Counting and Sampling Triangles from a Graph Stream

A. Pavan[†], Kanat Tangwongsan[◇], Srikanta Tirthapura[‡], Kun-Lung Wu[◇]
pavan@cs.iastate.edu, ktangwo@us.ibm.com, snt@iastate.edu, klwu@us.ibm.com
[†] Dept. of Computer Science, Iowa State University
[◇] IBM T. J. Watson Research Center, Yorktown Heights
[‡] Dept. of Electrical and Computer Engineering, Iowa State University

ABSTRACT

This paper presents a new space-efficient algorithm for counting and sampling triangles—and more generally, constant-sized cliques—in a massive graph whose edges arrive as a stream. Compared to prior work, our algorithm yields significant improvements in the space and time complexity for these fundamental problems. Our algorithm is simple to implement and has very good practical performance on large graphs.

1. INTRODUCTION

Triangle counting has emerged as an important building block in the study of social networks [23, 14], identifying thematic structures of networks [7], spam and fraud detection [4], link classification and recommendation [21], and more. The triangle is an important subgraph, and the number of triangles reveals important structural information about the network. For example, in social network analysis, widely used metrics such as *transitivity coefficient* and *clustering coefficient* use the number of triangles as a component. In these applications, streaming algorithms not only provide an attractive option for real-time processing of live data, but also benefit the analysis of large disk-resident graph data, allowing computations in one or a small number of passes over the data.

This paper addresses the question of counting and sampling triangles, as well as complete subgraphs, in the adjacency stream model [3, 9, 5]. More specifically, we study the following closely related problems:

(1) *Triangle Counting:* maintain an (accurate) estimate of the number of triangles in a graph;
(2) *Triangle Sampling:* maintain a uniformly-chosen random triangle from the set of all triangles in a graph;
(3) *Transitivity Coefficient:* maintain the transitivity coefficient of a graph, defined as three times the ratio between the number of triangles and the number of paths of length two (wedges); and
(4) *Higher-Order Clique Counting and Sampling:* maintain an accurate estimate of the count and a uniform sample of cliques of 4 or more vertices ($K_\ell, \ell \geq 4$).

The *adjacency stream model* is a streaming graph model where a given graph $G = (V, E)$ is presented as a stream of edges $\mathcal{S} = \langle e_1, e_2, e_3, \ldots, e_{|E|}\rangle$. In this notation, e_i denotes the i-th edge in the stream order, which is arbitrary and potentially chosen by an adversary. Let $n = |V|$, $m = |E|$, $\mathcal{T}(G)$ be the set of all triangles, and $\tau(G)$ denote the number of triangles, i.e., $\tau(G) = |\mathcal{T}(G)|$. We assume that the input graph is simple (no parallel edges and no self-loops).

Our algorithms are randomized and provide the following notion of probabilistic guarantees: for parameters $\varepsilon, \delta \in [0, 1]$, an (ε, δ)-approximation for a quantity X is a random variable $\hat{X}$ such that $|\hat{X} - X| \leq \varepsilon X$ with probability at least $1 - \delta$. We write $s(\varepsilon, \delta)$ as a shorthand for $1/\varepsilon^2 \cdot \log(1/\delta)$.

1.1 Our Contributions

— *Neighborhood Sampling:* We present *neighborhood sampling*, a new technique for counting and sampling cliques (K_ℓ for $\ell \geq 3$) in a graph stream. Neighborhood sampling is a multi-level inductive random sampling procedure: first, a random edge in the stream is sampled; then, in subsequent steps, an edge that shares an endpoint with the sampled edge(s) is sampled. We show that this simple technique leads to efficient one-pass algorithms with small space complexity for triangle counting and related problems.

— *Counting and Sampling Triangles:* Using neighborhood sampling, we present one-pass streaming algorithms for triangle counting and triangle sampling. The space complexity is $O(s(\varepsilon, \delta)m\Delta/\tau(G))$ for triangle counting and $O(m\Delta/\tau(G))$ for triangle sampling, where Δ is the maximum degree of any vertex. We provide a sharper space bound for triangle counting in terms of the "tangle coefficient" of the graph, which we define in our analysis. While in the worst case, this results in the space bound we have stated above, it is often much smaller in a typical case. These improve upon prior algorithms for the same problem (see Section 1.2).

