Retrieving Top Weighted Triangles in Graphs

Raunak Kumar*
Cornell University
Ithaca, NY
raunak@cs.cornell.edu

Paul Liu*
Stanford University
Stanford, California
paul.liu@stanford.edu

Moses Charikar
Stanford University
Stanford, California
moses@cs.stanford.edu

Austin R. Benson
Cornell University
Ithaca, NY
arb@cs.cornell.edu

ABSTRACT

Pattern counting in graphs is a fundamental primitive for many network analysis tasks, and there are several methods for scaling subgraph counting to large graphs. Many real-world networks have a notion of strength of connection between nodes, which is often modeled by a weighted graph, but existing scalable algorithms for pattern mining are designed for unweighted graphs. Here, we develop deterministic and random sampling algorithms that enable the fast discovery of the 3-cliques (triangles) of largest weight, as measured by the generalized mean of the triangle's edge weights. For example, one of our proposed algorithms can find the top-1000 weighted triangles of a weighted graph with billions of edges in thirty seconds on a commodity server, which is orders of magnitude faster than existing "fast" enumeration schemes. Our methods open the door towards scalable pattern mining in weighted graphs.

CCS CONCEPTS

• **Information systems → Top-k retrieval in databases**; *Social networks*; *Recommender systems.*

KEYWORDS

weighted graphs, subgraphs, random sampling, networks

ACM Reference Format:
Raunak Kumar, Paul Liu, Moses Charikar, and Austin R. Benson. 2020. Retrieving Top Weighted Triangles in Graphs. In *The Thirteenth ACM International Conference on Web Search and Data Mining (WSDM '20), February 3–7, 2020, Houston, TX, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3336191.3371823

1 INTRODUCTION

Small subgraph patterns, also called graphlets or network motifs, are fundamental to the understanding of complex network structure [7, 34, 35]. One of the simplest non-trivial subgraph patterns is the triangle (3-clique), and the basic problem of triangle counting and enumeration has been studied extensively from theoretical and practical perspectives [4, 8, 18, 48, 51]. These developments are often driven by the desire to scale graph counting to large networks, where performing computations naively is infeasible. The focus on triangles is in part spurred by the widespread use of the pattern in graph mining applications, including community detection [9, 21, 45], network comparison [15, 33, 40], representation learning [23, 46], and generative modeling [43, 44]. In addition, triangle-based network statistics such as the clustering coefficient are used extensively in the social sciences [12, 17, 31, 60].

Nearly all of the algorithmic literature on scalable counting or enumeration of triangles focuses on *unweighted* graphs. However, many real-world network datasets have a natural notion of *weight* attached to the edges of the graph [5]. For example, edge weights can capture tie strength in social networks [59], traffic flows in transportation networks [27], or co-occurrence counts in projections of bipartite networks [62]. Such edge weights offer additional insight into the network structure. Moreover, edge weights can enrich the types of small subgraph patterns used in analysis. For instance, the network clustering coefficient has been generalized to account for edge weights [37, 38]; in these cases, a triangle is given a weight derived from the weights of its constituent edges. Roughly speaking, the a triangle's weight is typically some combination of the arithmetic mean, geometric mean, minimum, and maximum of the edge weights of the triangle. All that being said, we still lack algorithms for fast analysis of modern large-scale weighted networks, especially for weighted triangle listing and counting.

In applications of weighted triangles in this big data regime, it can often suffice to retrieve only the k triangles of largest weight for some suitable k. For example, in large online social networks, the weight of an edge could reflect how likely it is for users to communicate with each other, and top weighted triangles and cliques in this network could be used for group chat recommendations. In such a scenario, we would typically only be interested in a small number of triangles whose nodes are very likely to communicate with each other as opposed to finding *all* triangles in the graph.

Another application for finding top-weighted triangles is in prediction tasks for higher-order network interactions. The goal of "higher-order link prediction" is to predict which new groups of nodes will simultaneously interact (such as which group of scientists will co-author a paper in the future) [6]. In this setting, existing algorithms first create a weighted graph where an edge weight is the number of prior interactions that involve the two end points.

*Both authors contributed equally to this research.

WSDM '20, February 3–7, 2020, Houston, TX, USA

ACM ISBN 978-1-4503-6822-3/20/02…$15.00
https://doi.org/10.1145/3336191.3371823

After, top-weighted triangles in this graph are predicted to appear as higher-order interactions in the future (weight here is measured by a generalized mean of the triangle's edge weights). Again, it is not necessary to find all triangles since only the top predictions will be acted upon. Existing triangle enumeration algorithms do not scale to massive graphs for these problems, and we need efficient algorithms for retrieving triangles in large weighted graphs.

In this work, we address the problem of enumerating the top-weighted triangles in a weighted graph. To be precise, let $G = (V, E, w)$ be a simple, undirected graph with positive edge weights w. We define the weight of a triangle (i, j, k) with edge weights w_{ij}, w_{jk}, and w_{ik} to be the p-mean of the edge weights:

$$m_p(i, j, k) := \left[\frac{1}{3}(w_{ij}^p + w_{jk}^p + w_{ik}^p)\right]^{1/p}. \quad (1)$$

Given G, an integer parameter k, and a scaling factor p, we develop algorithms to extract the *top-k triangles* in G. We use top-k to refer to the triangles having the k largest weights, or in other words, the k heaviest triangles. Special cases of the p-mean include the arithmetic mean ($p = 1$), geometric mean ($p = 0$), and harmonic mean ($p = -1$), and in general the p-mean is a flexible measure for the weight of a triangle. Furthermore, the p-mean subsumes measures of weighted triangles from prior studies [6, 38].

At a high level, we develop two families of algorithms for finding top-weighted triangles. The first family is deterministic and optimized for extracting top-k weighted triangles for small k, typically up to a few tens of thousands (Section 3). These algorithms take advantage of the heavy-tailed edge weight distribution common in real-world networks. In the most general case, we show that under a modified configuration model, these algorithms are even "distribution-oblivious," in the sense that they can automatically compute optimal hyper-parameters to the algorithm for a wide range of input graph distributions. Additionally, the algorithmic analysis is done in a continuous sense (rather than discrete), which may be of independent interest. The second family of algorithms is randomized and aims to find many heavy triangles that are not necessarily the top-k (Section 4). We show how this family of sampling algorithms is connected to prior sampling algorithms for *counting* triangles on *unweighted* graphs [48]. Furthermore, we show that these sampling algorithms are easily parallelizable.

A carefully tuned parallel implementation of our deterministic algorithm performs well across a broad range of large weighted graphs, even outperforming the fast random sampling algorithms that are not guaranteed to enumerate all of the top-weighted triangles. A parallel implementation running on a commodity server with 64 cores can find the top 1000 weighted triangles in under 10 seconds on several graphs with hundreds of millions of weighted edges and in 30 seconds on a graph with nearly two billion weighted edges. We compare this with the off-the-shelf alternative approach, which would be an intelligent triangle enumeration algorithm that maintains a heap of the top-weighted triangles. Our proposed algorithms are orders of magnitude faster than this standard approach.

