Counting Triangles in Large Graphs using Randomized Matrix Trace Estimation

Haim Avron*
School of Computer Science
Tel-Aviv University
haima@post.tau.ac.il

ABSTRACT
Triangle counting is an important problem in graph mining with several real-world applications. Interesting metrics, such as the clustering coefficient and the transitivity ratio, involve computing the number of triangles. Furthermore, several interesting graph mining applications rely on computing the number of triangles in a large-scale graph. However, exact triangle counting is expensive and memory consuming, and current approximation algorithms are unsatisfactory and not practical for very large-scale graphs. In this paper we present a new highly-parallel randomized algorithm for approximating the number of triangles in an undirected graph. Our algorithm uses a well-known relation between the number of triangles and the trace of the cubed adjacency matrix. A Monte-Carlo simulation is used to estimate this quantity. Each sample requires \( O(|E|) \) time and \( O(\epsilon^{-2} \log(1/\delta) \rho(G)^2) \) samples are required to guarantee an \((\epsilon, \delta)\)-approximation, where \( \rho(G) \) is a measure of the triangle sparsity of \( G \) (\( \rho(G) \) is not necessarily small). Our algorithm requires only \( O(|V|) \) space in order to work efficiently. We present experiments that demonstrate that in practice usually only \( O(\log^2 |V|) \) samples are required to get good approximations for graphs frequently encountered in data-mining tasks, and that our algorithm is competitive with state-of-the-art approximate triangle counting methods both in terms of accuracy and in terms of running-time. The use of Monte-Carlo simulation support parallelization well: our algorithm is embarrassingly parallel with a critical path of only \( O(|E|) \), achievable on as few as \( O(\log^2 |V|) \) processors.

Categories and Subject Descriptors
F.2.0 [Analysis of Algorithms and Problem Complexity]: General; G.2.1 [Combinatorics]: Counting problems

*This research was supported in part by an IBM Faculty Partnership Award and by grant 1045/09 from the Israel Science Foundation (founded by the Israel Academy of Sciences and Humanities).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

KDD-LDMTA '10, July 25, 2010, Washington, DC, USA.
Copyright 2010 ACM 978-1-4503-0215-9/10/07 ...$10.00.

1. INTRODUCTION
The use of graphs to model many real-world phenomena (like the World Wide Web, social networks and protein interaction networks) gives rise to many different graph-mining tasks. A very common graph-mining task is counting the number of triangles in the graph (i.e., the number of connected vertex triplets), because the number of triangles is closely connected with estimating the clustering coefficients and the transitivity ratio of the graph. Counting the number of triangles has other applications. In explorations of networks, triangles are a frequently used network statistic in the exponential random graph model [14, 19] and naturally appear in models of real-world network evolution [17]. Triangles have been used in several applications such as spam detection [7] and uncovering the hidden thematic layers in the web [12]. In the context of social networks triangles have a natural interpretation: friends of friends tend to be friends [20], and this can be used in link recommendation/friend suggestion [24].

However, counting triangles in large-scale graphs, with millions and billions of edges, is expensive. It is not practical to count triangles with a single machine, and parallel algorithms are required. Furthermore, the graph may be too large to hold in memory and streaming algorithms are needed (the total number of indexed nodes in the Web is in the order of 10^{10}, and the typical number of links per Web page is between 20 and 30 [7]).

In this paper we present TraceTriangle, a new embarrassingly-parallel randomized algorithm for approximating the number of triangles. TraceTriangle is based on a numerical approximation of the trace (sum of diagonal value) of \( A^3 \), where \( A \) is the adjacency matrix of the graph, using a Monte-Carlo simulation. It is a well-known fact that the number of triangles is equal to trace(\( A^3 \))/6. Each Monte-Carlo sample requires \( O(|E|) \) operations. We present three variants, based with a different sampling space. The first variant is theoretically the best, requiring \( O(\epsilon^{-2} \log(1/\delta) \rho(G)^2) \) to guarantee an \((\epsilon, \delta)\)-approximation, where \( \rho(G) \) is a measure of the triangle sparsity (\( \rho(G) \) can be very large, but it seems that often it is not too large). Weaker bounds are established for other variants, but these variants have advantages in
addition to the space required for storing the graph. The

Table 1: Notations and symbols used throughout
the paper.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G = (V, E)$</td>
<td>Undirected graph, no loops.</td>
</tr>
<tr>
<td>$n =</td>
<td>V</td>
</tr>
<tr>
<td>$\Delta(G)$</td>
<td># of triangles in $G$.</td>
</tr>
<tr>
<td>$A$</td>
<td>Adjacency matrix of $G$.</td>
</tr>
<tr>
<td>$\lambda_1, \ldots, \lambda_n$</td>
<td>Eigenvalues of $A$.</td>
</tr>
<tr>
<td>$\text{trace}(M)$</td>
<td>Sum of diagonal elements, $\text{trace}(M) = \sum_{i=1}^{\omega} M_{ii} = \sum_{i=1}^{\omega} \lambda_i$.</td>
</tr>
</tbody>
</table>

terms of implementation. Only $O(|V|)$ space is required in
addition to the space required for storing the graph. The
algorithm can be implemented efficiently when the graph is
kept in secondary storage, therefore effectively only $O(|V|)$
space is needed. TraceTriangle is embarrassingly parallel:
most of the work can be done in parallel without any
synchronization. The critical path is $O(|E|)$ (the work re-
quired for a single sample).

We report an experimental evaluation of TraceTriang-
le. We show that in practice usually only $O(\log^2 |V|)$ sam-
ple are required to get good approximation for graphs fre-
cently encountered in data-mining applications. We com-
pare TraceTriangle to the state-of-the-art method Eigen-
Triangle [23] and show that TraceTriangle is competi-
tive both in terms of accuracy and in terms of running-time.

