multiway Data Analysis*. Wiley, 2008.

[95] J. B. Kruskal. Rank, decomposition, and uniqueness for 3-way and n-way arrays. pages 7–18, 1989.

[96] M. Kuramochi and G. Karypis. Finding frequent patterns in a large sparse graph. *SIAM Data Mining Conference*, 2004.

[97] R. Lämmel. Google's mapreduce programming model – revisited. *Science of Computer Programming*, 70:1–30, 2008.

[98] M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor. Comput. Sci.*, 407(1-3):458–473, 2008.

[99] L. D. Lathauwer, B. D. Moor, and J. Vandewalle. On the best rank-1 and rank-(r1,r2,. . .,rn) approximation of higher-order tensors. *SIAM J. Matrix Anal. Appl.*, 21(4):1324–1342, 2000.

[100] 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.

[101] J. Leskovec and C. Faloutsos. Scalable modeling of real graphs using kronecker multiplication. In *ICML '07: Proceedings of the 24th international conference on Machine learning*, pages 497–504, New York, NY, USA, 2007. ACM.

[102] J. Leskovec and E. Horvitz. Planetary-scale views on an instant-messaging network, Mar 2008.

[103] D. Luo, H. Huang, and C. Ding. Are tensor decomposition solutions unique? on the global convergence of hosvd and parafac algorithms, 2009.

[104] J. Ma and S. Ma. Efficient parallel algorithms for some graph theory problems. *JCST*, 1993.

[105] M. W. Mahoney and P. Drineas. Cur matrix decompositions for improved data analysis. *Proceedings of the National Academy of Sciences*, 2009.

[106] M. W. Mahoney, M. Maggioni, and P. Drineas. Tensor-cur decompositions for tensor-based data. In *KDD*, 2006.

[107] M. Mcglohon, L. Akoglu, and C. Faloutsos. Weighted graphs and disconnected components: patterns and a generator. *KDD*, 2008.

[108] M. Mcpherson, L. S. Lovin, and J. M. Cook. Birds of a feather: Homophily in social networks. *Annual Review of Sociology*, 27(1):415–444, 2001.

[109] M. Mihail and C. Papadimitriou. the eigenvalue power law, 2002.

[110] A. Mislove, M. Marcon, K. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. page 42, 2007.

[111] D. Muti and S. Bourennane. Survey on tensor signal algebraic filtering. *Signal Process.*, 87(2):237–249, 2007.

[112] M. E. J. Newman. The structure and function of complex networks. *SIAM Review*, 45:167–256, 2003.

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

[114] 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, New York, NY, USA, 2008.

[115] 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.

[116] J.-Y. Pan, H.-J. Yang, C. Faloutsos, and P. Duygulu. Automatic multimedia cross-modal correlation discovery. *ACM SIGKDD*, Aug. 2004.

[117] G. Pandurangan, P. Raghavan, and E. Upfal. Using pagerank to characterize web structure. *COCOON*, August 2002.

[118] C. H. Papadimitriou, P. Raghavan, H. Tamaki, and S. Vempala. Latent semantic indexing: A probabilistic analysis. In *PODS*, 1998.

[119] C. H. Papadimitriou and M. Yannakakis. The clique problem for planar graphs. *Inf. Process. Lett.*, 13(4/5):131–133, 1981.

[120] S. Papadimitriou, A. Brockwell, and C. Faloutsos. Adaptive, hands-off stream mining. *VLDB*, Sept. 2003.

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

[122] S. Papadimitriou, J. Sun, and C. Faloutsos. Streaming pattern discovery in multiple time-series. In *VLDB '05*, pages 697–708. VLDB Endowment, 2005.

[123] 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.

[124] R. Pike, S. Dorward, R. Griesemer, and S. Quinlan. Interpreting the data: Parallel analysis with sawzall. *Scientific Programming Journal*, 2005.

[125] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, 2nd edition, 1992.

[126] T. Qian, J. Srivastava, Z. Peng, and P. C. Sheu. Simultaneouly finding fundamental articles and new topics using a community tracking method. *PAKDD*, 2009.

[127] S. Ranu and A. K. Singh. Graphsig: A scalable approach to mining significant subgraphs in large graph databases. *ICDE*, 2009.

[128] T. Sarlos. Improved approximation algorithms for large matrices via random projections. In *FOCS '06: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 143–152, Washington, DC, USA, 2006. IEEE Computer Society.

[129] T. Sarlos. Improved approximation algorithms for large matrices via random projections. In *FOCS '06*, pages 143–152, Washington, DC, USA, 2006. IEEE Computer Society.

[130] V. Satuluri and S. Parthasarathy. Scalable graph clustering using stochastic flows: applications to community discovery. *KDD*, 2009.

[131] B. Savas and L.-H. Lim. Quasi-Newton methods on Grassmannians and multilinear approximations of tensors. *Submitted to SIAM Journal on Optimization*, 2009.

