Triangle counting in large networks: a review

Article *in* Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery · October 2017

DOI: 10.1002/widm.1226

CITATIONS
43

READS
1,747

2 authors:

Mohammad Hasan
Indiana University-Purdue University Indianapolis
141 PUBLICATIONS **2,978** CITATIONS
SEE PROFILE

Vachik S. Dave
WalmartLabs
21 PUBLICATIONS **272** CITATIONS
SEE PROFILE

Advanced Review

Triangle counting in large networks: a review

Mohammad Al Hasan* and Vachik S. Dave

Counting and enumeration of local topological structures, such as triangles, is an important task for analyzing large real-life networks. For instance, triangle count in a network is used to compute transitivity—an important property for understanding graph evolution over time. Triangles are also used for various other tasks completed for real-life networks, including community discovery, link prediction, and spam filtering. The task of triangle counting, though simple, has gained wide attention in recent years from the data mining community. This is due to the fact that most of the existing algorithms for counting triangles do not scale well to very large networks with millions (or even billions) of vertices. To circumvent this limitation, researchers proposed triangle counting methods that approximate the count or run on distributed clusters. In this paper, we discuss the existing methods of triangle counting, ranging from sequential to parallel, single-machine to distributed, exact to approximate, and off-line to streaming. We also present experimental results of performance comparison among a set of approximate triangle counting methods built under a unified implementation framework. Finally, we conclude with a discussion of future works in this direction.

How to cite this article:
WIREs Data Mining Knowl Discov 2018, 8:e1226. doi: 10.1002/widm.1226

INTRODUCTION

Network data appear in many domains, including social, communication, and information sciences. Although the networks in these domains differ in terms of their structural composition, some topological structures, specifically, triangles, appear in abundance across networks in all different domains. Abundance of triangles in real-life networks motivated scientists to invent metrics, such as clustering coefficient[1] or transitivity ratio[2] to characterize and analyze networks. The existence of triangles in social networks has also been studied and explained from various social science theories such as homophily,[3] and transitivity. A key computational task for all these studies is to count the number of triangles in a network, which is the focus of this work.

There are many real-life applications of triangle counting. The most well-known among them, of course, is to compute the transitivity ratio (or, simply transitivity) of a network, which is defined as the ratio between the counts of triangles and triples (a path of length two) in a network. Given that the number of triples can be computed simply from the degree of the vertices of a network, transitivity computation then it becomes identical to the task of triangle counting. Clustering coefficient is another similar metric, but its value is defined for a given vertex of a network—for a vertex u, its clustering coefficient is the fraction of u's neighbors who are neighbor themselves. Both clustering coefficient and transitivity have been used as a key metric for network analysis and network evolution models.[4]

Triangle count has also been used for several other nonobvious applications. Becchetti et al.[5] have used distribution of local triangles for detecting web spam. Specifically, they have shown that the distribution of local triangle frequency of spam hosts is significantly different from those of the nonspam hosts. The distribution of triangles is also used to uncover

*Correspondence to: alhasan@iupui.edu

Department of Computer Science, IUPUI, Indianapolis, IN, USA

Conflict of interest: The authors have declared no conflicts of interest for this article.

hidden thematic structure in the World Wide Web. Eckmann and Moses have shown that connected regions of web graph, which are dense in triangles represents a common topic.[6] Bar-Yossef et al.[7] have used triangle count for query plan optimization in databases. Overlapping triangles (or more generally k-cliques) have been used for community discovery.[8]

Triangle counting, though appears to be a simple task algorithmically, has attracted many contributions over the years from scientists in diverse domains, including data mining and graph theory. While earlier works[9,10] mainly care for asymptotic computational complexity, in recent works, real-life execution time has been a major consideration, motivation for which comes from the enormous size of real-life networks having vertices in the ranges of millions to billions. To achieve efficiency, approximate triangle counting through sampling has been a very active direction in many recent works.[11–16] Also, researchers have tried to achieve efficiency through algorithms that run on multi-core or distributed environment.[13,14,17] Some variants of triangle counting algorithms have also been inspired by data access constraints. For example, triangle counting algorithms have been proposed for various data access scenarios which are different from traditional random memory access, examples include restricted access,[12] and streaming data access.[16,18]

Computational complexity of a triangle counting algorithm is a good indicator of its efficiency, but in real-life the execution time of two algorithms can be widely different even if they have the same computational complexity. The main reason for this fact is the hidden constant of the computation complexity, which depends on various properties of the input graph. Sparsity is one of such properties. Large real-life networks are very sparse, in which the number of edges is typically a constant factor of the number of vertices; in other words, the average degree of a vertex is constant. Another important property is that the degree distribution of real-life networks is skewed. Although, the average degree of a network is constant, there always exist a few vertices that have a very large degree. This phenomenon is commonly known as power-law degree distribution,[19] which significantly affects the performance of a triangle counting algorithm.

In this paper, we provide a thorough review of triangle counting algorithms. We group the existing methods based on their computation model or data access patterns. Then, we discuss the algorithms by comparing and contrasting their time complexity. Finally, we show some experimental results that compare the performance of some of these algorithms.

The following section provides definitions of various concepts which are related to the task of counting triangles. For reader's convenience, in Table 1, we also provide notations used throughout the paper.

BACKGROUND

$G(V, E)$ is a graph where V is the set of vertices and E is the set of edges. We use n and m for representing the number of vertices ($|V|$) and the number of edges ($|E|$). Each vertex in the graph can be uniquely identified by a number between 1 and n. The assignment of identifier can be arbitrary, but it is fixed. We also consider that G is simple, connected, and undirected. Because G is simple, between a pair of vertices u and v, there exists at most one edge, which we define by (u, v) where $u < v$. For a vertex u, we use $d(u)$ to denote the degree value of u, $adj(u)$ to denote the set of u's neighboring vertices, and inc(u) to denote the edges that are incident to u. Likewise, for an edge e, we use inc(e) to denote the incidence vertices of the edge e. It is easy to see that $\sum_{u \in V} d(u) = 2m$. The maximum degree value over the vertices is defined as $d_{\max}$.

Triples and Triangles

A (connected) triple (u, v, w) at a vertex v is a path of length two for which v is the center vertex. If the other two vertices (u and w) are also connected by an edge, the triple is called a closed triple (triangle), otherwise it is called an open triple. A triangle actually contains three closed triples, one centered on each of its vertices.

We use the symbol Π_v to represent the set of triples that are centered at the vertex v. The set of triples in a graph $G = (V, E)$ is Π, which is the union of the set of triples at each of its node, i.e., $\Pi = \cup_{v \in V} \Pi_v$. If the degree of each of the vertices is known, the total number of triples can be computed efficiently as below:

$$|\Pi| = \sum_{v \in V} |\Pi_v| = \sum_{v \in V} \binom{d(v)}{2}. \quad (1)$$

Based on whether the triple is open or closed (in terms of its induced embedding in the graph G), we partition the set Π into $\Pi^{\angle}$ (open triples) and $\Pi^{\triangledown}$ (closed triples). Note that, each of the nodes of a triangle in a graph G contributes one distinct triple in the set $\Pi^{\triangledown}$. We use Λ to represent the set of distinct triangles in a graph G. Clearly, the size of $\Pi^{\triangledown}$ is three times the size of Λ, as the former contains three

copies of a distinct triangle each centered at one of the triangle vertices. Mathematically, $|\Lambda| = \frac{1}{3}|\Pi^{\Delta}|$.

To represent the set of open and closed triples centered at a vertex v, we use $\Pi_v^{\angle}$ and Π_v^{Δ}, respectively. If $t(G)$ is the number of triangles in the graph G, then

$$t(G) = |\Lambda| = \frac{1}{3}|\Pi^{\Delta}| = \frac{1}{3}\sum_{v \in V}|\Pi_v^{\Delta}|. \quad (2)$$

Counting, Enumeration, and Sampling of Triangles

For a given graph G, triangle counting is the task of obtaining the number $t(G)$ as defined in Eq. (2). On the other hand, triangle enumeration task is to enumerate the members of Λ, i.e., to list all unique triangles in a given graph. Enumeration is a costlier task than counting because the former solves the latter immediately, but the latter does not necessarily solves the former. Nevertheless, for many real-life applications, one may need to enumerate the triangles rather than simply finding a total count of it, so both counting and enumeration tasks stand on their own merit. Finally, sampling of triangles is to obtain a subset of Λ, typically the size of the subset is a user defined parameter. Depending on the sampling algorithm, the triangles in the sample set can be chosen uniformly (each triangle is sampled with uniform probability) or they may be sampled with a biased probability. Sometimes we are only interested to find a count of triangles that are incident to a given vertex. This task is then known as *local triangle counting*. Local triangle count is important to find *clustering coefficient* of a given vertex.

TABLE 1 | Summary of the Notations

Notations	Meaning
n	Number of vertices
m	Number of edges
$d(u)$	Degree of vertex u
$adj(u)$	Set of neighboring vertices of the vertex u
$\text{inc}(u)$	Set of edge incident to vertex u
Π	Set of all triples
Π_v	Set of triples centered at vertex v
$\Pi^{\angle}$	Set of all open triples
$\Pi^{\triangledown}$	Set of all closed triples
Λ	Set of distinct triangles
$t(G)$	Number of triangles in the graph G
$\gamma(G)$	Transitivity of the graph G
$N(u)$	Sorted neighbors of vertex u
$\mathbf{A}$	Adjacency matrix
$d_{\max}$	Max degree of a vertex in the graph

(Local) Clustering Coefficient

Clustering coefficient is a metric denoting the clustering tendency of the vertices in a graph. When the metric is defined on a vertex of a graph it is called local clustering coefficient. For a given vertex, u, its local clustering coefficient $C(u)$ is the fraction of u's neighbors who are neighbor themselves. Mathematically,

$$C(u) = \frac{|(v,w) : (v,w) \in E \wedge v,w \in adj(u)|}{adj(u)(adj(u)-1)/2}. \quad (3)$$

The average of local clustering coefficient over the vertices is called clustering coefficient of the network.