2 ADDITIONAL RELATED WORK

Due to wide applicability, there is a plethora of research on *unweighted* triangle-related algorithms. In the context of enumeration algorithms, recent attention has focused on the distributed and parallel setting [3, 14, 41, 53]. These algorithms typically employ an optimized brute force method on each machine [10, 30], with the main algorithmic challenge in deciding how to partition the data amongst the machines. Although each machine uses a brute force algorithm, early research shows that these algorithms run in time almost linear in the number of edges so long as the *degeneracy* of the graph is small [13], which has led to efficient enumeration algorithms [8, 53]. For comparison with our methods, we modify such a fast enumeration method (specifically NodeIterator++ [53]) to retain the top-k weighted triangles. Although enumeration algorithms are agnostic to edge weights, the sheer number of triangles in massive graphs renders such an approach prohibitively expensive.

When triangle *enumeration* is infeasible, algorithms focus instead on triangle *counts* or graph statistics such as clustering coefficients. Again, these statistics are in the unweighted regime, as only the number of triangles are considered. There is a progression of sampling methods depending on what kind of structures one is sampling from the graph. At a basic level, edge-based sampling methods sample an edge and counts all incident triangles on that edge. So-called wedge-based methods sample length-2 paths [48], and this concept has been generalized for counting 4-cliques [26]. Finally, tiered-sampling combines sampling of arbitrary subgraphs to count larger subgraphs (with a focus on 4-cliques and 5-cliques) [52]. Beyond enumeration and sampling, there are several other triangle-based algorithms, such as graph sparsification [19, 39, 56], spectral and matrix methods [2, 55], and approaches for computing clustering and closure coefficients [42, 47, 48, 63]. Hasan and Dave provide a deeper review on triangle counting [22].

All of the above methods are for triangles. These ideas have been extended in several ways. There are sampling methods for estimating counts of more general motifs [1, 11, 25] and motifs with temporal structure [32] as well as parallel clique enumeration methods [16]. Still, these methods do not work for weighted graphs, where subgraph patterns appear in generalizations of the clustering coefficient [36] as well as in graph decompositions [50].

3 DETERMINISTIC ALGORITHMS

We begin by developing two types of deterministic algorithms for finding the top-k weighted triangles in a graph, where the triangle weight is a generalized p-mean of its edge weights as defined in Eq. (1). A robust baseline is to use a fast triangle enumeration algorithm for unweighted graphs, compute the weight on each triangle, and pick out the top-k weighted triangles (or, to save memory, maintain a heap of the top-weighted triangles). In our tests, we use an optimized sequential version of NodeIterator++ [8], which is the basis for many parallel enumeration algorithms. We call this a "brute force" approach. Although faster, parallel versions of the brute force approach exist, our results in Section 5 show that brute force with perfect parallelism would require over 2000 cores to beat our sequential deterministic algorithm in certain cases.

The brute force approach is agnostic to the distribution of edge weights—it is the same regardless of the weights. However, we expect that triangles of large weight are formed by edges of large weight. We exploit this intuition below to develop faster algorithms.

Algorithm 1: Static heavy-light algorithm for finding top-weighted triangles.

Input: Weighted graph $G = (V, E, w)$, scaling p, number of triangles k

$H \leftarrow \{e \in E : w_e > \tau\}$

$T \leftarrow$ all triangles formed by edges in H

return *k triangles in T with largest p-mean weight*

At a high level, our main deterministic algorithm will try to dynamically partition the edges into "heavy" and "light" classes based on edge weight. Following this partition, we find triangles adjacent to the heavy edges until the top-k heaviest are identified.

A simple heavy-light algorithm. As a precursor to our dynamic algorithm, consider the following static threshold-based algorithm. Given a threshold τ, partition the edges into a "heavy" set $H = \{e \mid w_e > \tau\}$ and a "light" set $L = E \setminus H$. For a large threshold τ, we expect most edges to be in L. Thus, the subgraph induced by H is small, and we can use an enumeration algorithm to get a collection of heavy triangles (Algorithm 1). This is not guaranteed to find the heaviest triangles as edges in H might only be in triangles with edges in L. However, in practice, the heaviest triangles often have all edges in H. With no additional *asymptotic cost*, we can also use existing enumeration algorithms to find triangles with just one or two heavy edges. Unfortunately, the constant factor slow-down substantially increases running time on real-world graphs.

In practice, we find that this simple algorithm vastly outperforms brute force and can always find the top-weight triangle given a proper threshold; thus, this method is a robust baseline. Nonetheless, there are a couple of issues with this "static" heavy-light algorithm. The first is that since the algorithm relies on a static threshold τ to partition the edges into light and heavy sets, many more triangles may be enumerated than is necessary. The second is the difficulty in finding an appropriate threshold τ given no prior knowledge of the input graph. In the next section, we develop a dynamic variant of Algorithm 1 to deal with these issues.

3.1 Dynamic heavy-light algorithm

We now develop a dynamic algorithm that uses the concepts of Algorithm 1 but is significantly more efficient. Suppose we preprocess the edges $E = \{e_0, e_2, ..., e_{m-1}\}$ so that they are sorted by *decreasing* weight ($w_i \geq w_{i+1}$ where w_i denotes the weight of the i-th edge). Our dynamic heavy-light algorithm maintains a dynamic partition of the edges into three sets based on weight:

- $S = \{e_0, \ldots, e_h\}$ are the "super-heavy" edges of the h largest weights;
- $H = \{e_{h+1}, \ldots, e_l\}$ are the "heavy" edges consisting of the next $l - h$ edges of largest weights; and
- $L = \{e_{l+1}, \ldots, e_{m-1}\}$ are the remaining "light" edges that are neither heavy nor super-heavy.

As the algorithm evolves, we adjust the sets S, H, and L by changing the values of h and l. Any triangle can be broken down into a combination of super-heavy, heavy, and light edges. As a first-order approximation, we would expect the heaviest class of triangles to have three super-heavy edges, the next heaviest to have two super-heavy edges and one heavy edge, and so on down to the case of three light edges. Furthermore, by considering the edges in a specific order, we can also obtain useful bounds on the weight of the heaviest triangles we have not yet enumerated. Suppose we have enumerated all triangles containing three super-heavy edges. Then the heaviest triangle not yet enumerated must have at least one edge from H or L. This upper bounds the p-th power of the weight of that triangle to be $\frac{1}{3}(2w_0^p + w_h^p)$. Our method tries to enumerate triangles so that this bound decreases as quickly as possible.

Algorithm 2: Dynamic heavy-light algorithm for finding the top-k weighted triangles.

Input: Weighted graph $G = (V, E, w)$, scaling p, number of triangles k, parameter α_p.

1 Sort E in decreasing order of weight
2 Initialize threshold $\tau = \infty$, triangle set $T = \emptyset$
3 Initialize partitions $S = H = \emptyset$, $L = E$
4 Initialize edge pointers $h = l = -1$
`// We take the convention that` $e_{-1} = \infty$.
5 **while** *there are $< k$ triangles above weight τ in T* **do**
6 **if** $w_{l+1} > w_{h+1}^{\alpha_p}$ **then**
7 Move e_{l+1} from L to H.
8 Y = triangles formed by e_{l+1} and 2 edges from $S \cup H$
9 Z = triangles formed by e_{l+1}, 1 edge from L, and 1 edge from $S \cup H$
10 $T = T \cup (Y \cup Z)$, $\quad$ l = l + 1
11 **else**
12 Move e_{h+1} from H to S.
13 Y = triangles formed by e_h and 2 edges from L
14 $T = T \cup Y$, $\quad$ h = h + 1
15 **end**
16 Update threshold $\tau = w_h^p + 2w_l^p$
17 **end**
18 **return** *k triangles in T with largest p-mean weight*

Algorithm 2 outlines our procedure. Each step of the algorithm consists of two steps: (i) update the partition by moving an edge to a heavier class; and (ii) enumerate triangles whose edges come from certain classes. At the end of each step, we maintain the invariant that we have enumerated all triangles with at least one super-heavy edge or at least two heavy edges. This invariant allows us to obtain a bound τ on the heaviest triangle we have not yet enumerated. The constant α_p of Algorithm 2 determines how edges get promoted to heavier classes; this parameter is used in our analysis to optimize the expected decrease in the threshold τ. We will specify this constant later in our analysis. When the algorithm begins, the partitions are initialized with $S = H = \emptyset$, $L = \{e_0, \ldots, e_{m-1}\}$, and $\tau = \infty$. The algorithm runs until it enumerates k triangles above a dynamically decreasing threshold $\tau = w_h^p + 2w_l^p$ (Line 16 of Algorithm 2).