2. RELATED WORK

2.1 Triangle counting algorithms

A typical strategy for counting triangles is to list all trian-
gles. The most widely-used listing methods are the NodeIt-
erator and the EdgeIterator algorithms. The NodeIt-
erator algorithm considers each one of the nodes and ex-
amines which pairs of its neighbors are connected. The
EdgeIterator algorithm computes for each edge the num-
ber of triangles that contain it. Asymptotically, both meth-
ods have the same time complexity: $\Theta(\sum_{v \in V} \deg(v)^2)$, where $\deg(v)$ is the degree of $v$. NodeIterator and EdgeIterator’s have two major drawbacks when it comes to count-
ing triangles in very large-scale graphs. First, they do not
parallelize well. It is hard to load-balance work and space
requirements between compute units. We are not aware of
any paper presenting an efficient parallel implementation of
NodeIterator and EdgeIterator. Secondly, both algo-

Theoretically most efficient counting algorithm is due
to Alon, Yuster and Zwick [4]. It is based on fast matrix
multiplication, and runs in $\Theta(|E|^{\omega/2})$ where $\omega$ is the fast
matrix multiplication exponent. Thus, the Alon et al. algo-
rithm currently runs in $O(|E|^{1.41})$ time. Space complexity is $\Theta(|V|^2)$ which is prohibitive for large graphs.

In many cases an exact count of triangles is not crucial, and
an high quality approximation will suffice. The goal
such an algorithm is to produce a relative $c$-approximation
of the number of triangles with high probability. Most
of the approximate triangle counting algorithms have been
developed in the streaming model, i.e. these algorithms make
only a constant number of passes over the input, usually by
employing some form of Monte-Carlo simulation.

The simplest method uses na"ive sampling: three random
nodes are tested whether they form a triangle [21]. The total
number of triangles is estimated as the fraction of the triples
that formed a triangle. For a fixed $c$ and failure probability
the number of samples required to find an $c$-approximation
is $\Theta(\frac{3}{c^2} + \frac{T_1 + T_2}{c})$ where $T_i = \#of$ triples with $i$ edges. For
graphs with $\Omega(|V|^2)$ triangles this approach is not suitable.
Unfortunately, this is the typical case when dealing with
real-world networks.

A much more subtle approach was presented in [6]. The
authors reduce the problem of triangle counting efficiently
to estimating moments for a stream of node triples. They
then use algorithm presented by Alon-Matias-Szegedy [3] to
proceed. Along the same lines, Buriol et al. [8] proposed
two space-bounded sampling algorithms to estimate the number
of triangles. Again, the underlying sampling procedures are
simple. 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 Monte-Carlo algorithms to the best of our knowledge,
but they still require the graph to be fairly dense in order
to get a good approximation.

Recently, a new triangle counting accelerator has been
suggested by Tsourakakis et al. [25]. The algorithm ran-
domly throws out a fraction of the edges, and then counts
the number of triangles in the remaining graph (using any
other triangle counting algorithm). An approximation for
the number of triangles in the full graph can be computed
from the exact count of triangles in the smaller graph. More
theoretical analysis is provided in [9]. The aim of their algo-
rithm is to enable the acceleration of a core triangle counting
algorithm: after the initial sparsification the actual number
of triangles is computed using some other triangle counting
algorithm. The triangle sparsification, can, of course, be
parallelized quite easily. But in order to fully parallelize a
triangle counting code the core algorithm must be parallel as
well. Tsourakakis et al. do not suggest any new way to par-
allelize the core algorithm. Furthermore, they not solve the
issue of memory load. The number of bytes held in memory
drops, but only by a constant factor. It is still the case that
the entire (sparsified) graph must be kept in main memory
for an efficient implementation.

2.2 Spectral counting

Tsourakakis recently presented a method based on com-
puting eigenvalues [23]. It is based on the following obser-
vation [10].

Lemma 1. The total number of triangles $\Delta(G)$ in an un-
directed graph is equal to the trace of $A^3$ and to the sum of the
cubes of the eigenvalues of the adjacency matrix divided by
six, i.e.:

$$\Delta(G) = \frac{1}{6} \text{trace}(A^3) = \frac{1}{6} \sum_{i=1}^{n} \lambda_i^3.$$ 

Tsourakakis observed that the absolute values of the eigen-
values tend to be skewed, typically following a power law [13,
18, 11]. Therefore, the number of triangles can be well ap-
proximated using the largest magnitude eigenvalues. The
algorithm, EigenTriangle, uses a Lanczos iteration to find
largest magnitude eigenvalues, and if implemented using
state-of-the-art software like ARPACK it is very fast. The method main drawbacks are: it is hard to determine how many eigenvalues need to be computed, there are no approximation guarantees, nor are there any guarantees on the number of steps the algorithm will do. We also note that there is a data-dependency between Lanczos iterations. The algorithm can be parallelized by parallelizing the matrix-vector multiplication, an operation whose parallelism is usually modeled as the ability to partition a graph. For some sparse matrices only very mediocre parallelism can be achieved. For some treatment on the subject see [15]. We do not know if matrices frequently encountered in graph mining tasks exhibit this problem, and we are unaware of any study on these issues. Here it should be noted that the use of SCALAPACK is not an option as it is targeted for dense matrices, while the matrices involved in triangle counting are sparse.

In contrast, in our method there is no data-dependency between matrix-vector multiplication, so our algorithm can benefit from both the parallelism of the matrix-vector operation, and the parallelism inherent in the use of independent Monte-Carlo samples.

2.3 Numerical trace estimation

Computing the trace of an explicit matrix is a computationally cheap operation. However, in some cases the matrix can only be given implicitly as an operation on vectors. Counting triangles falls into that category: for large real networks it is impossible to actually form $A^3$. The standard Monte-Carlo simulation for estimating the trace of an implicit matrix is due to Hutchinson [16], who proves the following Lemma.

**Lemma 2.** Let $A$ be an $n \times n$ symmetric matrix with trace($A$) $\neq 0$. Let $z$ be a random vector whose entries are i.i.d Rademacher random variables ($Pr(z_i = \pm 1) = 1/2$). $z^T A z$ is an unbiased estimator of trace($A$) i.e., $E(z^T A z) = \text{trace}(A)$ and $\text{Var}(z^T A z) = 2 \left( \|A\|_F^2 - \sum_{i=1}^n A_{ii}^2 \right)$.