Transitivity

Newman et al.[20] defined the transitivity of a graph G as the fraction that represents the number of closed triples divided by the number of all the triples over the entire network. We use $\gamma(G)$ to denote transitivity of G

$$\gamma(G) = \frac{|\Pi^{\Delta}|}{|\Pi|} = \frac{|\Pi^{\Delta}|}{|\Pi^{\angle}| + |\Pi^{\Delta}|}. \quad (4)$$

Using Eqs. (2) and (4), the triangle count ($t(G)$) of a network can be obtained from the transitivity of the network as below:

$$t(G) = \frac{1}{3}\cdot\gamma(G)\cdot|\Pi|. \quad (5)$$

Metropolis–Hastings (MH) Algorithm

Several approximate triangle counting methods sample triangles or triples using random walk-based indirect sampling strategies, also known as Markov Chain Monte Carlo (MCMC) sampling. Metropolis–Hastings (MH) algorithm is a variant of MCMC algorithm; its goal is to draw samples from some distribution $\pi(x)$, called the *target distribution*, where, $\pi(x) = f(x)/K$; here, $f(x)$ is any function which assigns a nonnegative real-value to a population object x denoting its desirability in regards to sampling. K is a normalization constant to make the sum of $\pi(x)$ over the population object equal to 1. Typically, K is not known or difficult to compute.

MH algorithm is used together with a random walk to perform MCMC sampling. For this, the MH

algorithm draws a sequence of samples from the target distribution as follows:

1. It picks an initial state (say, x) satisfying $f(x) > 0$.
2. From current state x, it samples a state y using a distribution $q(x, y)$, referred as *proposal distribution*.
3. Then, it calculates the *acceptance probability* $\alpha(x, y)$ (Eq. (6)) and accepts the proposal move to y with probability $\alpha(x, y)$. The process continues until the Markov chain reaches to a stationary distribution.

$$\alpha(x,y) = \min\left(\frac{\pi(y)q(y,x)}{\pi(x)q(x,y)},1\right) = \min\left(\frac{f(y)q(y,x)}{f(x)q(x,y)},1\right). \quad (6)$$

Importance Sampling

Importance sampling (IS) is a sampling strategy, which is used to estimate expectation of a function $f(x)$ relative to some distribution $p(x) = \widetilde{p}(x)/K$, called the target distribution, whereas the samples are actually obtained from a different distribution $q(x)$, called the proposal distribution. IS is useful when it is easier to sample from the distribution q but we need to obtain expectation with respect to a different distribution p. For instance, for triangle counting, we want to obtain triple samples from a uniform distribution, i.e., the target distribution p is uniform, but it may be easier to sample triples from a biased distribution, say q. Using the idea of IS, the expectation of $f(x)$ with respect to the target distribution is equal to

$$\mathbb{E}_p[f(x)] = \sum_{i=1}^{S} f(x_i)\cdot w(x_i), \quad (7)$$

where,

$$w(x_i) = \frac{\widetilde{p}(x_i)/q(x_i)}{\sum_{j=1}^{S}\widetilde{p}(x_j)/q(x_j).} \quad (8)$$

ORGANIZATION OF THE REVIEW

We organize this review based on classification of the triangle counting methods as depicted in Figure 1. Our first level classification of triangle counting methods is based on data (graph) access pattern. We consider two kinds of data access patterns: random access and restricted access.

Random access methods assume that the entire network is available in the memory in an adjacency vector data structure (or in other format) and we also know the size of the network—the number of vertices (n) and the number of edges (m). These random access methods are further divided into three sub-categories: (1) exact triangle counting; (2) approximate triangle

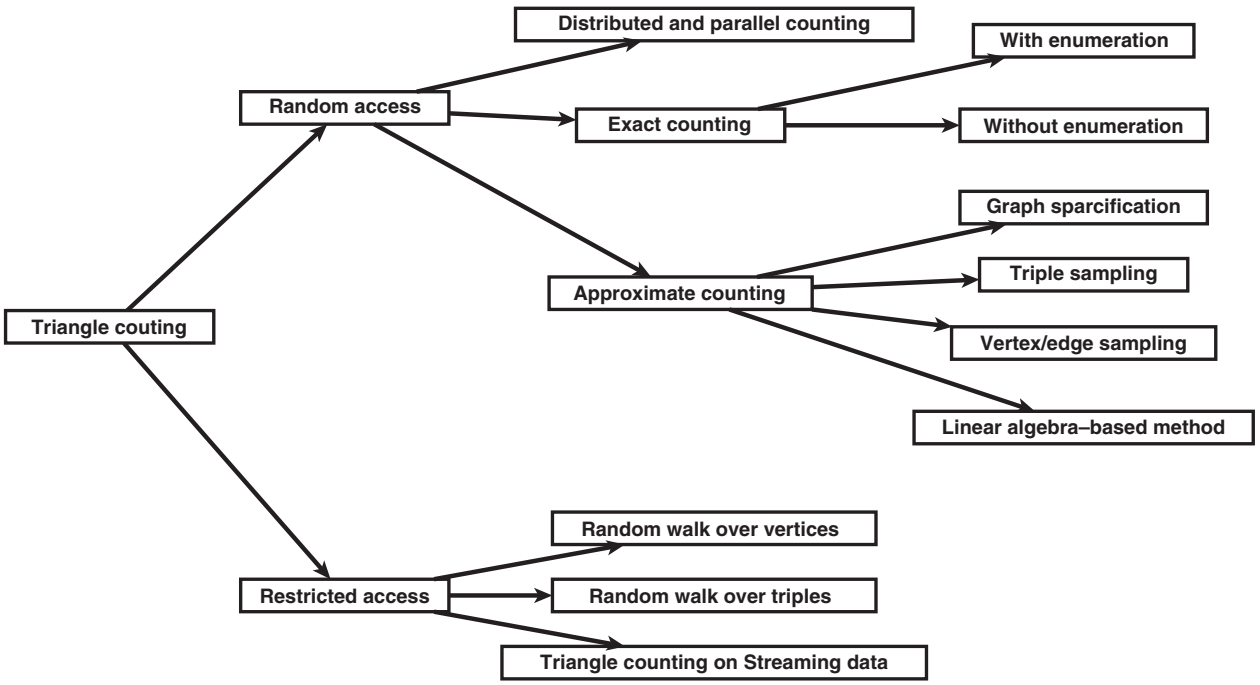

FIGURE 1 | Classification of triangle counting works.

counting; (3) distributed and parallel triangle counting. Methods in the first sub-category, i.e., the exact triangle counting methods provide actual count of triangles in the given input network, which can be obtained with or without enumeration of each triangle. Methods in the second sub-category are approximate methods, which calculate triangle counts with acceptable error in the count. Nevertheless, approximation methods are much faster compared to exact counting methods. Most of the approximation counting methods are based on different sampling approaches. Methods in the final sub-category run on distributed and parallel platforms. For triangle counting, such methods have recently become popular, as they can provide exact or approximate triangle counts for huge networks, which cannot be stored into the main memory of a single machine.

In real world, there are many networks, which are not fully accessible so random access based methods for triangle counting is not an option for such networks. Such networks can only be crawled, i.e., an analyst can only explore the neighbors of the currently visiting node. For restricted access, it is assumed that access to one seed vertex or a collection of seed vertices of the network is available so that the crawling can be initiated. Another assumption is that the network is connected or the largest (giant) connected component of the network covers the majority of the vertices and the part of the network excluding the giant component can be ignored. Because, the network is connected, one can attempt to crawl the entire network by using graph traversal methodologies and save the network (in memory or disk) for counting triangles by using random access-based methodologies. However, we assume that the network is very large (say, Internet network) and it does not fit in the main memory. So, random access-based methodologies do not work on such networks, or to the least, such methodologies are highly inefficient due to frequent I/O access. In such restricted access scenarios, one cannot obtain an exact count of triangles but random walk over the network provides a viable option for approximate triangle counting.

Another type of restricted access is streaming data access, where the graph data appears as a stream of edges. Limited memory does not allow all the edges to be stored in the memory so a triangle counting method requires to store a judiciously selected sample of edges or some form of summary statistics computed over the edges. Edges that appear in the stream are lost if they are not saved. Because, streaming data access works with a sample of edges it only provides an approximate count of triangles in a graph.

In the following section, we discuss exact triangle counting algorithms, which is followed by discussion of approximate triangle counting algorithms. Then, we discuss triangle counting algorithms that work for restricted access and streaming access scenarios. After that we discuss some triangle counting methods which work on distributed or parallel platforms. There after, we present experimental results from the comparison among a collection of approximate triangle counting methodologies. Lastly, we discuss other two related counting tasks before concluding the paper.

EXACT TRIANGLE COUNTING WITH RANDOM ACCESS

We first discuss triangle counting algorithms with random memory access assumption. Under this assumption, we can obtain the adjacency vector of any vertex in $O(1)$ time. We also assume that, in the adjacency vector of a vertex u, the neighbors of u, $adj(u)$ is sorted. So, the existence of an edge (u, v) can be answered in $O(\lg n)$ time using binary search on that vector. Another option is to use a hash-table of edges to answer the edge existence query in expected $O(1)$ time. Note that even if we use binary search for answering edge existence query, the complexity $O(\lg n)$ is only a worst-case time complexity, which applies to a very high degree vertex. Given the fact that the average degree of real-life networks is constant, and for triangle counting we need to ask the edge existence query over a very large number of small adjacency lists and occasionally a few large adjacency lists, we can amortize the cost of costly binary searches over a large number of cheap searches and assume that the cost of edge existence query is constant.

A brute-force triangle counting algorithm can be designed by enumerating all distinct three vertex sets $\{u, v, w\}$ (not necessarily connected) in a network and then testing whether the three vertices form a triangle. Because the number of such three-vertex sets is in the order of $\Theta(n^3)$, the brute-force complexity of triangle counting is $\Theta(n^3)$. Note that, such an algorithm not only counts the triangles but also iterates (or lists) the triangles. Note that, any algorithm that iterates each of the triangles has a worst-case complexity of $\Theta(n^3)$, because the maximum possible number of triangles in a graph of n vertices is exactly $\frac{n}{3}$, which is realized when the given graph is a clique.

