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

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

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.
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:

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

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 Nodelerator++) 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 Nodelerator++ [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.
We now develop a dynamic algorithm that uses the concepts of Al-
As the algorithm evolves, we adjust the sets
At a high level, our main deterministic algorithm will try to dynam-
τ may be enumerated than is necessary. The second is the difficulty
brute force and can always find the top-weight triangle given a
15 \end{algorithm}

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

\begin{algorithm}
\caption{Static heavy-light algorithm for finding top-weighted triangles.}
\begin{algorithmic}[1]
\State {\bf Input:} Weighted graph \(G = (V, E, w)\), scaling \(p\), number of triangles \(k\)
\State \(H \leftarrow \{ e \in E : w_e > \tau \} \)
\State \(T \leftarrow \) all triangles formed by edges in \(H\)
\State \textbf{return} \(k\) triangles in \(T\) with largest \(p\)-mean weight
\end{algorithmic}
\end{algorithm}

At a high level, our main deterministic algorithm will try to dynam-
A simple heavy-light algorithm. As a precursor to our dynamic
Given a threshold \(\tau\), 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 \(\tau\), 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
In practice, we find that this simple algorithm vastly outperforms
In practice, we find that this simple algorithm vastly outperforms
brute force and can always find the top-weight triangle given a
\begin{algorithm}
\caption{Dynamic heavy-light algorithm for finding the top-\(k\) weighted triangles.}
\begin{algorithmic}[1]
\State {\bf Input:} Weighted graph \(G = (V, E, w)\), scaling \(p\), number of triangles \(k\), parameter \(a_p\).
\State Sort \(E\) in decreasing order of weight
\State Initialize threshold \(\tau = \infty\), triangle set \(T = \emptyset\)
\State Initialize partitions \(S = H = \emptyset\), \(L = E\)
\State Initialize edge pointers \(h = l = -1\)
\State \begin{algorithmic}
\State if \(w_{\tau+1} > w_{\tau+1}^{a_p}\) then
\State \quad Move \(e_{\tau+1}\) from \(L\) to \(H\).
\State \quad \(Y\) = triangles formed by \(e_{\tau+1}\) and 2 edges from \(S \cup H\)
\State \quad \(Z\) = triangles formed by \(e_{\tau+1}\), 1 edge from \(L\), and 1 edge from \(S \cup H\)
\State \quad \(T = T \cup (Y \cup Z)\), \(h = h + 1\)
\State else
\State \quad Move \(e_{\tau+1}\) from \(H\) to \(S\).
\State \quad \(Y\) = triangles formed by \(e_{\tau+1}\) and 2 edges from \(L\)
\State \quad \(T = T \cup Y\), \(h = h + 1\)
\State \end{algorithmic}
\State \textbf{return} \(k\) triangles in \(T\) with largest \(p\)-mean weight
\end{algorithmic}
\end{algorithm}

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
eedge from \(H \cup L\). This upper bound is the \(p\)-th power of the
weight of that triangle to be \(\left(\frac{1}{3}(2w_n^p + w_k^p)\right)\). Our method tries to enumerate
triangles so that this bound decreases as quickly as possible.

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 \(\tau\) on the heaviest triangle we have not yet enumerated. The constant \(a_p\) 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 \(\tau\). We will specify this constant
later in our analysis. When the algorithm begins, the partitions are initialized with \(S = H = \emptyset\), \(L = \{e_1, \ldots, e_{m-1}\}\), 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 \(\tau^*\) 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 \(\tau\) hits \(\tau^*\) exactly, we only enumerate around \(k\) triangles.
As \(\tau\) 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
\(a_p\), or show how an optimal \(a_p\) 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 \( w_h + 2w_l \). By case analysis, there must either exist two edges with weight at least \( w_j \), 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 \( w_h + w_{t+1} + w_{t+2} \), as \( e_{h+1}, e_{l+1}, e_{l+2} \) is potentially an unenumerated triangle. However, this sum is at most \( w_h + 2w_l \) due to the monotonicity of the edge weights.

Since \( \tau \) 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 \( \alpha \). We next analyze how we might set this parameter in an optimal way.

3.3 Optimizing parameter settings
We now derive principles for setting the parameter \( \alpha \) 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 \( \tau(t) = w_h(t)^\beta + 2w_l(t)^\beta \). As a proxy to maximizing the enumeration rate, we maximize the rate at which the threshold decreases. To do this, we maximize the derivative \( d\tau/dt \) by adjusting \( w_h(t) \) and \( w_l(t) \) based on the input parameter \( \alpha \).

The derivatives \( dw_h/dt \) and \( dw_l/dt \) 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 \( CDF(w_h) \) and the cumulative 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 \( 1/PDF(w_l) \).

**Analysis for power-law distributed weights.** At this point, we are free to continue our analysis with any model 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 \( \beta \).

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 \( D \) 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 \( \alpha \) 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 \( \bar{d}_H \) 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 \( O(\bar{d}_G) \). Assuming that \( G_L \) has approximately as many edges as \( G \) (valid in the case of small \( k \)), \( \bar{d}_G \approx \bar{d}_G \). 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 \( O(\bar{d}_G) \). Since the number of edges in \( G_H \) is exactly \( |S \cup H| \), the assumptions on the weight distribution say that \( \bar{d}_G \approx O(\bar{d}_G) \). Thus, the cost of moving \( e_l \) is approximately \( O(\bar{d}_G) \). Combining this with Eq. (2), we obtain the following expressions for the derivatives

\[
dw_h/dt \approx O(\bar{d}_G^{\beta}/\bar{d}_G), \quad dw_l/dt \approx O(\bar{d}_G^{\beta}/\bar{d}_G). \tag{3}
\]

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 \( dt/dt = p w_i^{p-1} dw_i/dt + 2 p w_i^{p-2} dw_i/dt. \) Since at each iteration of the algorithm we can only choose to change one of \( e_k \) or \( e_l \), we should greedily change the pointer that gives the most "bang per buck", i.e., choose \( e_k \) and \( e_l \) such that

\[
w_h^{p-1} dw_h/dt = 2 w_l^{p-1} dw_l/dt \implies w_h \sim O \left( \frac{2}{w_l^{p-2} - p} \right). \tag{4}
\]

In other words, we should maintain the edge pointers \( e_k \) and \( e_l \) such that the weights are separated geometrically by \( \alpha = 2 - \frac{p}{p-1} \).