It is easy to see that for a general matrix Hutchinson’s method can be ineffective because the variance can be arbitrarily large. Even for a symmetric positive definite the variance can be large: the variance for the matrix of all ones is $\sum_{i=1}^n A_{ii}^2$. Such a large variance precludes the use of Chebyshev’s inequality to establish a bound on the number of iterations required to obtain a given relative error in the trace. For such a bound to hold, the variance must be $o(\text{trace}(A)^2)$. It is hard to use Lemma 2 to give rigorous bounds on the number of samples/matrix multiplications, and this difficulty carries over to applications of this method.

Recently, rigorous bounds have been discovered using Chernoff-type analysis [5]. If the entries of $z$ are i.i.d standard normal variables then $O(\varepsilon^{-2} \log(1/\delta))$ samples are required for an $(\varepsilon, \delta)$-approximation for a positive definite matrix. For i.i.d Rademacher random variables (i.e., Hutchinson’s method) $O(\varepsilon^{-2} \log(n/\delta))$ samples are required for a positive definite matrix. The analyses in this paper are based on these results.

3. ALGORITHM

We now describe the first variant of TraceTriangle and state its properties. This is the variant with the best bounds. The other two variants, which describe in Sections 4 and 5, will enjoy some advantages but have weaker bounds. In order to shorten the expressions in the complexity analysis we assume that $|E| \geq |V|$, although it is easy to generalize the results for other cases.

We propose Algorithm 1, which we call TraceTriangle. The subscript $(N)$ denotes that this variant uses standard normal variables in line 4. TraceTriangle uses Monte-Carlo simulation to estimate the trace of $A^3$. Instead of using vectors composed of Rademacher random variables standard normal variables are used, since they allow us to prove better bounds. The number of Monte-Carlo samples is set to $O(\log^2 |V|)$. For a large class of graphs this is sufficient, although it might not be sufficient for graphs with very few triangles.

The following definition and theorem summarizes the main properties of TraceTriangle. The definition describes a measure of how sparse triangles are in the graph.

**Definition 1.** The eigenvalue triangle sparsity of a graph is $G$ is

$$
\rho(G) = \sum_{i=1}^n |\lambda_i^3| \over 6\Delta(G).
$$

If there are no triangles in the graph then $\rho(G) = \infty$.

**Theorem 1.** Given a fixed approximation ratio $\varepsilon > 0$, fixed probability of failure $\delta > 0$ and fixed sparsity growth ratio $C \geq 1$ there exists a $\gamma = f(\varepsilon, \delta, C)$ such that TraceTriangle with $\gamma$ as the parameter in line 2 on undirected graphs for which $\rho(G) \leq C \log |V|$ will return $\Delta$ such that

$$
(1 - \varepsilon)\Delta(G) \leq \Delta \leq (1 + \varepsilon)\Delta(G)
$$

with probability of at least $1 - \delta$. Furthermore, $\gamma = O(\varepsilon^{-2} C^2 \log(1/\delta))$. The running time of the algorithm is $\Theta(|E| \log^2 |V|)$ (for a fixed $\varepsilon$, $\delta$ and $C$) and it requires only $\Theta(|V|)$ space in addition to the space required for keeping the matrix (which is $O(|E|)$).

TraceTriangle guarantees an $(\varepsilon, \delta)$-approximation only for graphs with logarithmically bounded triangle sparsity (i.e., $\rho(G) = O(\log |V|)$). The number of samples could have been decoupled from an a-priori bound on the sparsity by setting $M = O(\varepsilon^{-2} \rho(G)^2 \log(1/\delta))$, but the quantity $\rho(G)$ is unknown and probably impossible to approximate as it involves two unknown quantities: the number of triangles and $\sum_{i=1}^n |\lambda_i^3|$, so instead we decided to limit the number of samples and characterize the class of graphs for which our algorithms finds an $(\varepsilon, \delta)$ approximation. Furthermore, even if $\rho(G)$ was known in advance it would be impractical.

**Algorithm 1 TraceTriangle**

$\Delta = \text{TraceTriangle}_{(0)}(G, \text{undirected graph with } n \text{ nodes})$

$\triangleright \text{ parameter: } \gamma.$

1. Form the adjacency matrix $A \in \mathbb{R}^{n \times n}$.
2. $M = [\gamma \ln^2 n]$
3. Do $M$ times, possibly in parallel
4. Form the vector $x = (x_k)$, where $x_i \sim N(0, 1)$ are i.i.d random variables.
5. $y \leftarrow A x$
6. $T_i \leftarrow (y^T A y) / 6,$
   where $i$ is the index of the do loop
7. end
8. $\Delta \leftarrow \frac{1}{M} \sum_{i=1}^M T_i$
to use the explicit formula for $\gamma$. Reasonable values like $\epsilon = 0.05$ and $\delta = 0.05$ will result in a huge sample size. In practice, as the experiments shown in Section 6 show, we noticed that $O(\log^2 |V|)$ usually gives good approximations for graphs encountered in data-mining applications, so this is the sample size we used, thereby bounding running time to $\Theta(|E| \log^2 |V|)$.

We do, however, prove a simple, yet not fully satisfactory, bound on $\sum_{i=1}^n |\lambda^2_i|$.