We also present a method for quickly processing edges in bulks, which leads to a constant[1] amortized processing time per edge. This allows for the possibility of processing massive graphs quickly even on a modest machine.

—*Transitivity Coefficient:* An important metric closely related to triangle counting is transitivity coefficient—a measure of how "tight-knit" communities in the graphs are [15]. In the context of social networks, transitivity coefficient can be interpreted as a measure of relative frequency of "friend of a friend is a friend" occurrences. We extend our algorithms for neighborhood sampling and triangle counting to approximate the transitivity coefficient of a graph stream. Our streaming algorithm for estimating the transitivity coefficient has the same space complexity as the triangle counting algorithm.

— *Counting and Sampling Cliques:* We extend neighborhood sampling to the problem of sampling and counting cliques of size ℓ in

[1] In particular, by setting the batch size to $w = \Theta(r)$, where r is the number of unbiased estimators maintained, the update time per edge is amortized $O(1 + r/w) = O(1)$.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were invited to present their results at The 39th International Conference on Very Large Data Bases, August 26th - 30th 2013, Riva del Garda, Trento, Italy.

Proceedings of the VLDB Endowment, Vol. 6, No. 14
Copyright 2013 VLDB Endowment 2150-8097/13/14... $ 10.00.

the graph, $\ell \geq 4$. For $\ell = 4$, the space complexity of the counting algorithm is $O(s(\varepsilon, \delta) \cdot \eta/\tau_4(G))$, and the space complexity of the sampling algorithm is $O(\eta/\tau_4(G))$, where $\eta = \max\{m\Delta^2, m^2\}$ and $\tau_4(G)$ is the number of 4-cliques in G. General bounds for ℓ-cliques are presented in Section 5. To our knowledge, this is the best space complexity for counting the number of ℓ-cliques in a graph in the streaming model and improves on prior work due to Kane et al. [10].

— *Experiments:* Experiments with large real-world graphs show that our streaming algorithm for counting triangles is fast and accurate in practice. For instance, the Orkut network (for a description, see Section 4) with 117 million edges and 633 million triangles can be processed in 103 seconds, with a (mean) relative error of 3.55 percent, using 1 million instances of estimators. The experiment was run on a laptop, with an implementation that did not use parallelism. Our experiments also suggest that fewer than $\Theta(s(\varepsilon, \delta)m\Delta/\tau(G))$ estimators may be necessary to obtain good estimates.

1.2 Prior and Related Work

For triangle counting in adjacency streams, Bar-Yossef et al. [3] present the first algorithm using reductions to the problem of computing the zero-th and second frequency moments of streams derived from the edge stream. Their algorithm on the adjacency stream takes $s = O(\frac{s(\varepsilon,\delta)}{\varepsilon} \cdot (mn/\tau(G))^3)$ space and poly(s) time per item. They also show that in general, approximating $\tau(G)$ in the streaming model requires, in the worst-case, at least $\Omega(n^2)$ space.

The space and time bounds were subsequently improved. Jowhari and Ghodsi [9] present a one-pass streaming algorithm that uses space and per-edge processing time $O(s(\varepsilon, \delta)m\Delta^2/\tau(G))$. Our algorithm significantly improves upon this algorithm in both space and time. Note that the maximum degree Δ for large graphs can be significant; for instance, the Orkut graph has a maximum degree of greater than 66,000. They also give a three-pass streaming algorithm with space and per-edge processing time $O\left(s(\varepsilon, \delta) \cdot (1 + T_2(G)/\tau(G))\right)$, where $T_2(G)$ is the number of vertex triples with exactly two edges connecting them. Later, Buriol et al. [5] improve the space complexity and per-edge time $O(s(\varepsilon, \delta)mn/\tau(G))$. When the maximum degree Δ is small compared to n, our algorithm substantially improves upon theirs in terms of space; many real-world graphs tend to have $\Delta \ll n$. Another difference is that Buriol et al.'s algorithm needs to know the vertex set of the graph stream in advance, but ours does not. This can be a significant advantage in practice when vertices are being dynamically added to graph, or being discovered by the stream processor. Recently, Pagh and Tsourakakis [16] give a triangle counting algorithm on static (non-streaming) graphs and an implementation on the MapReduce platform. With some work, their algorithm can be adapted to the adjacency stream setting requiring space $O(1/\varepsilon^2 \cdot m\sigma/\tau(G) \cdot \log(1/\delta))$, where σ is the maximum number of triangles an edge is contained in. Our algorithm has better update time using bulk processing and ours requires less space when the tangle coefficient γ (Section 3.2.1) is smaller than σ. Most recently, Jha et al. [8] give a $O(\sqrt{n})$-space algorithm for estimating the number of triangles and the closely related problem of computing the clustering coefficient of a graph stream. Their algorithm has an additive error guarantee as opposed to the algorithms mentioned earlier, which had relative error guarantees. None of the techniques mentioned so far, as stated, extend to counting higher-order cliques.