Over the years, many triangle counting methods have been proposed which have better runtime performance. Specifically, the methods that count but do

not list all the triangles have worst-case time complexity much better than $\Theta(n^3)$. Using this observation, we will discuss the algorithms positioned in two groups. The first group of algorithms only provides a count of triangles without listing (or enumerating) them. On the other hand, the second group of algorithms lists the triangles. Our discussion of exact triangle counting algorithms is brief, however, Schank's PhD thesis[21] is an excellent reference of the methodologies for exact triangle counting algorithms.

Triangle Counting Without Enumeration

The earliest methods of triangle counting without enumeration are based on matrix multiplication of the adjacency matrix. It is easy to see that, if $\mathbf{A}$ is the adjacency matrix of an undirected network G, the diagonal elements of $\mathbf{A}^3[i, i]$ contain the total number of closed walks of length 3 that begin and end at vertex i. Given that a triangle is counted as a closed walk starting and ending at each of its three vertices and also for an undirected graph each closed walk can be counted twice (counterclockwise and clockwise); thus, the total number of triangles in a graph G, $t(G) = (1/6)\mathrm{Tr}(\mathbf{A}^3)$. The complexity of this algorithm is $\Theta(n^3)$, however a fast matrix multiplication algorithm can be used to achieve a better algorithm, which runs in $\Theta(n^\omega)$, where the current best value of ω, the exponent of matrix multiplication, is around 2.373.[22] However, the hidden constants of many of the fast matrix multiplication algorithms are large, which makes these algorithms not much superior (if not worse) than the traditional $\Theta(n^3)$ based matrix multiplication algorithm for counting triangles in real-life large graphs.

Alon et al.[10] has proposed a triangle counting algorithm (hereby called as AYZ), which runs in $Om^{2\omega/(\omega+1)}$ time. In this algorithm, authors first define $\Delta = m^{(\omega-1)/(\omega+1)}$ and name a vertex *high degree* if its degree is higher than Δ, otherwise it is a *low degree* vertex. There are at most $m \cdot \Delta$ paths for which the intermediate vertices are low degree, each of these paths can be checked for triangle in $O(m\Delta)$ time. Then, the remaining triangles are involved with all high degree vertices. As there are at most $(2m/\Delta)$ high degree vertices, triangles involving those vertices can be found in $O((m/\Delta)^\omega)$ time. Then, overall complexity of this method is $O(m\Delta + (m/\Delta)^\omega) = O(m^{2\omega/(\omega+1)})$. Because AYZ uses matrix multiplication as a part of the method, it also belongs to non-enumeration-based triangle counting method. Note that, if $\omega = 3$ (which is the case of traditional matrix multiplication), the complexity of AYZ method is $O(m^{3/2})$

Triangle Counting With Enumeration

Enumeration-based triangle counting algorithms list all the triangles, then counting becomes a trivial task. The obvious advantage of an enumeration-based method is that it returns the list of all the triangles, which can be used for downstream tasks such as community discovery[8] or spam filtering.[5] Besides the above, enumeration-based methods are preferred over the matrix multiplication-based methods even for solving the counting task, because matrix multiplication-based methods suffer from large memory footprint.

One of the earliest triangle enumeration method is proposed by Itai and Rodeh.[9] This method was actually proposed to find just one triangle, but it can easily be extended to list all the triangles. This algorithm first finds a spanning tree $T(V, E_T)$ of the graph $G(V, E)$ and then for each edge $(u, v) \in E_T$, it checks whether $(\mathrm{pred}(u), v) \in E$ (pred stands for predecessor of a node in the tree); if true, it emits $(u, v, \mathrm{pred}(u))$ as a triangle. It also checks whether $(\mathrm{pred}(v), u) \in E$. If true, it outputs $(u, v, \mathrm{pred}(v))$ as a triangle. The edges of T are then removed from G and the process is repeated by building a new spanning tree of the updated graph. The algorithm terminates after no more edges exist in G. Each iteration takes $O(m)$ time, and it can be shown that there are at most $O(\sqrt{m})$ iterations, so the complexity of this method is $O(m^{3/2})$. However, this algorithm needs modification of the graph data structure, which is costly, hence its real-life execution time is not so competitive with other methods.

Algorithm 1 Node Iterator Algorithm

Require: $G(V, E)$

```
count = 0
for each node v ∈ V do
    count_v = 0
    for each pair of distinct neighbor u and w in adj(v) do
        if (u, w) ∈ E then
            count_v = count_v + 1
        end if
        count = count + count_v
    end for
end for
return count/3
```

A better algorithm is to enumerate over vertex-pairs that are adjacent to a given vertex v. As shown in Algorithm 1, this method iterates over the vertices through the variable v. For each pair of vertices u,

and w from $adj(v)$, it checks whether an edge exists between u and w; if yes, $\{u, v, w\}$ forms a triangle, otherwise not. A cumulative sum of total number of triangles is then returned after dividing the sum by 3. Division is required because each triangle is counted thrice, once in the iteration in which one of its vertices is chosen in the outermost loop. Because the algorithm iterates over the vertices of a network, it is also known as NodeIterator algorithm. The amount of work done at each vertex is $\Theta(d(v)^2)$, so the complexity of the method is $\Theta\left(n \cdot d_{\max}^2\right)$. For networks, for which the degree distribution is highly skewed, say if the maximum degree value of a network grows linearly with the number of vertices the time complexity of this algorithm is $\Theta(n^3)$; this is true even for a star network for which the triangle count is equal to 0.

Algorithm 2 Edge Iterator Algorithm

Require: $G(V, E)$

1: $count = 0$
2: **for** each edge $(u, v) \in E$ **do**
3: $adj_1 = \{x | x \in adj(u),\ x > u\}$
4: $adj_2 = \{x | x \in adj(v),\ x > v\}$
5: $count_e = |intersection(adj_1, adj_2)|$
6: $count+ = count_e$
7: **end for**
8: **return** $count$

Instead of iterating over the vertices, triangles can also be counted by iterating over the edges (Algorithm 2). While iterating over the edges the algorithm counts the number of triangles in which each of the edges contributes. Such an algorithm is known as EdgeIterator method. The complexity of an EdgeIterator algorithm is $O(m \cdot d_{\max})$.

Both node iterator and edge iterator algorithms iterate over each of the potential two-length paths and check whether it forms a triangle. To avoid duplicate counting and reduce counting time, strategies can be adopted so that each triple is checked exactly once.[23,24]

For node iterator algorithm, we can simply sort the nodes based on their degree and enforce an ordering on nodes $v < u < w$ in Algorithm 1. This ensures that each triangle is counted only once by it's smallest degree vertex in the variable $count_v$. In that case, the division by 3 in Line 11 of Algorithm 1 is not needed. Also, the sort order improves the running time of the algorithm by not considering many two-length paths at all. For example, for a star graph, the terminal nodes are degree 1 nodes and the star node is the highest degree node. Using this sort order, none of the triples centered at the star node need to be tested for triangle and the method can return a 0 value for the triangle count of a star graph, without testing any of its triples. Similar optimization can also be pursued for edge iterator algorithm; for example, if the adjacency list of the vertices are sorted, when computing the intersection of sets adj_1 and adj_2 (line 5 of Algorithm 2), we can restrict the intersection operation such that the third node of a triangle, $x \in adj(u) \cap adj(v)$, satisfies $x > \max\{u, v\}$.

When the redundant counting is avoided and the cost of edge existence test is $O(1)$, the time complexity of triangle enumeration is bounded by the total number of triples in a graph, which is equal to $\sum_{v \in V} \binom{d(v)}{2}$. We can also bound the triple count in terms of edge count (m) with a careful analysis. Say, we divide the vertices into two groups: *high degree*, having degree $> \sqrt{m}$, and *low degree*, the remaining. For each low degree vertex, the maximum number of possible triples that are centered at these vertices can be obtained by packing all edges with as few low degree vertices as possible. Because the total number of edges is m, we can pack them in $\sqrt{m}$ vertices each having degree $\sqrt{m}$. Thus, the number of triples that are centered at a low degree vertex is at most $O\left(\sqrt{m} \cdot (\sqrt{m})^2\right) = O\left(m^{3/2}\right)$. On the other hand, the number of high degree vertices is at most $(2m/\sqrt{m})$, as each of these vertices has a degree at least $\sqrt{m}$; then the number of triples consisting of high degree vertices is at most $O\left((\sqrt{m})^3\right) = O\left(m^{3/2}\right)$. Thus the total number of triples is $O(m^{3/2}) + O(m^{3/2}) = O(m^{3/2})$. Because, the efficient version of both node iterator and edge iterator test each of the triples in a network exactly once, the complexity of both of these algorithms are bounded by $O(m^{3/2})$.

There are a set of recent variants of edge iterator, Forward[21] by Schank T. and new-listing[24] by Latapy M. Forward orders the vertices by increasing degree. Latapy's method sorts the adjacency list and uses iterators to efficiently compute set intersection. The complexity of these methods still remains $O(m^{3/2})$, but real-world execution time may be smaller. Schank[21] has performed a through comparison among various exact triangle counting methods over a large number of real-life and synthetic networks.

APPROXIMATE TRIANGLE COUNTING

Time complexity of node iterator and edge iterator algorithm are $O(m^{3/2})$. For very large graphs with

hundreds of millions of edges, this cost may still be deemed costly. So, in recent years approximate triangle counting algorithms have become popular. Methods for approximate triangle counting do not list (enumerate) the triangles, rather they give an approximate count of triangles—sometimes, with an approximate guarantee. Also, their execution time is smaller, typically by order of magnitudes. For many applications, trading the large running time for a good approximation is adequate; for instance, to analyze the evolution pattern of a network, an approximate transitivity result is generally acceptable.

In existing literatures, there exist a few variants of approximate triangle counting methods. Algorithms in first variant are based on uniform triangle sampling by graph sparsification, the algorithms in the second variant are based on triple sampling, and the algorithms in the third variant are based on a stochastic version of node or edge iterator. There also exists an approximate triangle counting method, which uses ideas from linear algebra.

Graph Sparsification-Based Methods

The idea of graph sparsification-based method for triangle counting is to sparsify the graph by probabilistically deleting a subset of edges in the graph and then extrapolating the triangle count in the original graph from the exact triangle count in the sparse graph. Because sparse graph is much smaller than the original graph, the triangle counting in the sparse graph can be performed typically in a fraction of time, which makes the approximate method substantially faster than an exact counting method. Such a method can also be viewed as uniform triangle sampling-based method, because the triangles in the sparse networks are sampled with a uniform probability over all the triangles in the original network.