**Proposition 1.** Let $A$ be the adjacency matrix of an undirected graph $G = (V, E)$. Let $\lambda_1, \ldots, \lambda_n$ be $A$’s eigenvalues. The following hold

$$\sum_{i=1}^n |\lambda^2_i| \leq \sqrt{8 |E|^{3/2}}.$$

**Proof.** Let $x$ be the vector composed of the absolute eigenvalues. Since $||x||_2^2 = ||A||_F^2 = |E|$ and $||x||_3 \leq ||x||_2$ we have

$$\sum_{i=1}^n |\lambda^2_i| = ||x||_3^2 \leq ||x||_2^3 = \sqrt{8 |E|^{3/2}}.$$

\[ \square \]

Real-world graphs usually obey a power-law distribution of the degrees. This implies that $|E| \approx c |V|^{(13)}$ which in turn implies that if $\Delta(G) = \Omega(V^{(13)/2} / \log |V|)$ our algorithm will achieve a $(\epsilon, \delta)$-approximation with a constant $\gamma$ (depending on $\epsilon$, $\delta$, and $c$). The relation between number of triangles and the number of vertices and edges is not clear. Sergi [22] analyzes the number of triangles in random graph models and in some cases it is $\Omega(|E|^{3/2})$ but sometimes it is less. Tsourakakis [23] experimentally shows that the number of triangles per vertex in real-life networks follows a power-law with respect to the vertex degree. His experiments suggest that for real-life networks $\Delta(G) \sim \sum_{v \in V} \deg(v)^\beta$ where $\beta \geq 1.5$. In any case this is already an improvement over older randomized counting algorithm which require $\Delta(G) = \Omega(|V|^2)$.

**TraceTriangle** has a few other desirable properties. Unlike some approximation methods, like **EIGENTRIANGLE**, there is no threshold: the number of steps needed for completion is known in advance. The number of random bits required is only $O(|V| \log^2 |V|)$. It is embarrassingly parallel: each of the samples can be processed in parallel without any synchronization. The only step that requires synchronization is the summation in step 8. Even without parallelizing the implicit matrix multiplication the critical path is $O(|E|)$, which can be achieved with only $O(\log^2 |V|)$ processors. **TraceTriangle** is also a streaming algorithm (i.e., it can be implemented with $O(1)$ passes over the graph), although as a streaming algorithm the space requirement is $O(|V| \log^2 |V|)$ which is very large. As a semi-streaming algorithm space requirement are $O(|V| \log |V|)$ and $O(\log |V|)$ passes are needed. $O(|V|)$ space is achievable with $O(\log^2 |V|)$ passes. For some graphs the space requirements may be large, but luckily **TraceTriangle** is also embarrassingly parallel in its memory usage: with $\log^2 |V|$ independent machines (a cluster) each machine will use only $O(|V|)$ space. Connected to this issue, is the observation that **TraceTriangle** traverses the graph sequentially, so the graph itself can be kept in secondary storage without paying a big penalty. This is especially important for very large-scale graphs that cannot be held in main memory.

We now present the analysis of the trace estimator. Theorem 1 readily follows from this analysis. The first lemma shows that by using Gaussian variables we still get an unbiased estimator. We omit the proof.

**Lemma 3.** Let $A$ be an $n \times n$ symmetric matrix with trace$(A) \neq 0$. Let $z$ be a random vector whose entries are i.i.d standard normal random variables. $z^T A z$ is an unbiased estimator of trace$(A)$ i.e., $E(z^T A z) = \text{trace}(A)$ and $\text{Var}(z^T A z) = 2 \|A\|^2_F$.

The variance is not small enough to establish a bound on the error. Using Chernoff-type arguments we can bound the error.

**Definition 2.** A Gaussian trace estimator for a symmetric matrix $A \in \mathbb{R}^{n \times n}$ is

$$G_M = \frac{1}{M} \sum_{i=1}^M z_i^T A z_i,$$

where the $z_i$’s are $M$ independent random vectors whose entries are i.i.d. standard normal variables.

**Notation 1.** For a symmetric matrix $X \in \mathbb{R}^{n \times n}$ with eigenvalues $\lambda_1, \ldots, \lambda_n$ we denote $\rho(X) = \frac{\sum_{i=1}^n |\lambda_i|}{\text{trace}(X)}$.

Note that if $A$ is the adjacency matrix of a graph $G$ then $\rho(G) = \rho(A^T)$.

**Lemma 4.** Let $\delta > 0$ be a failure probability and let $\epsilon > 0$ be a relative error. For $M \geq 20 \epsilon^{-2} \rho(A)^2 \ln(4/\delta)$, the Gaussian trace estimator $G_M$ of a symmetric matrix $A \in \mathbb{R}^{n \times n}$ satisfies

$$\Pr(\|G_M - \text{trace}(A)\| \leq \epsilon |\text{trace}(A)|) \geq 1 - \delta.$$ 

**Proof.** $A$ is symmetric so it can be diagonalized. Let $A = U^T A U$ be the unitary diagonalization of $A$ (its eigendecomposition), and define $y_i = U z_i$. Since $U$ is a rotation matrix the entries of $y_i$ are i.i.d Gaussian variables. Let $\lambda_1, \ldots, \lambda_n$ be the eigenvalues of $A$, and assume that $\lambda_1, \ldots, \lambda_\beta$ are negative and $\lambda_{\beta+1}, \ldots, \lambda_n$ are positive. Notice that $G_M = \frac{1}{M} \sum_{i=1}^M \sum_{j=1}^n \lambda_i y_{ij}$ where $y_{ij}$ is the $j$’th entry of $y_i$. Let us write $\tau_+ = \sum_{j=\beta+1}^n \lambda_j$ and $G_M^+ = \sum_{i=1}^M \sum_{j=\beta+1}^n \lambda_j y_{ij}$. We will show that if $M \geq 12 \epsilon^{-2} \rho(A)^2 \ln(4/\delta)$ we have

$$\Pr(\|G_M^+ - \tau_+ - \rho(A)\| \leq \epsilon |\tau_+|/\rho(A)) \geq 1 - \delta/2.$$ 

The bound will be proved using a Chernoff-style argument. Since $y_{ij}$ is standard normal then $\sum_{i=1}^M y_{ij}^2 \sim \chi^2(M)$. Therefore, the moment generating function of $Z = M G_M^+$ is

$$m_Z(t) = E(\exp(tZ)) = \prod_{t_{i=1}^M} (1 - 2 \lambda_i t)^{-M/2} = (1 - 2 \tau t + h(t))^M$$

where

$$h(t) = \sum_{s=2}^n (-2)^s t^s \sum_{S \subseteq \Lambda_1(A) \atop |S| = s} \prod_{x \in S} \lambda_x$$

$$\leq \Lambda_1(A)$$
as long as \(|\lambda_it| \leq \frac{1}{2} \) (\(\Lambda^+ (A)\) is the set of positive eigenvalues of \(A\)).