On counting cliques, Kane et al. [10] present estimators for the number of occurrences of an arbitrary subgraph H in the stream. When applied to counting cliques on ℓ vertices in a graph, their space complexity is $O(s(\varepsilon, \delta) \cdot m^{\binom{\ell}{2}}/\tau_\ell^2(G))$ which is much higher than the space complexity that we obtain. We note that their algorithm works in the model where edges can be inserted or deleted (turnstile model) while ours is insert-only.

Manjunath et al. [13] present an algorithm for counting the number of cycles of length k in a graph; their algorithm works under dynamic inserts and deletes of edges. Since a triangle is also a cycle, this algorithm applies to counting the number of triangles in a graph, but uses space and per item processing time $\Theta(s(\varepsilon, \delta)m^3/\tau^2(G))$. When compared with our algorithm, their space and time bounds can be much larger, especially for graphs with a small number of triangles. Recent work on graph sketches by Ahn, Guha, and McGregor [1] also yield algorithms for triangle counting with space complexity, whose dependence on m and n is the same as in [5].

Newman, Watts, and Strogatz [15] proposed the notion of *transitivity coefficient* as a metric for analyzing and understanding the complexity of social network graphs. Transitivity coefficient is closely related to clustering coefficient of a graph [24] and is the same as weighted clustering coefficient[2] (for an appropriate choice of weight function). The algorithm of Buriol et al. [6] estimates the transitivity coefficient in the *incidence stream* model, which assumes that all edges incident at a vertex arrive together, and that each edge appears twice, once for each endpoint. In the incidence streams model, counting triangles is an easier problem, and there are streaming algorithms [5] that use space $O\left(s(\varepsilon, \delta)\left(1 + T_2(G)/\tau(G)\right)\right)$, and counting the number of length-two paths is straightforward. By contrast, we show that this space bound for triangle counting is not possible in the adjacency stream model.

Becchetti et al. [4] present algorithms for counting the number of triangles in a graph in a model where the processor is allowed $O(\log n)$ passes through the data and $O(n)$ memory. Their algorithm also returns for each vertex, the number of triangles that the vertex is a part of. There is a significant body of work on counting the number of triangles in a graph in the non-streaming setting, for example [19, 22]. We do not attempt to survey this literature. An experimental study of algorithms for counting and listing triangles in a graph is presented in [18].

2. PRELIMINARIES

For an edge e, let $V(e)$ denote the two end vertices of e. We say that two edges are adjacent to each other if they share a vertex. Given an edge e_i, the *neighborhood of* e_i, denoted by $N(e_i)$, is the set of all edges in the stream that arrive after e_i and are adjacent to e_i. Let $c(e_i)$ denote the size of $N(e_i)$. Further, for a triangle $t^* \in \mathcal{T}(G)$, define $C(t^*)$ to be $c(f)$, where f is its first edge in the stream. Our algorithms use a procedure $\texttt{coin}(p)$ that returns heads with probability p and a procedure $\texttt{randInt}(a, b)$ that returns an integer between a and b uniformly at random. We assume both procedures take constant time. Finally, we remember standard measure concentration bounds that will be used in the proofs:

Theorem 2.1 (Chernoff Bounds) *Let $\lambda > 0$ and $X = X_1 + \cdots + X_n$, where each X_i, $i = 1, \ldots, n$, is independently distributed in* $[0, 1]$. *Then, if $\mu = \mathbf{E}[X]$,*

$$\begin{aligned} \mathbf{Pr}[X \geq (1 + \lambda)\mu] &\leq e^{-\frac{\lambda^2}{2+\lambda}\cdot\mu} \quad \textit{and} \\ \mathbf{Pr}[X \leq (1 - \lambda)\mu] &\leq e^{-\frac{\lambda^2}{2}\cdot\mu}. \end{aligned}$$

3. SAMPLING & COUNTING TRIANGLES

In this section, we present algorithms for sampling and counting triangles. We begin by describing *neighborhood sampling*, a basic method upon which we build an algorithm for counting triangles (Section 3.2), an efficient implementation for bulk processing (Section 3.3), and an algorithm for sampling triangles (Section 3.4).

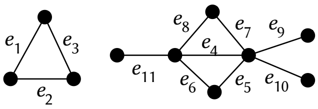


Figure 1: An example of a streaming graph, where the edges arrive in order $e_1, e_2, \ldots$, forming triangles $t_1 = \{e_1, e_2, e_3\}$, $t_2 = \{e_4, e_5, e_6\}$, and $t_3 = \{e_4, e_7, e_8\}$.

3.1 Neighborhood Sampling for Triangles

Neighborhood sampling is an algorithm for maintaining a random triangle in a graph stream. In broad strokes, it first samples a random edge r_1 from the edge stream (using, e.g., reservoir sampling). It then samples a random edge r_2 from a "substream" of edges that appear after r_1 and are adjacent to r_1. That is, the second edge is sampled from the neighborhood of the first edge. This sample can also be maintained using reservoir sampling on the appropriate substream. With these edges selected, the wedge $r_1 r_2$ defines a potential triangle, and the algorithm tries to close it with a subsequent edge in the stream, forming a triangle t.

The triangle found by this procedure, however, is *not* necessarily uniformly chosen from $\mathcal{T}(G)$. As an example, in the graph in Figure 1, the probability that the neighborhood sampling procedure chooses triangle t_1 is the probability that e_1 is chosen into r_1 (which is $\frac{1}{10}$), and then from among the edges adjacent to e_1 (i.e., e_2 and e_3), e_2 is chosen into r_2, for a total probability of $\frac{1}{2} \cdot \frac{1}{10} = \frac{1}{20}$. But the probability of choosing t_2 is the probability of choosing e_4 into r_1 (which is $\frac{1}{10}$), and then from among those edges adjacent to e_4 and arrive after e_4 (i.e., $\{e_5, e_6, e_7, e_8, e_9, e_{10}, e_{11}\}$), e_5 is chosen into r_2 (which is $\frac{1}{7}$), for a total probability of $\frac{1}{7} \cdot \frac{1}{10} = \frac{1}{70}$. This bias poses a challenge in our algorithms but can be normalized away by keeping track of how much bias is incurred on the potential triangle.

We briefly contrast our algorithm with two other algorithms for the adjacency stream that use random sampling: Like ours, Buriol et al.'s algorithm [5] first samples a random edge from the stream, say r_1, but then unlike ours, it picks a random vertex that is not necessarily incident on an endpoint of r_1. The edge and the vertex together form a potential triangle, and the algorithm then waits for the triangle to be completed by the remaining two edges. In our algorithm, instead of selecting a random third vertex, we select a vertex that is already connected to r_1. This leads to a greater chance that the triangle is completed, and hence better space bounds.

The approach of Pagh and Tsourakakis [16], unlike Buriol et al.'s and ours, does not rely on producing a random triangle. Instead, it uses randomness on the vertices to whittle down the input stream: Each vertex is assigned a random "color" and the algorithm maintains a subgraph $\tilde{G}$ of the stream by admitting only edges whose endpoints are given the same color. Thus, the number of colors controls the size of $\tilde{G}$. Then, queries about the stream are answered by computing the statistic in $\tilde{G}$ and scaling the answer appropriately. This leads to space bounds that depend on different graph parameters than ours and are incomparable in general.