Let τ^* be the weight of the k-th heaviest triangle. As soon as $\tau \leq \tau^*$, we will have enumerated all of the top-k triangles. This algorithm solves both issues of our static heavy-light algorithm. If the threshold τ hits τ^* exactly, we only enumerate around k triangles. As τ is computed on the fly, there is no need to choose the threshold at which we partition the edges. In the following sections, we show the algorithm's correctness and derive the optimal parameter value of α_p, or show how an optimal α_p can be implicitly computed.

3.2 Algorithm correctness

We claim that at the end of each iteration of the while loop in Algorithm 2, T contains the top-$|T|$ weighted triangles in the graph.

We first bound the heaviest possible triangle not yet enumerated. Observe that when an edge moves from L to H (Line 6), all triangles including that edge and at least one edge from $S \cup H$ are enumerated and when an edge moves from H to S (Line 11) all triangles including that edge are enumerated. Thus, the if-else clause maintains the invariant that all triangles with at least one edge from S or at least two edges from H are enumerated. Now consider any triangle with weight at least $w_h^p + 2w_l^p$. By case analysis, there must either exist two edges with weight at least w_l, or one edge with weight at least w_h. This means that either two edges are from H, or one edge is from S. In either case, our invariant ensures that the triangle must have been enumerated. Similar reasoning shows a tight threshold is $w_{h+1}^p + w_{l+1}^p + w_{l+2}^p$, as $e_{h+1}, e_{l+1}, e_{l+2}$ is potentially an unenumerated triangle. However, this sum is at most $w_h^p + 2w_l^p$ due to the monotonicity of the edge weights.

Since τ is monotonically decreasing, this implies that at the end of each iteration of the while loop in Algorithm 2, T contains the top-$|T|$ weighted triangles in the graph (there may be triangles not enumerated with *equal* weight to one of the triangles in T). Therefore, as a corollary of this claim, Algorithm 2 correctly returns the top-k triangles, provided the graph has at least k triangles. Note that correctness is independent of the parameter α_p. We next analyze how we might set this parameter in an optimal way.

3.3 Optimizing parameter settings

We now derive principled ways for setting the parameter α_p in Algorithm 2. Although the algorithm is discrete, we present a simple analysis using continuous differentials. Let $w_h(t)$ and $w_l(t)$ be the weight of edges e_h and e_l at time t respectively (one can think of t as a continuous counter for the while loop iterations). At time t, the threshold is $\tau(t) = w_h(t)^p + 2w_l(t)^p$. As a proxy to maximizing the enumeration rate, we maximize the rate at which the threshold decreases. To do this, we maximize the derivative $\mathrm{d}\tau/\mathrm{d}t$ by adjusting $w_h(t)$ and $w_l(t)$ based on the input parameter α_p.

The derivatives $\mathrm{d}w_h/\mathrm{d}t$ and $\mathrm{d}w_l/\mathrm{d}t$ approximate the maximum change in w_h or w_l "per unit of computation time." In each iteration of the while loop, we can choose to spend time decreasing w_h or w_l. Thus, a rough approximation to the derivatives is the ratio of the change in weight (by incrementing either the h or l pointer) to the computational cost of changing the corresponding pointer.

Let $w_{<h} := \max\{w : w < w_h\}$ and let $CDF(w)$ and $PDF(w)$ be the cumulative and probability density functions of the edge weight distribution. If we move the edge pointer e_h, the average change in w_h is the ratio of $(w_h - w_{<h})$ to the number of edges of weight w_h. The number of edges of weight w_h is proportional to $CDF(w_h) - CDF(w_{<h})$, so the average change in w_h is approximately

$$\frac{w_h - w_{<h}}{CDF(w_h) - CDF(w_{<h})} \approx \frac{1}{PDF(w_h)}. \tag{2}$$

Similarly, the change in w_l is approximately $1/\mathrm{PDF}(w_l)$.

Analysis for power-law distributed weights. At this point, we are free to continue our analysis with any model for for the distribution of weights on G. One important analyzable case is a power law distribution on the weights, and this type of distribution is a reasonable model for several of our datasets (Fig. 1). Thus in this section, we carry out the analysis assuming that the edge weights follow a power law distribution with parameter β.

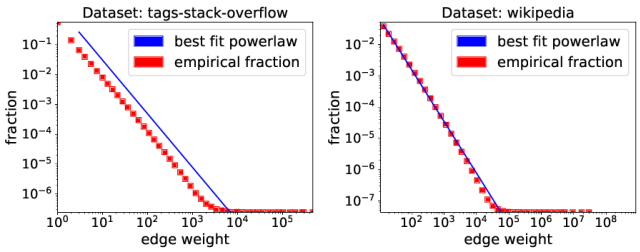

Figure 1: Edge weight distribution in two datasets (see Section 5.1 for a description of the datasets). These plots suggest that a power law distribution on edge weights is a reasonable assumption. With this, we have a simple condition to choose which pointer to move in the dynamic heavy-light algorithm (Line 6 of Algorithm 2).

Formally, let X be a random variable. We say X follows a power law distribution with parameter $\beta > 1$ and some constant $a > 0$ if $P(X \geq x) \sim ax^{1-\beta}$ for large x. Thus, the probability that a random edge weight is greater than or equal to w is $O(w^{1-\beta})$ and this implies that the probability that a random edge weight is equal to w is $O(w^{-\beta})$ for large w. Using this assumption we can write the change in w_h and w_l as $O(w_h^\beta)$ and $O(w_l^\beta)$ respectively.

Now we analyze the computational cost of changing e_h and e_l. To do this we impose a simple configuration model on the way that G is generated [20]. We assume that each vertex v draws its degree d from a univariate degree distribution $\mathcal{D}$ with the sum of degrees being even. We assume that the graph is generated from the following random process: (1) each vertex v starts out d_v stubs. While there are stubs available, two random stubs are drawn from the set of all stubs, and the vertices corresponding to those stubs are connected. Furthermore, upon connection a random edge weight drawn from the edge weight distribution is assigned to the edge. At the end, all self-loops in the graph are discarded. Note that these assumptions are quite strong, however we find that even this simple analysis yields good estimates for optimal values of α_p in practice (see Section 5). We now analyze the expected cost to increment the h or l pointer. Let $G_H := G[S \cup H]$ and $G_L := G[L]$, and let $\bar{d}_U$ denote the average degree in a graph U. With appropriate data structures for checking the existence of edges in G_H, the cost of incrementing h is bounded by the degree sum of the endpoints of e_h in G_L, which is on average $O(\bar{d}_{G_L})$. Assuming that G_L has approximately as many edges as G (valid in the case of small k), $\bar{d}_{G_L} \approx \bar{d}_G$. Thus the computational cost of moving e_h is approximately $O(\bar{d}_G)$.