It is easy to see if \(x_1, \ldots, x_n\) is a set of non-negative real numbers. For all \(i = 1, \ldots, n\) we have

\[
\sum_{S \subseteq \{1, \ldots, n\}} \prod_{j \in S} x_j \leq \left( \sum_{i=1}^n x_i \right)^n.
\]

Therefore, we can bound

\[
|h(t)| \leq \sum_{j=2}^{\infty} (2\tau t)^j.
\]

Denote \(\epsilon_0 = \epsilon / \rho(A)\). Set \(t = \epsilon_0 / 4^\tau (1 + \epsilon_0 / 2)\). For \(i = r + 1, \ldots, n\) we have \(\lambda_i t \leq \frac{1}{2}\) so (1) is the correct formula for \(\text{m}_Z(t)\). We have

\[
|h(t)| \leq \sum_{j=2}^{\infty} \left( \frac{\epsilon_0}{2(1 + \epsilon_0 / 2)} \right)^j \leq \frac{\epsilon_0}{4(1 + \epsilon_0 / 2)}.
\]

Markov’s inequality asserts that

\[
\Pr (|G_M| \geq \tau_+ (1 + \epsilon_0)) = \Pr (Z \geq \tau_+ M(1 + \epsilon_0)).
\]

\[
\leq m_Z(t) \exp(-\tau_+ M(1 + \epsilon_0) t) \leq (1 - \epsilon_0 / 2(1 + \epsilon_0 / 2)) - \epsilon_0^2 / 4(1 + \epsilon_0 / 2)^2 \cdot \exp(-\tau_+ \frac{\epsilon_0}{2} \cdot 1 + \epsilon_0 / 2)
\]

\[
= \exp(-\frac{\epsilon_0}{2} \ln(1 - \epsilon_0 / 2)) + \frac{\epsilon_0}{2} \cdot 1 + \epsilon_0 / 2)
\]

\[
\leq \exp(-\frac{\epsilon_0}{20}).
\]

We now propose another variant of TraceTriangle, which is called TraceTriangleE. In this variant we replace the standard normal distribution with the Rademacher distribution. That is, in line 4 of Algorithm 1 \(x_i\) is \(\pm 1\) with equal probability.

There are two reasons why this change makes sense. By using vectors composed of i.i.d Rademacher variables we are using the Hutchinson estimator which has a smaller variance than the Gaussian estimator (compare lemma 2 to lemma 3). The second reason is related to implementation. It is more expensive to generate samples from a standard normal distribution then it is to generate samples from the Rademacher distribution. Furthermore, in some settings, for example a database setting, implementing TraceTriangleE can be non-trivial, while working with \(\pm 1\) is much easier [1]. The following lemma analyzes convergence of Hutchinson’s estimator. We omit the proof, and refer the reader to [5] for analysis of the positive definite case.

**Definition 3.** An Hutchinson trace estimator for a symmetric matrix \(A \in \mathbb{R}^{n \times n}\) is

\[
H_M = \sum_{i=1}^n z_i^T A z_i,
\]

where the \(z_i\)’s are \(M\) independent random vectors whose entries are i.i.d Rademacher random variables

**Lemma 5.** Let \(\delta > 0\) be a failure probability and let \(\epsilon > 0\) be a relative error. For \(M \geq 6e^{-2} \rho(A)^2 \ln(2 \text{rank}(A)/\delta)\), the Hutchinson trace estimator \(H_M\) of a symmetric matrix \(A \in \mathbb{R}^{n \times n}\) satisfies

\[
\Pr (|H_M - \text{trace}(A)| \leq \epsilon |\text{trace}(A)|) \geq 1 - \delta.
\]

Unfortunately, our current convergence bounds on Hutchinson’s estimator introduce a new factor of \(O(\log n)\) to the number of samples required. The experiments reported in Section 6 show that in practice Hutchinson’s estimator is comparable to using a Gaussian estimator when the same number of samples is used (this may indicate that the bounds are weak), so we do not actually increase the number of samples.

**5. Reducing Randomness**

Our third variant, which is called TraceTriangleEM, is designed to reduce the amount of randomness used by the algorithm. Although reducing randomness has some theoretical interest, in our case there is also a practical interest. The previous variants of TraceTriangle required \(O(|V|)\) random bits per sample. This is too many bits to generate and store before the Monte-Carlo tests. Therefore,
in a parallel implementation all processors will need access to the pseudo-random number generator, incurring synchronization costs. TraceTriangleM uses only $O(\log |V|)$ random bits per sample, so all random bits can be generated in advance. A preprocessing step required an additional $O(|V|)$ random bits, so the overall cost is $O(|V| + M \cdot \log |V|)$. This is obviously less than $O(M \cdot |V|)$ required by the previous variants of TraceTriangle. The reduction in randomness does not come without a price, at least from a theoretical standpoint: an additional factor of $O(\log^2 |V|)$ samples is required. The experiments reported in Section 6 show that the estimator used by TraceTriangleM is comparable to using a Gaussian estimator when the same number of samples is used (this may indicate that the bounds are weak), so we do not actually increase the number of samples.

Our algorithm and its analysis are based on the following two ideas. The first idea is to restrict the sample vectors to a small sample space ($O(|V|)$ instead of $O(2^{|V|})$). More specifically we use unit vectors $e_i$ as sample vectors. By reducing the size of the sample space to $O(|V|)$ we are able to form a sample with only $O(\log |V|)$ bits. The quadratic forms $e_j^T A e_j$ are simply the diagonal elements $A_{jj}$, so the expectation of $e_j^T A e_j$ is clearly $\text{trace}(A)/n$. In contrast to previous variants, these quadratic forms do not depend in any way on the off-diagonal elements of $A$, only on the diagonal elements. The next lemma formalizes these observations. We omit the proof.