Tsourakakis et al.[25] proposed one of the earliest approximate triangle counting method called DOULION, which works by graph sparsification. Given a graph $G(V, E)$, DOULION keeps each edge of G with p probability and remove the edge with $1 - p$ probability to generate a sparse graph, G_s. It then runs an exact triangle counting method on G_s to obtains $t(G_s)$—the exact triangle count of G_s. Each triangle in the original graph is retained in the sparse graph when all three of its edges are retained, for which the probability is p^3. Hence, the probability of sampling a triangle from G in G_s is p^3 and thus, the expected count of the triangles in the original graph is $\hat{t}(G) = (1/p^3) \cdot t(G_s)$. For large network with millions of vertices p value as small as 0.01 can provide very good approximate triangle count. A p value of 0.01 yields almost 100 times speed-up in the running time over the running time of an exact counting method.

Pagh and Tsourakakis[26] proposed another graph sparsification work for approximate triangle counting which they named as 'colorful triangle counting.' For each vertex, this method assigns a color between 1 and N uniformly and retains only those edges for which both the endpoints have the same color, all other edges are removed. Identical to DOULION, an exact counting algorithms is used to find the number of triangles, $t(G_s)$, in the sparse network G_s. If $p = 1/N$, each triangle in the original network is retained in the sparse network with probability p^2. This is so because when two of the edges of a triangle is monochromatic, the third edge is also monochromatic by force and the probability of retaining two edges is p^2. So the expected count of the triangles in the original graph for this method is $\hat{t}(G) = (1/p^2) \cdot t(G_s)$. Like DOULION, the sparse graph G_s contains each edge with probability p, but unlike DOULION, this method retains (or samples) each triangle with a probability p^2, instead of p^3. As this method samples more triangles for the same p value, it has a better accuracy than DOULION. Pagh et al. had used variance analysis, and then proved probabilistic bounds on the approximation ratio of triangle estimation using this method. They also proposed a MapReduce-based distributed implementation of this algorithm.

Very recently, Etemadi et al.[27] proposed another method which is an adaptation of DOULION. Similar to DOULION, this method also samples edges of a given graph G with a uniform probability p to obtain the sparse graph G_s. However, besides counting triangles in G_s, it also checks whether the missing edge of each open triple in G_s exists in the original graph G; if yes, that partial triangle is also counted with the count of actual triangle in G_s. A triangle in G has a p^2 probability to be counted in $t(G_s)$, because a triangle in G will be accounted for in G_s as long as two of its edges are retained in G_s. So, the expected count of the triangles in the original graph using this method is $\hat{t}(G) = (1/p^2) \cdot t(G_s)$. By estimating the variance of the estimation, authors also provided a way to choose the value of p for achieving a targeted range of relative standard error. They have also proved that compared to their method, DOULION always needs more samples to achieve the same level of accuracy. Accuracy of this method is comparable to Pagh et al.'s method, because both methods sample a triangle with the same probability.

Note that all the graph sparsification-based methods need an exact triangle counting algorithm which runs on the sparse network. So, their execution time depends on the performance of the exact triangle counting algorithm. Clearly, Etemadi et al.'s method is the costliest among all of these, because besides counting triangles in the sparse network G_s, it also needs to check the graph data structure of G for determining the existence of the missing edges of an open triple in G_s.

Triple Sampling-Based Method

The graph sparsification-based method samples triangles, but there is another family of approximate triangle counting algorithms, which samples triples, instead of triangles. A triple sampling method lends to a triangle approximation algorithm by computing an unbiased estimate of transitivity (defined in the *Background*). This estimate is equal to the fraction of triples that are closed out of all the sampled triples. If T is a set of sampled triples, and $\hat{\gamma}$ is the estimate of transitivity

$$\hat{\gamma} = \frac{\sum_{t \in T} \mathbb{I}_{t \text{ is closed}}}{|T|}, \tag{9}$$

here $\mathbb{I}$ is an indicator random variable. Then using Eq. (5), an approximate estimate of total number of triangles in the graph is equal to $(1/3)\hat{\gamma}\cdot|\Pi|$, where $|\Pi| = \sum_{i=1}^{n} \binom{d(u_i)}{2}$, is the total number of triples in the given graph (see Eq. (1)). Such a method has been used in several works.[12,16,28]

A key requirement for computing an unbiased estimate of transitivity is to sample triples from a uniform distribution, which is a non-obvious task. The simplest approach to sample a triple is to select a node v uniformly and then select two of v's neighbors (u and w) uniformly. This method samples a triple $\langle u, v, w\rangle$ with v as the center node. However, this method does not sample a triple uniformly, because the number of triples centered at a node v is $|\Pi_v| = \binom{d(v)}{2}$, which is nonuniform over the vertices for a general graph. Hence triple $\langle u, v, w\rangle$ is sampled with probability $1/(n \cdot |\Pi_v|) \propto 1/|\Pi_v|$. So, the triples that are centered around high degree vertices will be under-sampled and those that are centered around low degree vertices will be over sampled.

Schank and Wagner[28] have proposed the earliest algorithm for approximating transitivity of a network by sampling triples uniformly. Their idea is as follows: first, sample a vertex v in proportional to the number of triples centered around that vertex. Then with uniform probability return one of the triples that are centered around v. If Π is the set of all triples in G and Π_v is the set of triples centered at node v then $\Pi = \sum_{i=1}^{n} \Pi_i$ and $|\Pi_v| = \binom{d(v)}{2}$. For uniform triple sampling, we sample the center vertex v with probability $|\Pi_v|/|\Pi|$ and then return the triple $\langle u, v, w\rangle$ by uniformly selecting one of the triples in Π_v. Thus, the probability of sampling the triple $\langle u, v, w\rangle$ is uniform,

$$P(\text{triple } \langle v,u,w\rangle \text{ is sampled}) = \frac{|\Pi_u|}{|\Pi|} \cdot \frac{1}{\binom{d(u)}{2}} = \frac{1}{|\Pi|}, \tag{10}$$

as desired. Schank et al. then used the set of sampled triples to approximate the transitivity using Eq. (9). However, one can easily obtain an approximate count of triangles in a graph G, $t(G)$, by using the estimated transitivity in the following equation:

$$\hat{t}(G) = \frac{1}{3}\cdot\hat{\gamma}\cdot|\Pi|, \tag{11}$$

the value of $|\Pi|$ is known, as is shown in Eq. (1). Kolda et al.[16] have reinvented the same method in a later work. Both Schank et al. and Kolda et al. have proved approximation error bound by using Hoeffding bound-based concentration inequality.

In a recent article, Al Hasan[29] has provided methodologies for obtaining an unbiased estimate of transitivity even for the case when the sampling algorithm samples the triples from a non-uniform distribution. He has used the idea of IS (discussed in the Background Section) for this task. For instance, the simplest (but biased) triple sampling method that we discussed at the beginning of this subsection samples each triple in inverse proportional to the number of triples centered at the first sampled vertex, v. If we want an unbiased estimate of transitivity, we need to have a uniform target distribution. But the triples are sampled from a distribution which is proportional to $1/|\Pi_v|$, so by using Eq. (7) of IS an unbiased estimate of transitivity can be computed as below. Consider we have a triple sample set $T = \{t_i = \langle u_i, v_i, w_i\rangle\}_{1\le i\le|T|}$, where v_i is the center node of the triple t_i. The importance weight is

$$w(t_i) = \frac{|\Pi_{v_i}|}{\sum_{j=1}^{|T|} |\Pi_{v_j}|}. \tag{12}$$

Now, the unbiased estimate of transitivity is simply:

$$\hat{\gamma} = \sum_{t_i \in T} \left(w(t_i) \cdot \mathbb{I}_{t_i \text{ is closed}} \right) = \frac{\sum_{t_i \in T} |\Pi_{v_i}| \cdot \mathbb{I}_{t_i \text{ is closed}}}{\sum_{t_j \in T} |\Pi_{v_j}|} \cdot \quad (13)$$

Approximation by Vertex or Edge Sampling

Another simple approximate triangle counting algorithm can be built by using a probabilistic counterpart of node iterator or edge iterator method. In the exact node iterator algorithm, we count the number of triangles by summing the count of triangles that are incident to each of the nodes. Instead of summing the count over all the vertices, we can simply sum over p fraction of uniformly sampled vertices, and then approximate the total count by dividing the sum value by p. This provides an approximate node iterator-based triangle counting algorithm. Likewise, we can build an approximate edge iterator-based algorithm by counting triangles over p fraction of edges. Rahman and Al Hasan[13] have proposed this method and they have shown that for large networks, this method achieves very good accuracy. They have also shown that an edge iterator-based sampling method achieves better accuracy than a node iterator-based sampling method. Note that, these methods do not sample the triangles uniformly, but the estimation is still unbiased because the expectation of triangle count is taken over the edges, which are sampled uniformly.

Linear Algebra-Based Method

We have shown earlier that if $\mathbf{A}$ is the adjacency matrix of an undirected network G, the total number of triangles in a graph G, $t(G) = (1/6)Tr(\mathbf{A}^3)$. If λ_i is the eigenvalue of $\mathbf{A}$, λ_i^3 is the eigenvalue of $\mathbf{A}^3$, hence, $Tr(\mathbf{A}^3) = \sum_{i=1}^{n} \lambda_i^3$. Note that, because $\mathbf{A}$ is symmetric, all the λ_i are real numbers. For exact counting, it requires to obtain all the eigenvalues of the adjacency matrix $\mathbf{A}$—a costly task.