Similarly, the cost of moving e_l is bounded by the degree sum of the endpoints of e_l in G_H, which is on average $O\left(\bar{d}_{G_H}\right)$. Since the number of edges in G_H is exactly $|S \cup H|$, the assumptions on the weight distribution say that $\bar{d}_{G_H} = O(CDF(w_l)\bar{d}_G)$. Thus, the cost of moving e_l is approximately $O(w_l^{1-\beta}\bar{d}_G)$. Combining this with Eq. (2), we obtain the following expressions for the derivatives

$$\mathrm{d}w_h/\mathrm{d}t = O(w_h^\beta/\bar{d}_G), \quad \mathrm{d}w_l/\mathrm{d}t = O(w_l^{2\beta-1}/\bar{d}_G). \tag{3}$$

Since w_l and w_h are decreasing, both derivatives are monotonically decreasing as the algorithm progresses. This property means

that greedily choosing the pointer to increment is optimal. The threshold decrease rate is $\mathrm{d}\tau/\mathrm{d}t = pw_h^{p-1}\mathrm{d}w_h/\mathrm{d}t + 2pw_l^{p-1}\mathrm{d}w_l/\mathrm{d}t$. Since at each iteration of the algorithm we can only choose to change one of e_h or e_l, we should greedily change the pointer that gives the most "bang per buck", i.e., choose e_h and e_l such that

$$w_h^{p-1}\mathrm{d}w_h/\mathrm{d}t = 2w_l^{p-1}\mathrm{d}w_l/\mathrm{d}t \implies w_h \sim O\left(w_l^{2-\frac{p}{p-1+\beta}}\right). \quad (4)$$

In other words, we should maintain the edge pointers e_h and e_l such that the weights are separated geometrically by $\alpha_p = 2 - \frac{p}{p-1+\beta}$.

Distribution-oblivious dynamic heavy-light algorithm. The analysis in the previous section yields a fast algorithm given a known prior on the power law parameter of the weight distribution. In many applications, this can be easily and robustly estimated. In this section, we present a method for which the parameter α_p can be implicitly estimated on the fly. This does not change the correctness of the algorithm but can change the running time in practice.

Although we assumed a power-law distribution on the edge weights, our analysis is actually much more general than that. As long as the derivatives $\mathrm{d}w_h/\mathrm{d}t$ and $\mathrm{d}w_l/\mathrm{d}t$ are monotonic, our greedy method of incrementing the pointers will be optimal. For the derivatives to be monotonically decreasing, the only requirement is that the *PDF* of the weight distribution is monotonically increasing as the weight decreases. This includes a wide family of distributions such as power laws and uniform distributions.

Furthermore, the analysis we used to derive α_p can also be used to compute α_p implicitly. By maintaining an estimate of the derivatives $\mathrm{d}w_{e_h}/\mathrm{d}t$ and $\mathrm{d}w_{e_l}/\mathrm{d}t$ as the algorithm runs, we can compute all the derivatives used in the analysis on the fly and greedily change the pointer with higher value of $w^{p-1}\mathrm{d}w/\mathrm{d}t$ (Eq. (4)).

Following the analysis in the previous section, the change in weight for e_h is estimated by the ratio of $w_h - w_{<h}$ and the number of edges that have weight w_h, and similarly for e_l. The computational cost of changing e_l can be estimated by the sum of the degrees of the endpoints of e_l in G_H, and similarly for e_h with G_L. Consequently, we can obtain a "distribution-oblivious" algorithm that works on a family of monotone distributions.

In our experiments, we find that this automatic way of implicitly computing α_p is successful, although in practice noise in the derivative estimates may cause this algorithm to be slower than using a fixed value of α_p. We find that setting $\alpha_p = 1.25$ works well.

4 RANDOM SAMPLING ALGORITHMS

In this section, we develop random sampling algorithms designed to sample a large collection of triangles with large weight. More formally, given a generalized p-mean as a weight function, these algorithms all sample triangles exactly proportional to their weight. The main difference between the algorithms is how efficiently they can generate samples.

We specifically generalize two types of sampling schemes that have been used to estimate triangle counts in unweighted graphs. The first scheme is based on *edge sampling* [28, 39, 57], where we first sample one edge and then enumerate triangles adjacent to the sampled edge. The second method uses ideas from *wedge sampling* [48], where we sample two adjacent edges and check whether these two edges induce a triangle. Although these ideas were designed for triangle counting in unweighted graphs, we show how they can be adapted and extended to design algorithms that sample highly weighted triangles. The major benefits of these algorithms are that they are simple to implement and also easy parallelize, since samples can be trivially generated in parallel.

Algorithm 3: Weighted edge sampling (ES) algorithm

Input: Weighted graph $G = (V, E, w)$, scaling p, number of iterations t, number of triangles k

```
Initialize triangle set T ← ∅
for iteration 1, ..., t do
    Sample edge (a, b) ∝ w_ab^p
    for each neighbor c ∈ N(a) ∩ N(b) do
        T ← T ∪ {(a, b, c)}
    end
end
return k triangles in T with largest p-mean weight
```

Throughout this section, we assume that our weighting function for a triangle is any generalized p-mean as given in Eq. (1). Since the weight ordering of triangles is independent of the scaling by $1/3$ and the exponent $1/p$, we can consider the more simple function:

$$w_p(a, b, c) = w_{ab}^p + w_{bc}^p + w_{ac}^p. \quad (5)$$

For a given vertex $a \in V$, we will use $N(a)$ to denote the set of neighbors of a: $N(a) = \{b \in V \mid (a, b) \in E\}$, and d_a to be the (unweighted) degree of node a (i.e., $d_a = |N(a)|$).

4.1 Weighted edge sampling

We first discuss an edge sampling (ES) algorithm (Algorithm 3). The algorithm is based on a simple two-step procedure. First, we sample a single edge according to the following distribution:

$$\Pr(\text{sample edge } (a, b)) := w_{ab}^p/Z, \quad Z = \textstyle\sum_{(u,v)\in E} w_{uv}^p.$$

Second, after we sample an edge (a, b), we enumerate all triangles (a, b, c) incident to (a, b). These two steps are repeated several times.

The above procedure has a few issues. First, it takes $O(d_a + d_b)$ time to find triangles adjacent to an edge (a, b), which can be expensive in graphs where high-degree nodes are connected; we get around this in our next sampling scheme. Second, there is no guarantee that the above procedure will generate at least k unique triangles. Moreover, even if the algorithm samples a sufficient number of triangles, it is not necessarily the case that these are the *top-weighted* triangles. The latter issue is an inherent limitation of random sampling schemes in general. All that being said, the procedure has the nice property of being biased in terms of sampling triangles with high weight, formalized as follows.