**Lemma 6.** Let $A$ be an $n \times n$ symmetric matrix. Let $j$ be a uniform random integer between 1 and $n$, and let $z = e_j$ be the $j$'th unit vector. $n z^T A z = \text{trace}(A)$ is an unbiased estimator of $\text{trace}(A)$ i.e., $E(n z^T A z) = \text{trace}(A)$ and $\text{Var}(n z^T A z) = n \sum_{i=1}^n A_{ii}^2 - \text{trace}^2(A)$.

The second idea is to not compute the trace of $A^3$, but the trace of $F A^3 F^T$ where $F$ is a unitary matrix. It is based on a simple consequence of Lemma 1: we are free to manipulate the adjacency matrix using unitary transformations since they preserve eigenvalues. More specifically for any unitary matrix $F$ we have $\Delta(G) = \frac{1}{\gamma} \text{trace}(F A^3 F^T)$.

We construct $F$ using a randomized algorithm that guarantees with high probability a relatively flat distribution of the diagonal elements of $F A^3 F^T$. More precisely, we construct $F$ in a way that attempts to flatten the distribution of all the elements of $F A^3 F^T$, not just its diagonal elements. Our construction of $F$ is the same as the one used in [2] to flatten entries of vectors.

**Definition 4.** A random mixing matrix is $F = FD$, where $F$ and $D$ are $n$-by-$n$ unitary matrices, and where $D$ is diagonal matrix with diagonal entries that are i.i.d Rademacher random variables ($\Pr(D_{ii} = \pm 1) = 1/2$).

The mixing matrix $F$ is the product of two unitary matrices, a seed matrix $F$ and a matrix $D$ with $\pm 1$ on the diagonal, so $F$ is unitary. For $F$ to be a good mixing matrix the value of $\eta = \max |F_{ij}|^2$ must be small. The following lemma shows this. We omit the proof, and refer the reader to [5] for analysis of the positive definite case.

**Definition 5.** A random mixed trace estimator for a symmetric matrix $A \in \mathbb{R}^{n \times n}$ is

$$D_M = \frac{n}{M} \sum_{i=1}^M z_i^T F A F^T z_i,$$

where the $z_i$’s are $M$ independent uniform random samples from $\{e_1, \ldots, e_n\}$, and $F$ is a random mixing matrix with $\max |F_{ij}|^2 \leq 2/n$.

**Remark 1.** Notice that the matrix $F$ is random, but is drawn only once. So in computing a single sample of $D_M$ it the random mixing matrix $F$ incurs a single cost of $n$ random bits.

**Lemma 7.** Let $\delta > 0$ be a failure probability and let $\epsilon > 0$ be a relative error. For $M \geq 8 \epsilon^{-2} \rho(A)^2 \ln(4/\delta) \ln^2(4n^2/\delta)$, the random mixed trace estimator $D_M$ of a symmetric matrix $A \in \mathbb{R}^{n \times n}$ satisfies

$$\Pr(|D_M - \text{trace}(A)| \leq \epsilon |\text{trace}(A)|) \geq 1 - \delta.$$

The are a few possibilities for $F$. The minimal value of $\eta$ for a unitary $F$ is $1/n$. A normalized DFT matrix achieves the minimum, but applying it requires complex arithmetic. A normalized Hadamard matrix also achieves the minimum and its entries are real. However, Hadamard matrices do not exist for all dimensions, so they are more difficult to use (they require padding). The Discrete Cosine Transform (DCT) and the Discrete Hartley Transform (DHT), which are real, exist for any dimension, and can be applied quickly, but their $\eta$ value is $2/n$, twice as large as that of the DFT and the Hadamard. All are valid choices. The decision should be based on the implementation cost of computing columns of $F$ and applying $D A D^T$ to them versus the value of $\eta$. In TraceTriangleM we use the DCT matrix. The pseudocode for the algorithm is given in Algorithm 2. Line 6 is an explicit formula for the rows of the DCT matrix.

**6. EXPERIMENTS**

We experimented with all three variants of TraceTriangle. This section reports the results of these experiments.

**6.1 Experimental Setup**

We compared a sequential implementation of TraceTriangle, to EigenTriangle [23], the current state-of-the-art
The results in Table 4 can be used to get some rule-of-thumb values for $\gamma$ based on the graph type. For example, collaborations/citations graph tend to work with low values of $\gamma$ (one or two), while web graphs, like the Wikipedia, require high values of $\gamma$. Arguably, the parameter that changes the most between different graphs is the value of $\rho(G)$. Different graph types have different typical values for $\rho(G)$. For collaboration/citations graphs $\gamma = 1 - 2$ seems adequate, for social networks $\gamma = 3$ and for large web graphs/communications networks $\gamma = 4$.

The second set of experiments we compares the running time of $\text{TraceTriangle}_R$ to $\text{EigenTriangle}$. Both algorithms are deterministic in their running time, so only one run was necessary. Figure 2 shows the speedup (or slowdown) of $\text{TraceTriangle}_R$ with respect to $\text{EigenTriangle}$. We see that for smaller graphs $\text{TraceTriangle}_R$ is faster than $\text{EigenTriangle}$, sometimes with a large factor. For the Wikipedia graphs, which were the largest graphs in our test set, $\text{EigenTriangle}$ was three times faster than $\text{TraceTriangle}_R$. On the other hand, on the Wikipedia matrices $\text{EigenTriangle}$ was not accurate. Furthermore, the ability to parallelize $\text{EigenTriangle}$ is very limited, while $\text{TraceTriangle}_R$ parallelizes well. We expect $\text{TraceTriangle}_R$ to scale much better with respect to the number of compute cores, therefore eliminating any running time advantage of $\text{EigenTriangle}$.