Fortunately, real-life networks have power-law property and due to this fact, the eigenvalues of its adjacency matrix are also skewed, typically following a power-law property. So if the eigenvalues are sorted by their absolute value (in other words, by their contribution to the sum), we can approximate the triangle count by taking only top-k eigenvalues. Note that using Lanczos method, top-k eigenvalues of a matrix can be easily computed in an incremental manner. This idea has been used in the EigenTriangle algorithm,[11] which accepts an adjacency matrix and a tolerance parameter. The tolerance parameter is used as a stopping criterion, as such that the ratio of $|\lambda_i^3|$ and $\sum_{i=1}^{n} \lambda_i^3$ has to be above the tolerance parameter. Although elegant, there are two key limitations of this method. First, the time-accuracy trade-off of this method is much poorer than other recently proposed approximate triangle counting methods. Second, no approximate guaranty is available regarding the accuracy of the method; in other words, there is no obvious relation between tolerance parameter and counting accuracy so that it can be set to achieve a desired level of approximation.

TRIANGLE COUNTING IN RESTRICTED ACCESS SCENARIO

As discussed before, for restricted access network random walk-based approximation methods are most suitable. Rahman and Al Hasan[12] have proposed a collection of random walk-based methods for approximating transitivity of a network in a restricted access scenario. If the total number of triples (Π) is known, then these methods can be used for approximating triangle count. Below we discuss two of the methods, one performing random walk over the vertices, and the other performing random walk over the triples.

Random Walk Over Nodes

Earlier we have shown that an approximate triangle counting algorithm can be obtained by first sampling a vertex v and then uniformly sampling one of the triples, $\langle u, v, w \rangle$, centered at vertex v. A random walk variant of this algorithm performs the first sampling task, i.e., sampling v, by a random walk over the graph. A crucial task, though, to ensure that we can design the random walk in such a way that the vertex v is sampled from a distribution, which enables unbiased triangle counting.

A simple random walk, which chooses the next vertex uniformly from the neighbors of currently visiting vertex has a stationary distribution, $\pi \sim d(\cdot)/2m$, where $d(\cdot)$ is the degree of a vertex. So, if we perform a simple random walk and return a triple $t = \langle u, v, w \rangle$ uniformly among all the triples incident to the currently visited vertex, v, the probability of sampling the triple t is equal to $\frac{d(v)}{2m} \times \frac{1}{\binom{d(v)}{2}}$, which is proportional to $1/(d(v) - 1)$, not uniform. Because unbiased triangle counting by transitivity approximation requires uniform triple sampling, we can use the idea of IS that we have discussed earlier.

If we have a triple sample set $T = \{t_i = \langle u_i, v_i, w_i\rangle\}_{1\le i\le |T|}$, where v_i is the center node of the triple t_i, then

$$\hat{\gamma} = \sum_{t_i\in T}\left(w(t_i)\cdot\mathbb{I}_{(t_i)\text{isclosed}}\right) = \frac{\sum_{t_i\in T}\left((d(v_i)-1)\cdot\mathbb{I}_{(t_i)\text{isclosed}}\right)}{\sum_{t_j\in T}(d(v_j)-1)}, \quad (14)$$

here, we used $w(t_i) = \frac{d(v_i)-1}{\sum_{t_j\in T}(d(v_j)-1)}$, using Eq. (8). Once an unbiased estimation of transitivity is obtained, triangle count can be approximated using Eq. (11).

However, Rahman et al.[12] did not use IS in their solution, they rather have used MH algorithm. Their solution, named Vertex-MCMC, works as follows: Use MH algorithm to design a random walk whose stationarity distribution $\pi \sim \binom{d(\cdot)}{2}$. Say, the random walk of a vertex-MCMC-based triple sampler is visiting a vertex a. To use MH algorithm, we need to uses a proposal distribution (q) to make a trial move; vertex-MCMC chooses q to be uniform over the neighbors of a; in other word, it chooses one of the vertices (say, b) from the adjacency list of a uniformly. Therefore, the proposal distribution $q(a, b) = 1/d(a)$, and $q(a, b)$ represents the probability of an adjacent node b to be selected from the node a. Similarly, $q(b, a) = 1/d(b)$. Now, using Eq. (6), the acceptance probability of the proposal move is as shown in Eq. (15).

$$\alpha(a,b) = \min\left\{1, \frac{\binom{d(b)}{2}\cdot\frac{1}{d(b)}}{\binom{d(a)}{2}\cdot\frac{1}{d(a)}}\right\} = \min\left\{1, \frac{d(b)-1}{d(a)-1}\right\}. \quad (15)$$

The above MCMC random walk ensures that each vertex is sampled from the target distribution, which is proportional to the number of triples at each vertex. So, while visiting a vertex v using the above random walk, a uniform triple sampler simply returns a triple $\langle u, v, w\rangle$, which is one of the triples centered at v, selected with uniform probability over all such triples.

Random Walk Over Triples

Instead of sampling triples in two stages (first sample a vertex, and then a triple which is centered at that vertex), we can also sample triple directly. Rahman and Al Hasan[12] have also proposed such an approach, which they call triple-MCMC. In this method, a random walk is designed which walks over the space of triples in a network. To facilitate this walk, a neighborhood graph is defined over the set of triples. Any reasonable neighbor definition works; Rahman et al. consider two triples as neighbor if they have two vertices in common. For example, the triples $\langle 1, 2, 3\rangle$ and $\langle 2, 3, 4\rangle$ are neighbors because they have two common vertices, {2, 3}. Starting from an arbitrary random triple, the random walk continues over the triples by moving from one triple to a neighbor triple based on the above neighborhood definition. Set of possible neighbors of the currently visiting triple can be computed on the fly by finding other triples that can be obtained by replacing exactly one of the vertices of the current triple. Thus, the walk resembles sampling of dependent triples, where a sampled triple shares two vertices with the previously sampled triple.

For the purpose of triangle counting, the triples need to be sampled uniformly. But a simple random walk does not guarantee this because the triples have different degree in the neighborhood graph on which the random walk proceeds. To ensure uniform sampling, Rahman et al. proposed to adopt MH algorithm. Let's assume that the random walk is visiting a triple t. For MH's proposal distribution (say q), they choose one of the triples from t's neighborhood (say, s) uniformly. So, $q(t, s) = 1/(|\Gamma(t)|)$. Here, $\Gamma(t)$ is the set of neighbors of the triple t. Using Eq. (6), the acceptance probability of the proposal move is obtained as shown below:

$$\alpha(t,s) = \min\left\{\frac{1\cdot\frac{1}{|\Gamma(s)|}}{1\cdot\frac{1}{|\Gamma(t)|}}, 1\right\} = \min\left\{\frac{|\Gamma(t)|}{|\Gamma(s)|}, 1\right\}. \quad (16)$$

Once a desired number of triples has been sampled, the fraction of closed triples over all the sampled triples provides an unbiased estimation of transitivity, from which an approximate count of triples can be returned by using Eq. (11). Note that for this method also, instead of using MH, one can perform simple random walk and then use IS for obtaining an unbiased statistics of transitivity. Note that, a random walk-based triple sampling method can approximate transitivity, but not triangle count unless the total number of triples, $|\Pi|$, is available.

TRIANGLE COUNTING ON STREAM DATA

For many datasets, graphs are too large to fit in main memory, but it is easier to access a graph as

streaming edges, such that the edges appear on the stream in an arbitrary order sequence. Even if a graph fits in the main memory, a streaming edge access model of graph is preferred for some computation model, such as, MapReduce. The main restriction in a streaming access model is that we cannot save all the edges of a graph in memory, so statistics of each edge must be processed instantaneously as the edge appears on the stream. For such restricted access model, it is allowed to go over the edge stream of a graph multiple times (aka, multi-pass streaming algorithm). Going over the input graph stream multiple times may appear inefficient, but if the graph does not fit in main memory, going over multiple passes is much cheaper than trying to access a large number of random vertex (or the adjacency list of a random vertex) in the disk.

The earliest streaming triangle counting method is proposed by Bar-Yossef et al.[7] Their method is based on stream-reduction, a general idea for computation over data stream, which is also proposed in the same paper. Stream-reduction idea can be used for approximating frequency moment over data stream, which they used for approximating triangles. Unfortunately, their method is mostly for theoretical interest and is not practical for approximating triangles in real-world networks.

Buriol et al.[18] have proposed several methods for triangle counting over edge stream. Their first method is a three-pass algorithm. In the first pass, the number of edges (m) and the number of vertices (n) are counted simply by using two counters. In the second pass, an edge $e = (a, b)$ is sampled uniformly from the set of edges. Also, a vertex $v \in \{V \setminus \{a, b\}\}$ is chosen uniformly. This leaves us with a triple (not necessarily connected) $\langle a, b, v\rangle$. Then in the third pass, the method simply tests whether $(a, v) \in E \wedge (b, v) \in E$, if yes, then $\beta = 1$, otherwise $\beta = 0$. In this way, β is an estimate of the triangles over all possible edge-plus-a-vertex combination. If T_1 is the number of disconnected triples, T_2 is the number of connected open triple, and T_3 is the number of triangles in the graph, then $T_1 + 2T_2 + 3T_3$ is equal to $m \cdot (n - 2)$, the population size. Besides, we also have the expectation of β, $\mathbb{E}[\beta] = \frac{3T_3}{(T_1+2T_2+3T_3)} = \frac{3T_3}{(m\cdot(n-2))}$. So, an approximate unbiased triangle estimate is equal to $(1/3)m(n-2)\mathbb{E}[\beta]$. This estimate can be improved by running s copies of this sampling and averaging their corresponding estimates of $\{\beta_i\}_{1\le i\le s}$. Thus, the final estimate of triangle count is $m(n-2)/3s\sum_{i=1}^{s}\beta_i$. Because each sample only takes $O(1)$ space, for s samples the total space is bounded by $O(s)$, which is linear with the number of samples but independent with the size of the network. Chernoff's inequality can be used to prove probabilistic bound on the approximation result.

The above three-pass algorithm can be converted to a two-pass algorithm by combining the first two passes in a single pass. That is, counting n, m and uniform sampling of edge (a, b) and vertex v can actually be done in the same pass using reservoir sampling.[30] The key idea of reservoir sampling for sampling an object (uniformly) from a stream of objects is to keep a running count of the number of objects as new objects are seen in the stream. The first object in the stream is always saved in the reservoir, but the i'th object replaces the object in the reservoir with $1/i$ probability only. When the stream ends, the object in the reservoir is the sampled object, chosen uniformly from the stream without prior knowledge of the number of objects in the stream.