Proposition 4.1. *The probability that a triangle (a, b, c) is enumerated in a given iteration of Algorithm 3 is $w_p(a, b, c)/Z$, where $Z = \sum_{e\in E} w_e^p$.*

Proof. The probability that any edge (u, v) is sampled initially is $w_{u,v}^p/Z$. Triangle (a, b, c) is enumerated if any one of if edge (a, b), (b, c), or (a, c) is this sampled edge. □

While ES is simple to describe, making the algorithm fast in practice requires careful implementation. First, a natural way of

Algorithm 4: Weighted wedge sampling (WS) algorithm

Input: Weighted graph $G = (V, E, w)$, scaling p, number of iterations t, number of triangles k

Initialize triangle set $T \leftarrow \emptyset$
for *iteration* $1, \ldots, t$ **do**
  Sample node a with probability as in Eq. (6)
  Sample $b \in N(a)$ with probability as in Eq. (7)
  Sample $c \in N(a)$ with probability as in Eq. (8)
  if *nodes* a, b, c *form a triangle* **then**
    $T \leftarrow T \cup \{(a, b, c)\}$
end
return k *triangles in* T *with largest* p*-mean weight*

sampling an edge is to simply pick one at random with probability proportional to its weight, but this is slow because there are a large number of edges. However, there is typically a much smaller number of *unique* edge weights. Thus, we first first sample an edge weight and then sample an edge with this weight. Pre-processing of sampling probabilities for this approach involves iterating over the edges and computing two quantities—a cumulative edge weight (in order to sample an edge weight) and a map of edge weight to edges (in order to sample an edge given an edge weight). This pre-processing step can take much longer than the sampling loop if implemented naively. In a sorted list of edges, all edges that share the same edge weight lie in a contiguous chunk, and this significantly speeds up the process of computing the above quantities.

4.2 Weighted wedge sampling

One of the issues with the simple edge sampling scheme described above is that we have to look over the neighbors of the end points of the sampled edge in order to find triangles. This can be expensive if the degrees of these nodes are large. An alternative approach is to sample adjacent edges, called *wedges*, with large weight and then check if each wedge induces a triangle. This scheme is called *wedge sampling* (WS) and has been used as a mechanism for estimating the total number of triangles in an unweighted graph [48, 58].

Algorithm 4 outlines the overall sampling scheme. Each iteration of the algorithm has three steps. First, we sample a node with a bias towards nodes participating in heavily weighted edges. Specifically, let $D(a) = \sum_{b \in N(a)} w_{ab}^p$ denote the sum of edge weights incident to a. We sample node a according to the following distribution:

$$\Pr(\text{sample } a) = \tilde{W}_1(a)/Z_1 := 2 \cdot d_a \cdot D(a)/Z_1, \tag{6}$$

where Z_1 is a normalizing constant. Next, we sample a neighbor of node a, again with a bias towards nodes that participate in heavily weighted edges. The specific distribution is

$$\begin{aligned} \Pr(\text{sample } b \in N(a) \mid a) &= \tilde{W}_2(b \mid a)/Z_2 \\ &:= (d_a \cdot w_{ab}^p + D(a))/Z_2, \end{aligned} \tag{7}$$

where Z_2 is a normalizing constant. We have now produced a single edge and want to produce an adjacent edge. We do this by sampling another neighbor of a, this time with probability

$$\begin{aligned} \Pr(\text{sample } c \in N(a) \mid a, b) &= \tilde{W}_3(c \mid a, b)/Z_3 \\ &:= (w_{ac}^p + w_{ab}^p)/Z_3, \end{aligned} \tag{8}$$

where Z_3 is again a normalizing constant. If the sampled wedge $\{(a, b), (a, c)\}$ induces a triangle, then we add it to our collection.

Similar to the unweighted scheme of Seshadhri et al. [48], we have the following result.

Proposition 4.2. *A given iteration of Algorithm 4 samples triangle* (a, b, c) *with probability* $w_p(a, b, c)/\tilde{Z}$, *where* $\tilde{Z} = \sum_{v \in V} d_v \cdot D(v)$.

Proof. The normalizing constants in Eqs. (6) to (8) are $Z_1 = 2\sum_{v \in V} d_v \cdot D(v)$, $Z_2 = \tilde{W}_1(a)$, and $Z_3 = \tilde{W}_2(b \mid a)$ Thus, the probability of sampling wedge (a, b, c) centered on a is equal to

$$\frac{\tilde{W}_1(a)}{Z_1} \cdot \frac{\tilde{W}_2(b \mid a)}{Z_2} \cdot \frac{\tilde{W}_3}{Z_3} = \frac{\tilde{W}_3}{Z_1} = \frac{w_{ab}^p + w_{ac}^p}{Z_1}.$$

Thus, the probability of sampling any of the three wedges consisting of nodes a, b and c is equal to

$$(w_{ab}^p + w_{ac}^p)/Z_1 + (w_{ac}^p + w_{bc}^p)/Z_1 + (w_{bc}^p + w_{ab}^p)/Z_1.$$

Plugging in the expression for Z_1 shows that this probability is equal to $w_p(a, b, c)/\tilde{Z}$. Since triangle (a, b, c) is sampled if any of its three wedges is sampled, this yields the desired result. □

4.3 Number of samples

Propositions 4.1 and 4.2 says that Algorithms 3 and 4 tend to sample edges with large weight, but this does not guarantee that the top weighted triangles are enumerated. Standard probabilistic analysis can give us some sense on how many iterations we need for a given triangle. Specifically, if $q = w_p(a, b, c)/Z$ is the probability of sampling a triangle (a, b, c), then for any $\delta \in (0, 1)$, $s \geq \log(1/\delta)/q$ samples guarantees that (a, b, c) is enumerated with probability at least $1 - \delta$. To see this, let $r = 1 - (1 - q)^s$ denote the probability that triangle (a, b, c) is sampled at least once in s samples. Using the fact that $1 - x \leq \exp(-x)$, we get that $r \geq 1 - \exp(-sq) \geq 1 - \delta$.

This analysis says that the normalizing constant for wedge sampling drives up the number of samples required to see top weighted triangles with high probability, as compared to edge sampling. However, obtaining samples in Algorithm 3 can be more costly as we might have to find common neighbors of large-degree nodes. It is not immediately clear which algorithm is better, but our results in the next section show that edge sampling is superior in practice.

5 NUMERICAL EXPERIMENTS

We now report the results of several experiments measuring the performance of our deterministic and random sampling algorithms compared to competitive baselines such as the static heavy-light thresholding method from Algorithm 1.[1] We find that edge sampling (ES) works much better than wedge sampling (WS), but our deterministic heavy light algorithm is even faster across a wide range of datasets, outperforming the baselines by orders of magnitude in terms of running time.

5.1 Data

We used a number of datasets in order to test the performance of our algorithms.[2] Table 1 lists summary statistics of the datasets and we describe them briefly below.

[1] Implementations of our algorithms and code to reproduce results are available at https://github.com/raunakkmr/Retrieving-top-weighted-triangles-in-graphs.
[2] Data available at http://www.cs.cornell.edu/~arb/data/index.html.

Table 1: Summary statistics of datasets.

dataset	# nodes	# edges	edge weight	
			mean	max
tags-stack-overflow	50K	4.2M	13	469
threads-stack-overflow	2.3M	21M	1.1	546
Wikipedia-clickstream	4.4M	23M	347	817M
Ethereum	38M	103M	2.8	1.9M
AMiner	93M	324M	1.3	13K
reddit-reply	8.4M	435M	1.5	165K
MAG	173M	545M	1.7	38K
Spotify	3.6M	1.9B	8.6	2.8M

tags-stack-overflow [6]. On Stack Overflow, users ask, answer, and discuss computer programming questions, and users annotate questions with 1–5 tags. The nodes in this graph correspond to tags, are the weight between two nodes is the number of questions jointly annotated by the two tags.

threads-stack-overflow [6]. This graph is constructed from a dataset of user co-participation on Stack Overflow question threads. Nodes correspond to users and the weight of an edge is the number of times that two users appeared in one of these threads.