Fixing the number of eigenvalues computed implies that the number of matrix-vector multiplication done by $\text{EigenTriangle}$ grows very slowly. This explains why when the graph get larger $\text{EigenTriangle}$ is faster than $\text{TraceTriangle}$. On the other hand, as the graph get larger one can argue that more eigenvalue should be computed. In Figure 1 compare the accuracy-runtime tradeoff of $\text{TraceTriangle}_R$ and $\text{EigenTriangle}$. We plot the accuracy of both algorithms for different running times. $\text{EigenTriangle}$ is deterministic and its data points are based on a single runs with different #eigenvalues values. $\text{TraceTriangle}$ is randomized, so for different values of $\gamma$ we ran the algorithm 100 times and computed the mean absolute error. We see that $\text{TraceTriangle}$ error is monotonically decreasing. $\text{EigenTriangle}$’s error is not always monotonic, and is always higher than $\text{TraceTriangle}$’s error. We note that currently there is no guideline for the number of eigenvalue in $\text{EigenTriangle}$ (and this limits our ability to compare running time).

### 6.3 DOULION sparsification

**DOULION** [25] is an algorithm designed to accelerate triangle counting algorithms by randomly keeping only a small fraction of the edges. **DOULION**’s authors suggest to use **DOULION** with any triangle counting algorithm, so we tested **DOULION** with **TraceTriangle**. The potential benefits for **TraceTriangle** are clear. If an edge is kept with probability $p$ the expected number of edges after the sparsification is $p|E|$. Therefore, **TraceTriangle** is expected to speedup by roughly $1/p$.

The main problem in using **DOULION** is that it changes the triangle sparsity. Every approximate counting algorithm must depend in some way on the triangle sparsity. In **TraceTriangle** the accuracy depends on $\rho(G)$ whose bound is expected to grow after sparsification. Other algorithms suffer from the same problem. Schank et al. method [21], for
Table 2: Graphs used for the experiments

<table>
<thead>
<tr>
<th>Graph</th>
<th>Vertices</th>
<th>Edges</th>
<th>Source</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arxiv-HEP-th</td>
<td>27,240</td>
<td>341,923</td>
<td>UF Sparse Matrix Collection², matrix 1502</td>
<td>Arxiv High Energy Physics paper citation network</td>
</tr>
<tr>
<td>CA-AstroPh</td>
<td>18,772</td>
<td>198,050</td>
<td>SNAP Network Data Repository³, 'ca-AstroPh' matrix</td>
<td>Arxiv Astro Physics collaboration network</td>
</tr>
<tr>
<td>CA-GrQc</td>
<td>5,242</td>
<td>14,484</td>
<td>SNAP Network Data Repository, 'ca-GrQc' matrix</td>
<td>Arxiv GR-QC General Relativity and Quantum Cosmology collaboration network</td>
</tr>
<tr>
<td>Epinions</td>
<td>75,879</td>
<td>405,740</td>
<td>SNAP Network Data Repository, 'soc-Epinions1' matrix</td>
<td>Who-trusts-whom network of Epinions.com</td>
</tr>
<tr>
<td>Slashdot0811</td>
<td>77,360</td>
<td>469,180</td>
<td>SNAP Network Data Repository, 'soc-Slashdot0811' matrix</td>
<td>Slashdot social network from November 2008</td>
</tr>
<tr>
<td>wiki-vote</td>
<td>7,115</td>
<td>100,762</td>
<td>SNAP Network Data Repository, 'wiki-vote' matrix</td>
<td>Wikipedia who-votes-on-whom network</td>
</tr>
<tr>
<td>email-EuAll</td>
<td>265,214</td>
<td>364,481</td>
<td>SNAP Network Data Repository, 'email-EuAll' matrix</td>
<td>Email network from a EU research institution</td>
</tr>
<tr>
<td>AS Oregon</td>
<td>13,579</td>
<td>37,448</td>
<td>Courtesy of Charalampos Tsourakakis</td>
<td>AS Oregon</td>
</tr>
<tr>
<td>Web-Stanford</td>
<td>281,903</td>
<td>1,992,636</td>
<td>SNAP Network Data Repository, 'web-Stanford' matrix</td>
<td>Web graph of stanford.edu</td>
</tr>
<tr>
<td>WikiNov2005</td>
<td>1,634,989</td>
<td>18,549,197</td>
<td>UF Sparse Matrix Collection, matrix 1843</td>
<td>Wikipedia graph, Nov 11, 2005</td>
</tr>
</tbody>
</table>

Table 3: Comparison between different variants of TraceTriangle, based on 100 runs. The “% err < 5%” and “% err < 10%” show the percent of cases for which the error was smaller than 5% and 10%.

<table>
<thead>
<tr>
<th>Graph</th>
<th>TRACE_TRIANGLE</th>
<th>TRACE_TRIANGLE</th>
<th>TRACE_TRIANGLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph</td>
<td>% err &lt; 5%</td>
<td>% err &lt; 10%</td>
<td>% err &lt; 5%</td>
</tr>
<tr>
<td>Arxiv-HEP-th</td>
<td>1</td>
<td>85%</td>
<td>100%</td>
</tr>
<tr>
<td>CA-AstroPh</td>
<td>1</td>
<td>98%</td>
<td>100%</td>
</tr>
<tr>
<td>CA-GrQc</td>
<td>2</td>
<td>70%</td>
<td>91%</td>
</tr>
<tr>
<td>Epinions</td>
<td>3</td>
<td>63%</td>
<td>95%</td>
</tr>
<tr>
<td>Slashdot0811</td>
<td>3</td>
<td>61%</td>
<td>91%</td>
</tr>
<tr>
<td>wiki-vote</td>
<td>3</td>
<td>51%</td>
<td>86%</td>
</tr>
<tr>
<td>email-EuAll</td>
<td>4</td>
<td>52%</td>
<td>81%</td>
</tr>
<tr>
<td>AS Oregon</td>
<td>4</td>
<td>45%</td>
<td>79%</td>
</tr>
<tr>
<td>Web-Stanford</td>
<td>3</td>
<td>36%</td>
<td>68%</td>
</tr>
<tr>
<td>WikiNov2005</td>
<td>4</td>
<td>34%</td>
<td>62%</td>
</tr>
<tr>
<td>WikiSep2006</td>
<td>4</td>
<td>58%</td>
<td>87%</td>
</tr>
<tr>
<td>WikiNov2006</td>
<td>4</td>
<td>65%</td>
<td>91%</td>
</tr>
<tr>
<td>WikiFeb2007</td>
<td>4</td>
<td>68%</td>
<td>97%</td>
</tr>
</tbody>
</table>
Table 4: Accuracy on various data sets, based on 100 runs of TraceTriangleR and a two (deterministic) runs
of EigenTriangle: with 20 eigenvalues, and with running time equal to TraceTriangle. The “% err < 5%” and
“% err < 10%” show the percent of cases for which the error was smaller than 5% and 10%.