[132] T. Schank. *Algorithmic Aspects of Triangle-Based Network Analysis*. Phd in computer science, University Karlsruhe, 2007.

[133] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *WEA*, pages 606–609, 2005.

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

[135] N. Shrivastava, A. Majumder, and R. Rastogi. Mining (social) network graphs to detect random link attacks. *ICDE*, 2008.

[136] 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.

[137] A. Smilde, R. Bro, and P. Geladi. *Multi-way Analysis: Applications in the Chemical Sciences*. Wiley, 2004.

[138] D. A. Spielman and N. Srivastava. Graph sparsification by effective resistances. Mar 2008.

[139] G. Strang. *Linear Algebra and Its Applications*. Academic Press, 2nd edition, 1980.

[140] J. Sun, C. Faloutsos, S. Papadimitriou, and P. S. Yu. Graphscope: parameter-free mining of large time-evolving graphs. In *KDD*, 2007.

[141] J. Sun, S. Papadimitriou, and P. Yu. Window-based tensor analysis on high-dimensional and multi-aspect streams. In *Proceedings of the International Conference on Data Mining (ICDM)*, 2006.

[142] J. Sun, D. Tao, and C. Faloutsos. Beyond streams and graphs: dynamic tensor analysis. In *KDD '06*, 2006.

[143] J. Sun, C. Tsourakakis, E. Hoke, C. Faloutsos, and T. Eliassi-Rad. Two heads better than one: pattern discovery in time-evolving multi-aspect data. *Data Mining and Knowledge Discovery*, 2008.

[144] J.-T. Sun, H.-J. Zeng, H. Liu, Y. Lu, and Z. Chen. Cubesvd: a novel approach to personalized web search. In *WWW*, pages 382–390, 2005.

[145] T. Tao and V. Vu. *Additive Combinatorics.* Cambridge Univ., 2006.

[146] J. tao Sun, H.-J. Zeng, H. Liu, and Y. Lu. Cubesvd: A novel approach to personalized web search. In *WWW*, 2005.

[147] Thomas Schank and Dorothea Wagner . DELIS-TR-0043 - finding, counting and listing all triangles in large graphs, an experimental study. techreport 0043, submitted, 2004.

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

[149] C. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos. Spectral counting of triangles in power-law networks via element-wise sparsification. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, 2009.

[150] C. E. Tsourakakis. Mach: Fast randomized tensor decompositions. *CoRR*, abs/0909.4969, 2009.

[151] C. E. Tsourakakis. Counting triangles using projections. *KAIS Journal*, 2010.

[152] C. E. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos. Spectral counting of triangles in power-law networks via element-wise sparsification. In *ASONAM*, pages 66–71, 2009.

[153] C. E. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos. Spectral counting of triangles in power-law networks via element-wise sparsification and triangle based link recommendation. In *ASONAM*, 2010.

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

[155] C. E. Tsourakakis, M. N. Kolountzakis, and G. L. Miller. Approximate triangle counting. Apr 2009.

[156] L. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 1966.

[157] M. A. O. Vasilescu and D. Terzopoulos. Multilinear analysis of image ensembles: Tensorfaces. In *ECCV*, 2002.

[158] M. A. O. Vasilescu and D. Terzopoulos. Multilinear analysis of image ensembles: Tensorfaces. In *ECCV*, pages 447–460, 2002.

[159] J. S. Vitter. Faster methods for random sampling. *Commun. ACM*, 27(7):703–718, 1984.

[160] V. H. Vu. On the concentration of multi-variate polynomials with small expectation, 1999.

[161] C. Wang, W. Wang, J. Pei, Y. Zhu, and B. Shi. Scalable mining of large disk-based graph databases. *KDD*, 2004.

[162] N. Wang, S. Parthasarathy, K.-L. Tan, and A. K. H. Tung. Csv: Visualizing and mining cohesive subgraph. *SIGMOD*, 2008.

[163] Y. Wang, D. Chakrabarti, C. Faloutsos, C. Wang, and C. Wang. Epidemic spreading in real networks: An eigenvalue viewpoint. In *In SRDS*, pages 25–34, 2003.

[164] S. Wasserman and K. Faust. *Social network analysis*. Cambridge University Press, Cambridge, 1994.

[165] D. Xu, S. Yan, L. Zhang, H.-J. Zhang, Z. Liu, and H.-Y. Shum. Concurrent subspaces analysis. In *CVPR*, 2005.

[166] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. *ICDM*, 2002.

[167] X. Yan, P. S. Yu, and J. Han. Graph indexing: A frequent structure-based approach, 2004.

[168] C. H. You, L. B. Holder, and D. J. Cook. Learning patterns in the dynamics of biological networks. In *KDD*, 2009.

[169] F. Zhu, X. Yan, J. Han, and P. S. Yu. gprune: A constraint pushing framework for graph pattern mining. *PAKDD*, 2007.