Wikipedia-clickstream [61]. This graph is derived from Wikipedia clickstream data from request logs in January, 2017 that capture how users transition between articles (only transitions appearing at least 11 times were recorded). The nodes of the graph correspond to articles and the weight between two nodes is the number of times users transitioned between the two pages.

Ethereum. Ethereum is a blockchain-based computing platform for decentralized applications. Transactions update state in the Ethereum network, and each transaction has a sender and a receiver address. We create a graph where the nodes are addresses and the weight between two nodes is the number of transactions between the two addresses, using all transactions on the platform up to August 17, 2018, as provided by `blockchair.com`.

AMiner and MAG [49, 54]. These graphs are constructed by two large bibliographic databases—AMiner and the Microsoft Academic Graph. We construct weighted co-authorship graphs, where nodes represent authors and the weight between two nodes is the number of papers they have co-authored. Papers with more than 25 authors were omitted from the graph construction.

reddit-reply [24, 32]. Users on the social media web site `reddit.com` interact by commenting on each other's posts. We derive a graph from a collection of user comments. Nodes are users and the weight of an edge is the number of interactions between the two users.

Spotify. As part of a machine learning challenge, the music streaming platform Spotify released a large number of user "listening sessions," each consisting of a set of songs. We constructed a weighted graph where the nodes represent songs and the weight of an edge is the number of times that the songs co-appeared in a session.

5.2 Algorithm benchmarking

We evaluate the performance of our proposed algorithms: (i) random edge sampling (ES) as in Algorithm 3; (ii) random wedge sampling (WS) as in Algorithm 4; (iii) the static heavy-light (SHL) scheme as in Algorithm 1 (see below for how we set the thresholds); (iv) the dynamic heavy-light scheme (DHL) as in Algorithm 2; (v) auto heavy-light (auto-HL), which is the distribution-obvlivious adaptation of DHL that automatically adjusts edge promotion to optimize the decrease in the threshold. As a baseline, we use an optimized sequential version of NodeIterator++ [8, 13, 53], which we call "brute force" (BF). This algorithm iterates over vertices with decreasing degree, and for each vertex, enumerates triangles that are formed by neighboring vertices with lower degree. All methods were implemented in C++, and all experiments were executed on a 64 core 2.20 GHz Intel Xeon CPU with 200 GB of RAM. We used parallel sorting for all algorithms and parallel sampling for the random sampling algorithms; other parts of the algorithms were executed sequentially. We evaluated the algorithms for two values of k: 1,000 and 100,000, using the arithmetic mean ($p = 1$ in Eq. (1)).

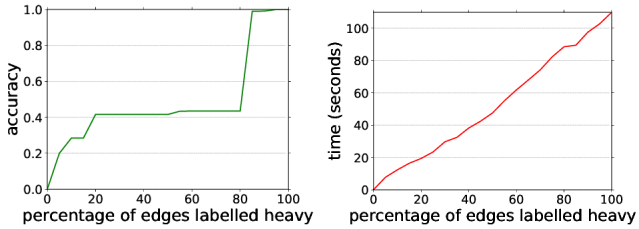

Figure 2: Accuracy and running time as a function of edges labeled "heavy" by the thresholding for the static heavy-light algorithm (Algorithm 1) on the Ethereum dataset for k = 1,000. As the threshold decreases, a larger percentage of edges are labelled heavy. This increases the accuracy but also increases the running time. For reasonable accuracy levels, we find that the running time is slower than our optimized dynamic heavy-light algorithm (Algorithm 2), which achieves 100% accuracy (see Table 2).

Recall that DHL (Algorithm 2) uses a power law distribution model of the edge weights and sets a parameter α_p based on the power law exponent. We fix $\alpha_p = 1.25$ for our experiments, which works well across a range of datasets.

The random sampling algorithms are not guaranteed to enumerate all of the top-k weighted triangles. Instead, we measure the performance of these algorithms in terms of running time and accuracy (the fraction of top-k triangles actually enumerated). We ran ES long enough to achieve at least 94% (k = 1,000) or 50% (k = 100,000) accuracy on all datasets. We also ran WS long enough for it to achieve at least 50% accuracy (for both values of k). However, in practice, its performance is poor, and we terminate the algorithm if it takes longer than BF to achieve this accuracy level.

Similarly, the static heavy-light algorithm (Algorithm 1) is not guaranteed to achieve 100% accuracy since it relies on a fixed threshold to partition the edges as heavy and light and only enumerates triangles formed by heavy edges. As the threshold decreases, a larger number of edges are labelled heavy. This increases the accuracy but also slows down the algorithm. Figure 2 illustrates this trade-off on the Ethereum dataset, and a similar trend is observed on the other datasets. In our experiments, we labelled the top 10% of edges as heavy and report the accuracy. As we discuss below, SHL is slower and attains sub-100% accuracy in practice; improving the accuracy would only make this baseline slower.

Table 2 lists the running times of the algorithms. BF did not terminate on Spotify after 24 hours, so running times for this baseline are not available on that dataset. We highlight a few important findings. First, our deterministic methods DHL and Auto-HL excel

Table 2: Running times of all of our algorithms in seconds averaged over 10 runs. BF is brute force enumeration of triangles, which is the out-of-the-box baseline; ES is the parallel edge sampling algorithm (Algorithm 3); WS is the parallel wedge sampling algorithm (Algorithm 4); DHL is the dynamic heavy-light deterministic algorithm (Algorithm 2); Auto-HL is the distribution oblivious modification of the dynamic heavy-light discussed in Section 3.3; and SHL is the static heavy-light threshold deterministic algorithm (Algorithm 1). ES was run to achieve 94% (k = 1,000) or 50% (k = 100,000) accuracy, while WS was run to achieve just 50% accuracy and was stopped early if taking longer than BF (or longer than SHL on the Spotify dataset). SHL is an approximation, and we report its accuracy in the final column. Overall, our deterministic algorithms (DHL or Auto-HL) are fast and achieve 100% accuracy. ES is slightly faster in some cases, but it is approximate.

k	dataset	BF	ES	WS	DHL	Auto-HL	SHL	Accuracy (SHL)
1000	tags-stack-overflow	12.71	0.57	11.28	**.08**	0.09	0.54	0.99
	threads-stack-overflow	34.92	1.31	>34.92	0.53	**0.38**	1.55	0.55
	Wikipedia-clickstream	16.32	14.31	>16.32	**5.44**	7.26	2.02	0.87
	Ethereum	52.91	9.03	>52.91	8.12	**6.94**	11.90	0.28
	Aminer	243.75	**3.72**	>243.75	13.35	12.36	43.47	0.32
	reddit-reply	4047.62	5.19	341.17	5.02	**4.74**	102.65	0.51
	MAG	512.24	**4.92**	48.58	29.19	20.89	72.49	0.91
	Spotify	>86400	60.33	>5300	31.82	**30.79**	5388.45	1.00
100000	tags-stack-overflow	13.06	0.58	>13.06	**0.23**	**0.23**	0.62	0.28
	threads-stack-overflow	33.99	**1.19**	>33.99	1.82	1.73	1.63	0.32
	Wikipedia-clickstream	17.34	13.64	>17.34	**5.49**	7.24	2.15	0.13
	Ethereum	57.35	10.03	>57.35	**18.11**	19.87	11.70	0.11
	Aminer	245.28	3.45	>245.28	15.38	**13.91**	43.28	0.24
	reddit-reply	3857.57	**6.52**	>3857.57	6.87	7.49	98.34	0.34
	MAG	524.80	**4.25**	>524.80	29.52	21.37	75.97	0.10
	Spotify	>86400	47.27	>5300	30.57	**29.89**	5384.17	0.92

at retrieving the top-k triangles. They achieve perfect accuracy and are orders of magnitude faster than BF. For instance, these algorithms get a 1000x speedup on reddit-reply (k = 1,000) and more than a 2000x speedup on Spotify (k = 100,000). These algorithms also outperform SHL by a significant margin in both running time and accuracy. For example, despite being 30x slower on reddit-reply, SHL only achieves 50% accuracy (k = 1,000). Again, our deterministic algorithms *always achieve 100% accuracy*, and do so in a fraction of the time taken by the baseline methods BF and SHL.