Buriol et al.[18] have actually proposed a one-pass version of their three-pass algorithm, which combines the works of all three passes in one-pass, as below. Say, the edge e appears in the stream, while (a, b) (uniformly sampled edge) and v (uniformly sampling vertex) are in the reservoir. If $e = (a, v)$, set the boolean variable $x = 1$ and if $e = (b, v)$, set the boolean variable $y = 1$. Once the stream ends, if $x = y = 1$, set $\beta = 1$, otherwise $\beta = 0$. $\beta = 1$ represent the fact that we have sampled a triangle (a, b, v) where (a, b) is the uniformly sampled edge and v is the uniformly sampled vertex, both in the reservoir. However, $\mathbb{E}[\beta] = T_3/(T_1 + 2T_2 + 3T_3)$, which is one-third (note the missing 3 in the numerator) of the $\mathbb{E}[\beta]$ of the three-pass method. This is due to the fact that for the one-pass version, the triangle $\langle a, b, v\rangle$ is counted (i.e., $\beta = 1$) only if the edge (a, b) appears on the stream before the edges (b, v) and (a, v) (probability of this event to happen is 1/3); on the other hand, the three-pass version counts the triangle for any ordering of the 3 edges. Besides this, three-pass and one-pass method are identical. So, similar to the three-pass method, the estimate of triangles is equal to $m(n-2)/s\sum_{i=1}^{s}\beta_i$, if s parallel copies of sampling are run together.

Jha et al.[15] have provided another one-pass streaming triangle counting algorithm which is a stream variant of triple sampling. Say, for a graph G, e_1, e_2, …, e_m is a sequence of distinct edges. Then $\{G_t\}_{1\le t\le m}$ are the graphs in time t, formed by the edge set $\{e_i | i \le t\}$; clearly $G_m = G$. With the arrival of an edge on the stream, the method performs an update on the estimate of the triangles in graph G_t. Once the stream ends, the estimated value is equal to the count of triangles in the entire graph, $G_m = G$. The main

idea of this method is to use two reservoir arrays R_e and R_w of size s_e and s_w, respectively. For the streaming graph, G_t, R_e stores a uniform subset of s_e edges from the graph G_t and R_w stores a uniform sample of s_w triples from the graph G_t. With the arrival of the edge e_t, the method first counts the number of triples in R_w that the edge e_g completes. Then it updates both R_e and R_w to maintain uniform sampling. R_e is updated by inserting e_t in R_e probabilistically (by following reservoir sampling's idea); if this insertion is successful, then R_w is updated by inserting (again probabilistically by reservoir sampling) the new triples that are formed by the edge e_t with the already existing edges in R_e. Also, for any state of R_e and R_w, the statistics of total triples (*tot_triples*) formed by the edges in R_e is computed. It is easy to see that if the triples in R_w are sampled uniformly over all the triples, the fraction of triples that are closed in R_w (denoted as ρ) approximates the ratio $t(G)/|\Pi|$. But, to obtain triangle count from this ratio, we also need to know the total number of triples in G ($|\Pi|$), which they estimates by using Birthday Paradox. The expected number of triangles is then $[\rho t^2/s_e(s_e - 1)] \times tot\_triples$.

DISTRIBUTED AND PARALLEL TRIANGLE COUNTING

We can use streaming method to approximate triangle count of huge graphs that do not fit in the main memory of conventional machines. However, if we strive for exact triangle counting on such large graphs, distributed computing provides a viable option. In this section, we will give an overview of parallel and distributed triangle counting methods. The parallel methods use multi-core machines and the distributed methods run on MapReduce platform.[31]

One of the earliest works on distributed triangle counting using MapReduce framework is proposed by Suri and Vassilvitskii.[17] They proposed a MapReduce variant of an efficient node iterator algorithm, which is shown in Algorithm 3. This algorithm has two rounds: first round generates all length-two paths in the graph from the edge list, in parallel. Second round counts how many of the length-two paths generated in the first round have a closing edge in the graph. To accomplish this, the second round takes the output of the first round (denoted as Type 1 input in Algorithm 3) along with the original edge list (denoted as Type 2 input in Algorithm 3) as inputs. Suri and Vassilvitskii[17] also proposed a graph partition-based MapReduce algorithm, which first partitions the graph and then runs an exact triangle counting method on each partition, in parallel. Later, Park and Chung[32] identify redundant computation in Suri et al.'s method and proposed another partitioning method called *Triangle Type Partitioning*. Pagh and Tsourakakis[26] proposed a MapReduce version of their edge sampling-based method, however, this method provides an approximate count only as it is based on sampling.

Arifuzzaman et al.[33] proposed a distributed memory-based parallel algorithm for triangle counting using message passing interface. This algorithm partitions the graph based on disjoint subsets of nodes (core nodes), and generates induced subgraph from the subset of nodes and their neighborhood. Each induced subgraph is assigned to a machine and triangles are counted independently in each machine for corresponding core nodes. Last, it combines the results from all the machines to get global triangle count.

Algorithm 3 MapReduce Node Iterator Algorithm

Require: $G(V, E)$

1: **Map 1:** Input: $\langle (u, v); \phi \rangle$
2: **if** $v > u$ **then**
3: emit $\langle u; v \rangle$
4: **end if**
5: **Reduce 1:** Input: $\langle v; S \subset adj(v) \rangle$
6: **for** $(u, w)|u, w \in S$ **do**
7: emit $\langle v; (u, w) \rangle$
8: **end for**
9: **Map 2:**
10: **if** Type 1 Input: $\langle v; (u, w) \rangle$ **then**
11: emit $\langle (u, w); v \rangle$
12: **end if**
13: **if** Type 2 Input: $\langle (u, v); \phi \rangle$ **then**
14: emit $\langle (u, v); \$ \rangle$
15: **end if**
16: **Reduce 2:** Input: $\langle (u, w); S \subseteq V \cup \{\$\} \rangle$
17: **if** $\$ \in S$ **then**
18: **for** $(u, w)|u, w \in S$ **do**
19: emit $\langle v; (u, w) \rangle$
20: **end for**
21: **end if**

Kim et al.[34] proposed a disk-based framework for triangle counting using multi-core CPU. They categorize the triangles into two types; internal triangles,

for which the adjacency lists of two connected nodes are in the main memory and external triangles, for which only one adjacency list is in the main memory. This framework stores adjacency list as slotted page structure in disk and use asynchronous read to load required page into buffer memory. The buffer memory is split such that it contains pages (adjacency lists) corresponding to both types of triangles (internal and external) at the same time. The triangles are counted when the adjacency lists are in the buffer, and to avoid redundancy each page is loaded in the buffer exactly once. In this framework, both type of triangles are counted by two separate threads and for maximum utilization of CPU, it also uses thread morphing if one of the threads completes its work and terminates. The framework also uses *openMP* to use additional available threads to count internal triangle counting and later use thread morphing, if required.

Shun and Tangwongsan[35] proposed a multi-core parallel algorithm for shared memory machines. The proposed algorithm has two steps: in the first step, each node is ranked basked on degree and ranked adjacency list of each node is generated, which contains only higher ranked nodes than the current node; the second step counts triangles from the ranked adjacency list for each node. For the first step, after ranking each node, the generation of ranked adjacency list is easily parallelizable as this task is independent for each node. For the second step, an array is created to put values of each locally counted triangles, here the size of the array is the total size of ranked adjacency list for all node. Lastly, the actual triangle count is the summation of the values in the array. Rahman and Al Hasan[13] also proposed a multi-core parallel algorithm for triangle counting, which distributes the loop of node/edge iterator algorithms across multiple cores.

EXPERIMENTAL COMPARISONS

Schank[21] has performed a thorough experimental comparison among different exact triangle counting methods. In this survey, we make a thorough comparison among various approximate triangle counting methods. For the comparison, we consider two sparsification-based methods: 'DOULION' and 'Colorful triangle counting.' For both the methods, the triangles in the sparse network is counted exactly by using efficient edge iterator algorithm. So, we call these methods *doulion_Edgeiter*, and *color_Edgeiter*, respectively. We also consider two triple sampling-based methods: 'Direct Sampling' (Eq. (10)) and uniform sampling with importance weight adjustment, hereby named as *Uniform_Importance* (Eq. (13)). The sampling version of edge iterator[13] hereby named as *Sampled_edgeIter* is also considered. We further consider two of the restricted access-based methods: random walk over nodes with importance weight adjustment hereby named as *randWalk_importance* (Eq. (14)), and 'vertexMCMC'[12], i.e., MCMC walk over nodes. We implement all the above methods ourself by using identical graph data structures and edge existence query module. This ensures fairness among the comparison. We intentionally omitted some of the approximate triangle counting methods in this experiment after we found that their performance is substantially poorer than the performance of the methods we report here.

For the experiment, we use four large graphs collected from the KONECT [the Koblenz Network Collection (http://konect.uni-koblenz.de/networks/)]. The first, 'as-skitter' is a network of autonomous systems on the Internet, where autonomous systems are nodes and connection between them are edges. The 'flickr,' 'livejournal' and 'orkut' are social networks, where each node is a user and an edge between users shows friendship between the users. The basic statistics of the datasets is shown in Table 2, where $|V|$, $|E|$, and $t(G)$ are the number of vertices, the number of edges and the number of triangles in the graph, and $time(ms)$ is the time taken in millisecond by edge iterator with hashing method, which is one of the fastest exact triangle counting method.