**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 \( \alpha \) 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 \( dw_h/dt \) and \( dw_l/dt \) 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 \( \alpha \) can also be used to compute \( \alpha \) implicitly. By maintaining an estimate of the derivatives \( dw_{h_1}/dt \) and \( dw_{c_1}/dt \) 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 \( w^{p-1} dw/dt \) (Eq. (4)).

Following the analysis in the previous section, the change in weight for \( e_k \) is estimated by the ratio of \( w_{h} - w_{c,k} \) 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 \( G_2 \), and similarly for \( e_k \) with \( G_2 \). 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 \( \alpha \) is successful, although in practice noise in the derivative estimates may cause this algorithm to be slower than using a fixed value of \( \alpha \). We find that setting \( \alpha = 1.25 \) 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. 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:

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

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:

\[
Pr(\text{sample edge } (a, b)) = \frac{w_{ab}^p}{Z}, \quad Z = \sum_{(u, v) \in E} w_{uv}^p.
\]

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 \( w_{uv}^p/Z \). 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 3: Weighted edge sampling (ES) algorithm

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

1. Initialize triangle set \( T \leftarrow \emptyset \)
2. for iteration 1, . . . , \( t \) do
3. \hspace{1em} Sample edge \((a, b) \propto w_{ab}^p \)
4. \hspace{1em} for each neighbor \( c \in N(a) \cap N(b) \) do
5. \hspace{2em} \( T \leftarrow T \cup \{(a, b, c)\} \)
6. \end
7. return \( k \) triangles in \( T \) with largest \( p \)-mean weight
Algorithm 4: Weighted wedge sampling (WS) algorithm

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

1. Initialize triangle set $T \leftarrow \emptyset$

2. for iteration 1, . . . , $t$

3. Sample node $a$ with probability as in Eq. (6)

4. Sample $b \in N(a)$ with probability as in Eq. (7)

5. Sample $c \in N(a)$ with probability as in Eq. (8)

6. if nodes $a, b, c$ form a triangle then

   $T \leftarrow T \cup \{(a, b, c)\}$

7. end

9. return $k$ triangles in $T$ with largest $p$-mean weight

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:

$$ P r(s a m p l e s a) = \frac{\hat{W}_1(a)}{Z_1} = 2 \cdot d_a \cdot D(a)/Z_1, \quad (6) $$

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

$$ P r(s a m p l e b \in N(a) \mid a) = \frac{\hat{W}_2(b \mid a)}{Z_2} = \frac{d_a \cdot w_{ab}^p + D(a)}{Z_2}, \quad (7) $$

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:

$$ P r(s a m p l e c \in N(a) \mid a, b) = \frac{\hat{W}_3(c \mid a, b)}{Z_3} = \frac{w_{ac}^p + w_{ab}^p}{Z_3}, \quad (8) $$

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 $w_p(a, b, c)/Z$, where $Z = \sum_{v \in V} d_v \cdot D(v)$.

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

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

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 $w_p(a, b, c)/Z$. 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 might 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.
Table 1: Summary statistics of datasets.

<table>
<thead>
<tr>
<th>dataset</th>
<th># nodes</th>
<th># edges</th>
<th>edge weight</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>mean</td>
</tr>
<tr>
<td>tags-stack-overflow</td>
<td>50K</td>
<td>4.2M</td>
<td>13</td>
</tr>
<tr>
<td>threads-stack-overflow</td>
<td>2.3M</td>
<td>21M</td>
<td>1.1</td>
</tr>
<tr>
<td>Wikipedia-clickstream</td>
<td>4.4M</td>
<td>23M</td>
<td>347</td>
</tr>
<tr>
<td>Ethereum</td>
<td>38M</td>
<td>103M</td>
<td>2.8</td>
</tr>
<tr>
<td>AMiner</td>
<td>93M</td>
<td>324M</td>
<td>1.3</td>
</tr>
<tr>
<td>reddit-reply</td>
<td>8.4M</td>
<td>435M</td>
<td>1.5</td>
</tr>
<tr>
<td>MAG</td>
<td>173M</td>
<td>545M</td>
<td>1.7</td>
</tr>
<tr>
<td>Spotify</td>
<td>3.6M</td>
<td>1.9B</td>
<td>8.6</td>
</tr>
</tbody>
</table>

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 the number of edges as 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).

**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 labeled 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).
WS was run to achieve just 50% accuracy and was stopped early if taking longer than BF (or longer than SHL on the Spotify well for finding top weighted triangles in large weighted graphs. Auto-HL. A similar speedup is observed for 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); 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.

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

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-processing 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 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;
REFERENCES


tions.


[6] Austin R. Benson, Rediet Abebe, Michael T. Schaub, Ali Jadbabaie, and Jon Kleinberg. 2018. Simons Investigator Award, Google Faculty Research Award; Amazon Research Award; and Google Cloud resources.

[7] Austin R. Benson, Rediet Abebe, Michael T. Schaub, Ali Jadbabaie, and Jon Kleinberg. 2018. Simons Investigator Award, Google Faculty Research Award; Amazon Research Award; and Google Cloud resources.