ES performs much better than WS. WS struggles to achieve high accuracy and is not competitive with the BF baseline or SHL. On the other hand, ES is quite competitive with even DHL and Auto-HL. ES retrieves the top 1,000 triangles on the AMiner and MAG datasets with 99% accuracy at speedups of 2x or 4x over DHL and Auto-HL. A similar speedup is observed for k = 100,000, but ES only achieves 50% accuracy in these cases. Even though ES works well in these cases, our deterministic algorithms are still competitive. We conclude that intelligent deterministic approaches work extremely well for finding top weighted triangles in large weighted graphs.

All of our algorithms except BF and WS sort edges by weight in a pre-procesing step. Surprisingly, this pre-processing step is a computational bottleneck, and parallel sorting is crucial to achieving high performance. In turn, this negates the possible benefit of parallel sampling for the randomized algorithms over our deterministic methods, whose main routines are inherently sequential.

6 DISCUSSION

Subgraph patterns, and triangles in particular, are used extensively in graph mining applications. However, most of the existing literature focuses on counting or enumeration tasks in unweighted graphs. In this paper, we developed deterministic and random sampling algorithms for finding the heaviest triangles in large weighted graphs. With some tuning, our main deterministic algorithm can find these triangles in a few seconds on graphs with hundreds of millions of edges or in 30 seconds on a graph with billions of edges. This is orders of magnitude faster than what one could achieve with existing fast enumeration schemes and is usually much faster than even our randomized sampling algorithms.

We anticipate that our work will enable scientists to better explore large-scale weighted graphs and can also spur new algorithmic developments on subgraph listing and counting in weighted graphs. For example, an interesting avenue for future research would be the development of random sampling algorithms that sample triangles with probability proportional to some arbitrary function of their weight, chosen to converge to the top weighted triangles faster. This could make random sampling approaches competitive with our fast deterministic methods. The edge sampling method can also be generalized to k-clique sampling by sampling an edge and then enumerating adjacent k-cliques, and we provide the details in an extended version of this paper [29]. How to extend the deterministic algorithms to top k-clique enumeration is less clear, so sampling may be more appropriate for larger clique patterns.

Acknowledgements. This research was supported by NSF Awards DMS-1830274 and CCF-1617577; ARO Award W911NF19-1-0057;

ARO MURI; Simons Investigator Award, Google Faculty Research Award; Amazon Research Award; and Google Cloud resources.

REFERENCES

[1] Nesreen K. Ahmed, Nick Duffield, Jennifer Neville, and Ramana Kompella. 2014. Graph sample and hold: a framework for big-graph analytics. In *KDD*.

[2] Noga Alon, Raphael Yuster, and Uri Zwick. 1997. Finding and Counting Given Length Cycles. *Algorithmica* 17, 3 (1997), 209–223.

[3] Shaikh Arifuzzaman, Maleq Khan, and Madhav V. Marathe. 2013. PATRIC: a parallel algorithm for counting triangles in massive networks. In *ICDM*. 529–538.

[4] Haim Avron. 2010. Counting triangles in large graphs using randomized matrix trace estimation. In *Workshop on Large-scale Data Mining: Theory and Applications*.

[5] Alain Barrat, Marc Barthelemy, Romualdo Pastor-Satorras, and Alessandro Vespignani. 2004. The architecture of complex weighted networks. *Proceedings of the national academy of sciences* 101, 11 (2004), 3747–3752.

[6] Austin R. Benson, Rediet Abebe, Michael T. Schaub, Ali Jadbabaie, and Jon Kleinberg. 2018. Simplicial closure and higher-order link prediction. *Proceedings of the National Academy of Sciences* 115, 48 (2018), E11221–E11230.

[7] Austin R. Benson, David F. Gleich, and Jure Leskovec. 2016. Higher-order organization of complex networks. *Science* 353, 6295 (2016), 163–166.

[8] Jonathan W. Berry, Luke A. Fostvedt, Daniel J. Nordman, Cynthia A. Phillips, C. Seshadhri, and Alyson G. Wilson. 2015. Why Do Simple Algorithms for Triangle Enumeration Work in the Real World? *Internet Mathematics* 11, 6 (2015), 555–571.

[9] Jonathan W Berry, Bruce Hendrickson, Randall A LaViolette, and Cynthia A Phillips. 2011. Tolerating the community detection resolution limit with edge weighting. *Physical Review E* 83, 5 (2011), 056119.

[10] Jonathan W Berry, Daniel J Nordman, Cynthia A Phillips, and Alyson G Wilson. 2010. *Listing triangles in expected linear time on a class of power law graphs*. Technical Report. Technical report, Sandia National Laboratories.

[11] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. 2017. Counting Graphlets: Space vs Time. In *WSDM*.

[12] Ronald S Burt. 2007. Secondhand brokerage: Evidence on the importance of local structure for managers, bankers, and analysts. *AMJ* (2007).

[13] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and Subgraph Listing Algorithms. *SIAM J. Comput.* 14, 1 (1985), 210–223.

[14] Shumo Chu and James Cheng. 2011. Triangle listing in massive networks and its applications. In *KDD*. 672–680.

[15] Noshir S Contractor, Stanley Wasserman, and Katherine Faust. 2006. Testing multitheoretical, multilevel hypotheses about organizational networks: An analytic framework and empirical example. *Academy of Management Review* (2006).

[16] Maximilien Danisch, Oana Denisa Balalau, and Mauro Sozio. 2018. Listing k-cliques in Sparse Real-World Graphs. In *WWW*. 589–598.

[17] Nurcan Durak, Ali Pinar, Tamara G Kolda, and C Seshadhri. 2012. Degree relations of triangles in real-world networks and graph models. In *CIKM*. 1712–1716.

[18] Talya Eden, Amit Levi, Dana Ron, and C. Seshadhri. 2017. Approximately Counting Triangles in Sublinear Time. *SIAM J. Comput.* 46, 5 (jan 2017), 1603–1646.

[19] Roohollah Etemadi, Jianguo Lu, and Yung H. Tsin. 2016. Efficient Estimation of Triangles in Very Large Graphs. In *CIKM*. 1251–1260.

[20] Bailey K. Fosdick, Daniel B. Larremore, Joel Nishimura, and Johan Ugander. 2018. Configuring Random Graph Models with Fixed Degree Sequences. *SIAM Rev.* (2018).

[21] David F Gleich and C Seshadhri. 2012. Vertex neighborhoods, low conductance cuts, and good seeds for local community methods. In *KDD*. 597–605.

[22] Mohammad Al Hasan and Vachik S. Dave. 2018. Triangle counting in large networks: a review. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* 8, 2 (2018).