Comparing approximate triangle counting methods is tricky, as many of these methods are sampling-based methods and hence, they have error-runtime trade-off, i.e., if we take more samples, the error decreases but runtime increases, and vice-versa. Besides, the population from which these methods sample are different; some sample triples, some sample triangles, and some sample edges. So it is not easy to simply compare the error of these methods using a unified sampling factor. So, we report both error and runtime of a method as a point on a graph. For each method, we take three points by running them for three different sampling factor values. For a given method, we connect the three points by a piece-wise linear curve. To obtain the data of this graph, the

TABLE 2 | Basic Statistics of the Datasets

Dataset	$\|V\|$	$\|E\|$	$t(G)$	Time (ms)
as-skitter	1.69M	11.09M	28.77M	38, 989.82
flickr	1.72M	15.55M	548.17M	174, 216.12
liveJournal	5.20M	48.71M	310.87M	231, 541.28
orkut	3.07M	117, 19M	627.58M	867, 634.33

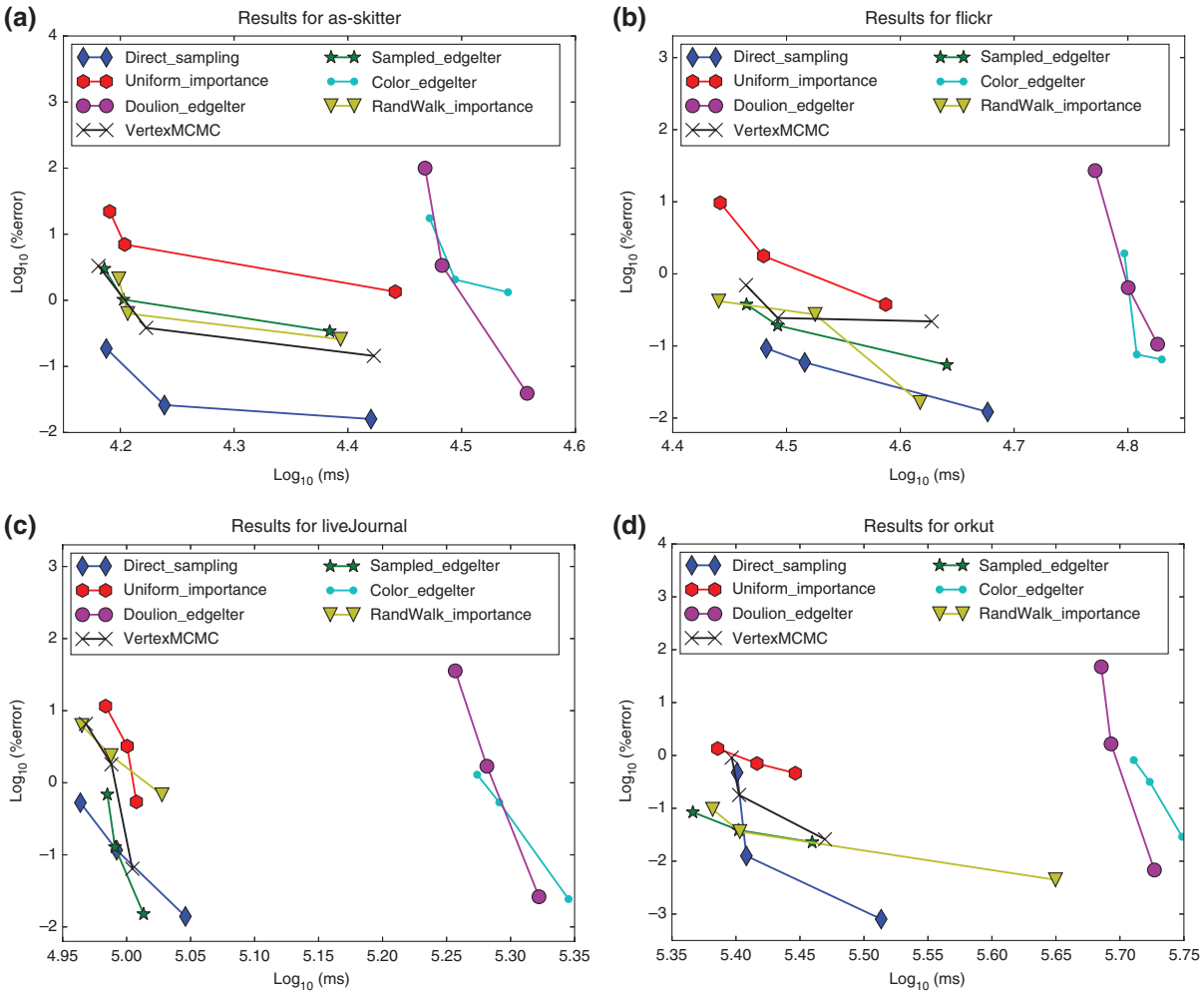

FIGURE 2 | Comparison of approximation methods.

error is computed as percentage error, which is equal to $\frac{|\text{exact-count} - \text{approx-count}| \times 100}{\text{exact-count}}$ and runtime is reported in millisecond.

For the sparsification-based methods and edge sampling-based methods, we use the sampling probability $p \in \{0.001, 0.01, 0.1\}$. For both, full access- and restricted access-based triple sampling methods, number of samples $|T|$ are selected as {0.001%, 0.01%, 0.1%} of $|\Pi|$, where $|\Pi|$ is the total number of triples. These values of p and $|T|$ help us to understand how time and errors are related for the approximation methods. For all the methods, we use a serial version of the method, although they can easily be parallelized to run faster. We run all our experiments on a machine with AMD 2.3 GHz processor, 128 GB RAM, and Red Hat Enterprise Server Release 7.3 OS. For all data points, the value is computed after running the corresponding method for 10 times and then taking the average value of those runs.

In Figure 2, we show four charts, each for one of the datasets. Each chart has seven piece-wise linear curves, each representing one of the approximate triangle counting methods. Each curve has three points, representing log(error) versus log(runtime) of a method for three sampling rates (p values). All the curves have a negative slope showing the inverse relationship between error and runtime, i.e., in the lowest sampling rate they have the smallest runtime but the largest error. The data point that is closest to the origin is the best as it has both small error and small runtime.

If we observe the graphs carefully, we can conclude that both the sparsification-based methods take more time to achieve as high accuracy as other approximation methods. For lower value of p, DOULION has very high error and colorful sampling method performs consistently better than DOULION in that case, but takes more time. Among other approximation methods, almost all methods take similar amount of time but have different error values. The best approximation method is 'Direct Sampling,'

which always provides the lowest error. *Sampled_edgeIter* method also performs very good consistently and on some occasions it is faster than 'Direct Sampling' method, but with a higher error. The indirect sampling-based methods (MCMC, and -importance weight-based sampling) are fast, but they generally have a higher error than the direct sampling-based method.

OTHER RELATED COUNTING TASKS

Although triangle counting task has received enormous attention, there are other counting tasks that count higher order graphical structures beyond triangles. Obvious extension of a triangle is a k-clique—a complete graph with k vertices, and a related counting task is to count the distinct k-cliques in a given graph for a user chosen value of k. However, counting k-clique is much more difficult than counting triangles, as the number of k-cliques increases exponentially with the value of k. An efficient sequential solution for k-clique counting algorithm can be obtained by modifying the well-known Bron–Kerbosch algorithm.[36] But, this algorithm does not scale to large real-life networks. To solve the lack of scalability issue, Finocchi et al.[37] proposed a distributed solution for k-clique counting algorithm which runs on MapReduce.

Graphlet counting is another related task which has become very popular in recent years because of its wide applicability in different domains for various tasks including network classification,[38] biological network comparisons,[39,40] image classification,[41] and building graph kernels for chemoinformatics.[42] For a given k, all possible k-size graphical topologies are collectively referred as graphlets. For undirected graphs, there are 2 size-3 graphlets (open triples and closed triples), 6 size-4 graphlets, and 21 size-5 graphlets. The number of distinct graphlets increases exponentially with the size of the graphlet. The counting task is to obtain the count of the total number of distinct-induced occurrences of all graphlets (of a given size) in a given network.

In existing literature, several works exist for solving the graphlet counting problem exactly, examples include FANMOD,[43] RAGE,[44] GRAFT,[38] and ESCAPE.[45] The earliest among the above, FANMOD and RAGE, are very slow, mainly because they use an enumeration-based approach. GRAFT is relatively better than the above two as it only enumerates tree graphlets and then counts the other graphlets by efficient edge existence check. Hočevar and Demšar[46] provided an efficient method, named ORCA, which does not enumerate all graphlets, but counts a subset of graphets and calculates other graphlet counts using a combinatorial approach. However, ORCA is not highly scalable when it needs to handle huge real-world graph with millions of nodes/edges. Recently, Ahmed et al.[47] provided a highly efficient and scalable method, namely PGD, for graphlet counting by utilizing graphlet transition. Graphlet transition relates two graphlets by using add/removal of an edge, which helps to calculate count of one graphlet using the count of other smaller graphlet. PGD is scalable, but works for upto four-sized graphlets. Pinar et al.[45] proposed a method (ESCAPE), which can provide count of five size graphlet very efficiently. Similar to PGD, ESCAPE also calculates counts of four- and five-sized graphlets using counts of specific set of other (mostly smaller) graphlets.

Similar to the case of triangle counting, approximate counting method has also been popular for graphlet counting. Rand-ESU (available in FANMOD library) is one of the earliest approximate graphlet counting method, but its accuracy is poor. In recent years, Bhuiyan et al.[48] has proposed a method called GUISE, which uses MCMC sampling for obtaining uniform samples of graphlets through random walk. GUISE samples upto size-5 graphlets, but Saha and Al Hasan[49] have generalized the method so that it can sample graphlets of any size. Wang et al.[50] proposed an improved and more efficient method based on random walk. Rahman et al.[51] proposed edge sampling-based approximation method (GRAFT), which aligns sampled edge with a specific edge of a graphlet and then enumerate all embeddings of the graphlet. Jha et al.[52] propose three-path sampling-based method for four size approximate graphlet counting, which has been proved to be more efficient than GUISE and GRAFT. Recently, Bressan et al.[53] proposed color coding-based approach and show its superiority over MCMC-based method.

CONCLUSIONS

Triangles play a very important role in network analysis. In social networks, triangles represent transitivity, which is important for understanding network evolution over the time. In biological networks, several motifs have been found to be triangle representing various biological pathways. Due to the importance of triangles, enumeration and counting them in large networks are important tasks. Both enumeration and counting of triangles have been studied for a long time, but in recent years, there has

been a renewed interest in triangle counting methods considering approximate counting, parallel and distributed implementation, and restricted and streaming data access scenarios.

In this survey, we discuss the existing methods of triangle counting, ranging from sequential to parallel, single-machine to distributed, exact to approximate, and off-line to streaming. We place more emphasis on the recent methods, specifically on the methods for approximate triangle counting by sampling. We also show some experimental comparison of the approximate triangle counting methods by implementing them with a uniform data structures. Our results show that triple sampling-based methods are superior over other approximate triangle counting methods, both in terms of accuracy and runtime. Future works in this direction will consider counting higher order graphical structures having more than three vertices. Some works on counting higher order structures have already emerged, but given that it is a very active area of research, we expect that many more studies will come in near future.