<table>
<thead>
<tr>
<th>Graph</th>
<th>γ</th>
<th>mean abs error</th>
<th>max abs error</th>
<th>% err &lt; 5%</th>
<th>% err &lt; 10%</th>
<th>EigenTriangle 20 eigenvalue</th>
<th>EigenTriangle γ time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arxiv-HEP-th</td>
<td>1</td>
<td>2.7%</td>
<td>6.7%</td>
<td>85%</td>
<td>100%</td>
<td>41%</td>
<td>75%</td>
</tr>
<tr>
<td>CA-AstroPh</td>
<td>1</td>
<td>1.6%</td>
<td>6.5%</td>
<td>98%</td>
<td>100%</td>
<td>47%</td>
<td>84%</td>
</tr>
<tr>
<td>CA-GrQc</td>
<td>2</td>
<td>3.2%</td>
<td>11.5%</td>
<td>77%</td>
<td>98%</td>
<td>12%</td>
<td>48%</td>
</tr>
<tr>
<td>Epinions</td>
<td>3</td>
<td>4.6%</td>
<td>12.3%</td>
<td>60%</td>
<td>94%</td>
<td>6.2%</td>
<td>9.6%</td>
</tr>
<tr>
<td>Slashdot0811</td>
<td>3</td>
<td>4.5%</td>
<td>16.0%</td>
<td>62%</td>
<td>95%</td>
<td>2.9%</td>
<td>3.1%</td>
</tr>
<tr>
<td>wiki-vote</td>
<td>3</td>
<td>5.6%</td>
<td>19.0%</td>
<td>52%</td>
<td>86%</td>
<td>1.9%</td>
<td>0.9%</td>
</tr>
<tr>
<td>email-EuAll</td>
<td>4</td>
<td>5.6%</td>
<td>22.4%</td>
<td>54%</td>
<td>85%</td>
<td>17%</td>
<td>16%</td>
</tr>
<tr>
<td>AS Oregon</td>
<td>4</td>
<td>6.5%</td>
<td>16.3%</td>
<td>39%</td>
<td>81%</td>
<td>1.5%</td>
<td>9.7%</td>
</tr>
<tr>
<td>Web-Stanford</td>
<td>3</td>
<td>7.9%</td>
<td>24.9%</td>
<td>40%</td>
<td>70%</td>
<td>28%</td>
<td>26%</td>
</tr>
<tr>
<td>WikiNov2005</td>
<td>4</td>
<td>8.7%</td>
<td>28.3%</td>
<td>34%</td>
<td>67%</td>
<td>49%</td>
<td>42%</td>
</tr>
<tr>
<td>WikiSep2006</td>
<td>4</td>
<td>4.1%</td>
<td>14.0%</td>
<td>69%</td>
<td>91%</td>
<td>54%</td>
<td>44%</td>
</tr>
<tr>
<td>WikiNov2006</td>
<td>4</td>
<td>4.4%</td>
<td>13.9%</td>
<td>58%</td>
<td>95%</td>
<td>54%</td>
<td>44%</td>
</tr>
<tr>
<td>WikiFeb2007</td>
<td>4</td>
<td>4.0%</td>
<td>11.3%</td>
<td>69%</td>
<td>99%</td>
<td>55%</td>
<td>48%</td>
</tr>
</tbody>
</table>

Figure 1: Closer inspection of the accuracy of TraceTriangle vs. EigenTriangle on WikiSep2006. We
varied EigenTriangle’s #eigenvalues parameter and TraceTriangle’s γ parameter.

Figure 2: Speedup of TraceTriangleR with respect to EigenTriangle.

Table 5: Results involving TraceTriangle with doullion sparsification. Error is the mean of 10 runs
for 10 different sparsifications.

<table>
<thead>
<tr>
<th>Graph</th>
<th>p</th>
<th>mean error</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA-AstroPh</td>
<td>0.1</td>
<td>6.2%</td>
<td>3.5</td>
</tr>
<tr>
<td>Slashdot0811</td>
<td>0.2</td>
<td>12.5%</td>
<td>2.3</td>
</tr>
<tr>
<td>WikiNov2006</td>
<td>0.2</td>
<td>16.7%</td>
<td>4.2</td>
</tr>
</tbody>
</table>

example, requires \(\Theta(T_0 + T_1 + T_2)\) samples. After the sparsification
the numerator grows and the denominator shrinks.

Table 5 shows the results for three matrices. In one case,
triangle density was high enough after sparsification to still
yield reasonable results. In other graphs triangle sparsity
was too high and results are of low quality. Speedup is
achieved at the cost of accuracy. We conclude that in order
to effectively use graph sparsification in approximate trian-
gle counting the sparsification algorithm must somehow keep
the triangle sparsity close to the original. This is a feature
that DOULION lacks. We currently are not aware of any such
algorithm and leave this issue for future research.

7. CONCLUSIONS

We have presented an embarrassingly parallel algorithm
for approximating the number of triangles in a graph. We
showed that given sufficient triangle density our algorithm
gives a good estimate to the number of triangles with high
probability. We also showed that for real-life graphs encoun-
tered in graph-mining applications our algorithm approxi-
mates the number of triangles well. An interesting feature
of our algorithm is the use of a randomized numerical linear
algebra algorithm to solve a combinatorial problem. Our
main contributions are:

- The new triangle counting algorithm and its theoretical
  analysis.
- Experiments on large graphs with several millions of
  vertices and edges.
Acknowledgments

It is a pleasure to thank Mark Tygert and Sivan Toledo for helpful comments and ideas, and Charalampos Tsourakakis for helpful discussions and for supplying his triangle counting code.

8. REFERENCES