[23] Keith Henderson, Brian Gallagher, Tina Eliassi-Rad, Hanghang Tong, Sugato Basu, Leman Akoglu, Danai Koutra, Christos Faloutsos, and Lei Li. 2012. RolX: structural role extraction & mining in large graphs. In *KDD*.

[24] Jack Hessel, Chenhao Tan, and Lillian Lee. 2016. Science, AskScience, and BadScience: On the Coexistence of Highly Related Communities. In *ICWSM*. 171–180.

[25] Shweta Jain and C. Seshadhri. 2017. A Fast and Provable Method for Estimating Clique Counts Using Turán's Theorem. In *WWW*. 441–449.

[26] Madhav Jha, C. Seshadhri, and Ali Pinar. 2015. Path Sampling: A Fast and Provable Method for Estimating 4-Vertex Subgraph Counts. In *WWW*. 495–505.

[27] Junteng Jia, Michael T. Schaub, Santiago Segarra, and Austin R. Benson. 2019. Graph-based Semi-Supervised & Active Learning for Edge Flows. In *KDD*.

[28] Mihail N Kolountzakis, Gary L Miller, Richard Peng, and Charalampos E Tsourakakis. 2012. Efficient triangle counting in large graphs via degree-based vertex partitioning. *Internet Mathematics* 8, 1-2 (2012), 161–185.

[29] Raunak Kumar, Paul Liu, Moses Charikar, and Austin R Benson. 2019. Retrieving Top Weighted Triangles in Graphs. *arXiv:1910.00692* (2019).

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

[31] Barbara S Lawrence. 2006. Organizational reference groups: A missing perspective on social context. *Organization Science* 17, 1 (2006), 80–100.

[32] Paul Liu, Austin R. Benson, and Moses Charikar. 2019. Sampling Methods for Counting Temporal Motifs. In *WSDM*. 294–302.

[33] Priya Mahadevan, Calvin Hubble, Dmitri Krioukov, Bradley Huffaker, and Amin Vahdat. 2007. Orbis: rescaling degree correlations to generate annotated internet topologies. In *ACM SIGCOMM Computer Communication Review*.

[34] Ron Milo, Shalev Itzkovitz, Nadav Kashtan, Reuven Levitt, Shai Shen-Orr, Inbal Ayzenshtat, Michal Sheffer, and Uri Alon. 2004. Superfamilies of Evolved and Designed Networks. *Science* (2004).

[35] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* 298, 5594 (2002), 824–827.

[36] Jukka-Pekka Onnela, Jari Saramäki, János Kertész, and Kimmo Kaski. 2005. Characterizing Motifs in Weighted Complex Networks. In *AIP*.

[37] Jukka-Pekka Onnela, Jari Saramäki, János Kertész, and Kimmo Kaski. 2005. Intensity and coherence of motifs in weighted complex networks. *PRE* (2005).

[38] Tore Opsahl and Pietro Panzarasa. 2009. Clustering in weighted networks. *Social networks* 31, 2 (2009), 155–163.

[39] Rasmus Pagh and Charalampos E Tsourakakis. 2012. Colorful triangle counting and a mapreduce implementation. *Inform. Process. Lett.* 112, 7 (2012), 277–281.

[40] Nataša Pržulj. 2007. Biological network comparison using graphlet degree distribution. *Bioinformatics* 23, 2 (2007), e177–e183.

[41] Mahmudur Rahman and Mohammad Al Hasan. 2013. Approximate triangle counting algorithms on multi-cores. In *IEEE Big Data*. 127–133.

[42] Mahmudur Rahman and Mohammad Al Hasan. 2014. Sampling Triples from Restricted Networks using MCMC Strategy. In *CIKM*. 1519–1528.

[43] Garry Robins, Pip Pattison, Yuval Kalish, and Dean Lusher. 2007. An introduction to exponential random graph p^* models for social networks. *Social Networks* 29, 2 (May 2007), 173–191.

[44] Pablo Robles, Sebastian Moreno, and Jennifer Neville. 2016. Sampling of attributed networks from hierarchical generative models. In *KDD*. 1155–1164.

[45] Karl Rohe and Tai Qin. 2013. The blessing of transitivity in sparse and stochastic networks. *arXiv* (2013).

[46] Ryan A. Rossi and Nesreen K. Ahmed. 2015. Role Discovery in Networks. *IEEE Transactions on Knowledge and Data Engineering* 27, 4 (2015), 1112–1131.

[47] Thomas Schank and Dorothea Wagner. 2005. Approximating Clustering Coefficient and Transitivity. *J. Graph Algorithms Appl.* 9, 2 (2005), 265–275.

[48] C. Seshadhri, Ali Pinar, and Tamara G. Kolda. 2014. Wedge sampling for computing clustering coefficients and triangle counts on large graphs. *Statistical Analysis and Data Mining* 7, 4 (2014), 294–307.

[49] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June (Paul) Hsu, and Kuansan Wang. 2015. An Overview of Microsoft Academic Service (MAS) and Applications. In *WWW '15 Companion*.

[50] Hossein Azari Soufiani and Edo Airoldi. 2012. Graphlet decomposition of a weighted network. In *AISTATS*. 54–63.

[51] Lorenzo De Stefani, Alessandro Epasto, Matteo Riondato, and Eli Upfal. 2017. TRIÈST: Counting Local and Global Triangles in Fully Dynamic Streams with Fixed Memory Size. *TKDD* 11, 4 (jun 2017), 1–50.

[52] Lorenzo De Stefani, Erisa Terolli, and Eli Upfal. 2017. Tiered sampling: An efficient method for approximate counting sparse motifs in massive graph streams. In *IEEE BigData 2017*. 776–786.

[53] Siddharth Suri and Sergei Vassilvitskii. 2011. Counting triangles and the curse of the last reducer. In *WWW*. 607–614.

[54] Jie Tang, Jing Zhang, Limin Yao, Juanzi Li, Li Zhang, and Zhong Su. 2008. ArnetMiner. In *KDD*.

[55] Charalampos E. Tsourakakis. 2008. Fast Counting of Triangles in Large Real Networks without Counting: Algorithms and Laws. In *ICDM*. 608–617.

[56] Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. 2009. Doulion: counting triangles in massive graphs with a coin. In *KDD*.

[57] Charalampos E Tsourakakis, Mihail N Kolountzakis, and Gary L Miller. 2011. Triangle Sparsifiers. *J. Graph Algorithms Appl.* 15, 6 (2011), 703–726.

[58] Duru Türkoglu and Ata Turk. 2017. Edge-Based Wedge Sampling to Estimate Triangle Counts in Very Large Graphs. In *ICDM*. IEEE, 455–464.

[59] Stanley Wasserman and Katherine Faust. 1994. *Social network analysis: Methods and applications*. Vol. 8. Cambridge university press.

[60] Brooke Foucault Welles, Anne Van Devender, and Noshir Contractor. 2010. Is a "friend" a friend? Investigating the structure of friendship networks in virtual worlds. In *CHI*. 4027–4032.

[61] Ellery Wulczyn and Dario Taraborelli. 2017. Wikipedia Clickstream.

[62] Kuai Xu, Feng Wang, and Lin Gu. 2014. Behavior Analysis of Internet Traffic via Bipartite Graphs and One-Mode Projections. *IEEE/ACM Transactions on Networking* 22, 3 (June 2014), 931–942.

[63] Hao Yin, Austin R. Benson, and Jure Leskovec. 2019. The Local Closure Coefficient: A New Perspective On Network Clustering. In *WSDM*. 303–311.