ACKNOWLEDGMENT

This research is supported partly by NSF-1149851 grant and a Research Award from CareerBuilder.

REFERENCES

1. Watts DJ, Strogatz S. Collective dynamics of 'small-world' networks. *Nature* 1998, 393:440–442.
2. Luce RD, Perry AD. A method of matrix analysis of group structure. *Psychometrika* 2001, 14:95–116.
3. McPherson M, Smith-Lovin L, Cook JM. Birds of a feather: homophily in social networks. *Annu Rev Soc* 2001, 27:415–444.
4. Aggarwal C, Subbian K. Evolutionary network analysis: a survey. *ACM Comput Surv* 2014, 47:10:1–10:36. https://doi.org/10.1145/2601412.
5. Becchetti L, Boldi P, Castillo C, Gionis A. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In: *Proc. of 4th ACM SIGKDD*, 2008, 6–24.
6. Eckmann JP, Moses E. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proc Natl Acad Sci U S A* 2002, 99:5825–5829.
7. Bar-Yossef Z, Kumar R, Sivakumar D. Reductions in streaming algorithms, with an application to counting triangles in graphs. In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '02, Philadelphia, PA, USA, 2002, 623–632. Society for Industrial and Applied Mathematics. ISBN: 0-89871-513-X. Available at: http://dl.acm.org/citation.cfm?id=545381.545464.
8. Palla G, Dereny I, Farkas I, Vicsek T. Uncovering the overlapping community structure of complex networks in nature and society. *Nature* 2005, 435:814–818.
9. Itai A, Rodeh M. Finding a minimum circuit in a graph. In: *Proceedings of the Ninth Annual ACM Symposium on Theory of Computing*, STOC '77, 1977, 1–10.
10. Alon N, Yuster R, Zwick U. Finding and counting given length cycles. *Algorithmica* 1997, 17:209–223.
11. Charalampos E, Tsourakakis E. Fast counting of triangles in large real networks without counting: algorithms and laws. In: *2008 I.E. 8th International Conference on Data Mining*, 2008, 608–617.
12. Rahman M, Al Hasan M. Sampling triples from restricted networks using MCMC strategy. In: *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM 2014, Shanghai, China, 3–7 November, 2014, 1519–1528. 10.1145/2661829.2662075.
13. Rahman M, Al Hasan M. Approximate triangle counting algorithms on multi-cores. In: *Proceedings of the 2013 I.E. International Conference on Big Data*, Santa Clara, CA, USA, 6–9 October, 2013, 127–133. 10.1109/BigData.2013.6691744
14. Tsourakakis CE, Kang U, Miller GL, Faloutsos C. Doulion: counting triangles in massive graphs with a coin. In: *Proceedings of the Fifteen ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, 2009.
15. Jha M, Seshadhri C, Pinar A. A space efficient streaming algorithm for triangle counting using the birthday paradox. In: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, New York, NY, USA, ACM, 2013, 589–597. ISBN: 978-1-4503-2174-7. 10.1145/2487575.2487678
16. Kolda TG, Pinar A, Seshadhri C. Triadic measures on graphs: the power of wedge sampling. In: *SIAM Data Mining*, SIAM, 2013, 10–18.
17. Suri S, Vassilvitskii S. Counting triangles and the curse of the last reducer. In: *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, 2011, 607–614.
18. Buriol LS, Frahling G, Leonardi S, Marchetti-Spaccamela A, and Sohler C. Counting triangles in data streams. In: *Proceedings of the Twenty-fifth ACM SIGMOD-SIGACT-SIGART Symposium on*

Principles of Database Systems, PODS '06, New York, NY, USA. ACM, 2006, 253–262. ISBN: 1-59593-318-2. 10.1145/1142351.1142388

19. Barabasi A-L, Albert R. Emergence of scaling in random networks. *Science* 1999, 286:509–512.
20. Newman MEJ, Watts DJ, Strogatz SH. Random graph models of social networks. *Proc Natl Acad Sci U S A* 2002, 99(suppl 1):2566–2572.
21. Schank T. Algorithmic aspects of triangle-based network analysis. PhD Thesis, Department of Computer Science, University of Karlsruhe, 2007.
22. Le Gall F. Powers of tensors and fast matrix multiplication. In: *Proceedings of the 39th International Symposium on Symbolic and Algebraic Computation*, 2014.
23. Schank T, Wagner D. Finding, counting and listing all triangles in large graphs, an experimental study. In: *Proceedings of the 4th International Conference on Experimental and Efficient Algorithms*, WEA '05, 2005, 606–609.
24. Latapy M. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theor Comput Sci* 2008, 407:458–473.
25. Tsourakakis CE, Kang U, Miller GL, Faloutsos C. Doulion: counting triangles in massive graphs with a coin. In: *Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '09, 2009, 837–846.
26. Pagh R, Tsourakakis CE. Colorful triangle counting and a mapreduce implementation. *Inf Process Lett* 2012, 112:277–281.
27. Etemadi R, Lu J, Tsin YH. Efficient estimation of triangles in very large graphs. In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, CIKM '16, 2016, 1251–1260.
28. Schank T, Wagner D. Approximating clustering-coefficient and transitivity. *J Graph Algorithms Appl* 2005, 9:265–275.
29. Al Hasan M. Chapter 5: Methods and applications of network sampling. In: Gupta A, Capponi A, eds. *Optimization Challenges in Complex, Networked and Risky Systems*. Catonsville, MD: INFORMS; 2016, 115–139. https://doi.org/10.1287/educ.2016.0147.
30. Vitter JS. Random sampling with a reservoir. *ACM Trans Math Softw* 1985, 11:37–57.
31. Dean J, Ghemawat S. Mapreduce: simplified data processing on large clusters. In: *Proc. of the 6th conference on Operating Systems Design and Implementation – Volume 6*, 2004, 137–149.
32. Park H-M, Chung C-W. An efficient mapreduce algorithm for counting triangles in a very large graph. In: *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, CIKM '13, 2013, 539–548.
33. Arifuzzaman S, Khan M, Marathe M. Patric: a parallel algorithm for counting triangles in massive networks. In: *Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management*, CIKM '13, 2013, 529–538.
34. Kim J, Han W-S, Lee S, Park K, Yu H. Opt: a new framework for overlapped and parallel triangulation in large-scale graphs. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, 2014, 637–648.
35. Shun J, Tangwongsan K. Multicore triangle computations without tuning. In: *2015 I.E. 31st International Conference on Data Engineering*, April 2015, 149–160.
36. Bron C, Kerbosch J. Algorithm 457: finding all cliques of an undirected graph. *Commun ACM* 1973, 16:575–577. https://doi.org/10.1145/362342.362367.
37. Finocchi I, Finocchi M, Fusco EG. Clique counting in mapreduce: algorithms and experiments. *J Exp Algorithmics* 2015, 20:1.7:1–1.7:20. https://doi.org/10.1145/2794080.
38. Rahman M, Bhuiyan MA, Al Hasan M. GRAFT: an efficient graphlet counting method for large graph analysis. *IEEE Trans Knowl Data Eng* 2014, 26:2466–2478. https://doi.org/10.1109/TKDE.2013.2297929.
39. Hayes W, Sun K, Pržulj N. Graphlet-based measures are suitable for biological network comparison. *Bioinformatics* 2013, 29:483.
40. Pržulj N. Biological network comparison using graphlet degree distribution. *Bioinformatics* 2007, 23:e177.
41. Zhang L, Hong R, Gao Y, Ji R, Dai Q, Li X. Image categorization by learning a propagated graphlet path. *IEEE Trans Neural Netw Learn Syst* 2016, 27:674–685.
42. Kashima H, Saigo H, Hattori M, Tsuda K. Graph kernels for chemoinformatics. In: *Chemoinformatics and Advanced Machine Learning Perspectives: Complex Computational Methods and Collaborative Techniques*. Hershey, PA: IGI Global; 2010, 1.
43. Wernicke S, Rasche F. Fanmod: a tool for fast network motif detection. *Bioinformatics* 2006, 22:1152–1153.
44. Marcus D, Shavitt Y. Rage – a rapid graphlet enumerator for large networks. *Comput Netw* 2012, 56:810–819.
45. Pinar, A, Seshadhri C, Vishal V. Escape: efficiently counting all 5-vertex subgraphs. In: *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, 2017, 1431–1440.
46. Hočevar T, Demšar J. A combinatorial approach to graphlet counting. *Bioinformatics* 2014, 30:559–565.
47. Ahmed NK., Neville J, Rossi RA, Duffield N. Efficient graphlet counting for large networks. In: *Proceedings of the 2015 I.E. International Conference on Data*

Mining (ICDM), ICDM '15, 2015, 1–10. ISBN: 978-1-4673-9504-5.

48. Bhuiyan MA, Rahman M, Rahman M, Al Hasan M. GUISE: uniform sampling of graphlets for large graph analysis. In: *2012 I.E. 12th International Conference on Data Mining*, December, 2012, 91–100. 10.1109/ICDM.2012.87.
49. Saha TK, Al Hasan M. Fs^3: a sampling based method for top-k frequent subgraph mining. *Stat Anal Data Min* 2015, 8:245–261. https://doi.org/10.1002/sam.11277.
50. P Wang, J C S Lui, B Ribeiro, D Towsley, J Zhao, and X Guan. Efficiently estimating motif statistics of large networks. *ACM Trans Knowl Discov Data*, 9:8:1–8:27 2014. ISSN: 1556-4681.
51. Rahman M, Bhuiyan M, Al Hasan M. Graft: An approximate graphlet counting algorithm for large graph analysis. In: *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, CIKM '12, 2012, 1467–1471.
52. Jha M, Seshadhri C, Pinar A. Path sampling: a fast and provable method for estimating 4-vertex subgraph counts. In: *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, 2015, 495–505.
53. Bressan M, Chierichetti F, Kumar R, Leucci S, Panconesi A. Counting graphlets: space vs time. In: *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, WSDM '17, 2017, 557–566.