We now describe the neighborhood sampling algorithm in detail. The algorithm maintains the following state:

— Level-1 edge r_1: uniformly sampled from the edges so far;

— Level-2 edge r_2: uniformly sampled from $N(r_1)$, i.e., those edges in the graph stream that are adjacent to r_1 and arrive after r_1;

— Counter c: maintains the invariant that $c = c(r_1) = |N(r_1)|$, i.e, the number of edges adjacent to r_1 and appearing after r_1; and

— Triangle t: a triangle formed using r_1 and r_2.

Algorithm 1: Algorithm NSAMP-TRIANGLE

```
Initialization: Set $(r_1, r_2, t, c) \leftarrow (\emptyset, \emptyset, \emptyset, 0)$
Upon receiving edge $e_i, i \geq 1$:
if coin($1/i$) = "head" then
    $(r_1, r_2, t, c) \leftarrow (e_i, \emptyset, \emptyset, 0)$          // $e_i$ is the new level-1 edge.
else
    if $e_i$ is adjacent to $r_1$ then
        $c \leftarrow c + 1$;
        if coin($1/c$) = "head" then
            $(r_2, t) \leftarrow (e_i, \emptyset)$          // $e_i$ is the new level-2 edge.
        else
            if $e_i$ forms a triangle with $r_1$ and $r_2$ then
                $t \leftarrow \{r_1, r_2, e_i\}$          // $e_i$ closes the wedge $r_1 r_2$.
```

We summarize the neighborhood sampling algorithm in Algorithm 1 and analyze the probability that a triangle is sampled by the algorithm in the following lemma:

Lemma 3.1 *Let t^* be a triangle in the graph. The probability that $t = t^*$ in the state maintained by Algorithm 1 after observing all edges (note t may be empty) is*

$$\mathbf{Pr}[t = t^*] = \frac{1}{m \cdot C(t^*)}$$

where we recall that $C(t^*) = c(f)$ if f is t^*'s first edge in the stream.

PROOF. Let $t^* = \{f_1, f_2, f_3\}$ be a triangle in the graph, whose edges arrived in the order f_1, f_2, f_3 in the stream, so $C(t^*) = c(f_1)$ by definition. Let $\mathcal{E}_1$ be the event that f_1 is stored in r_1, and $\mathcal{E}_2$ be the event that f_2 is stored in r_2. We can easily check that neighborhood sampling produces t^* if and only if both $\mathcal{E}_1$ and $\mathcal{E}_2$ hold.

Now we know from reservoir sampling that $\mathbf{Pr}[\mathcal{E}_1] = \frac{1}{m}$. Furthermore, we claim that $\mathbf{Pr}[\mathcal{E}_2 | \mathcal{E}_1] = \frac{1}{c(f_1)}$. This holds because given the event $\mathcal{E}_1$, the edge r_2 is randomly chosen from $N(f_1)$, so the probability that $r_2 = f_2$ is exactly $1/|N(f_1)|$, which is $1/c(f_1)$, since c tracks the size of $N(r_1)$. Hence, we have

$$\begin{aligned} \mathbf{Pr}[t = t^*] &= \mathbf{Pr}[\mathcal{E}_1 \cap \mathcal{E}_2] = \mathbf{Pr}[\mathcal{E}_1] \cdot \mathbf{Pr}[\mathcal{E}_2 \mid \mathcal{E}_1] \\ &= \frac{1}{m} \cdot \frac{1}{c(f_1)} = \frac{1}{m \cdot C(t^*)} \end{aligned}$$

□

3.2 Counting Triangles

For a given triangle $t^* \in \mathcal{T}(G)$, neighborhood sampling produces t^* with probability $\frac{1}{mC(t^*)}$. To estimate the number of triangles, we first turn this into a random variable with the correct expectation:

Lemma 3.2 *Let t and c be the values the neighborhood sampling algorithm maintains, and m be the number of edges observed so far. Define*

$$\tilde{\tau} = \begin{cases} c \times m & \text{if } t \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}$$

Then, $\mathbf{E}[\tilde{\tau}] = \tau(G)$.

PROOF. By Lemma 3.1, we sample a particular triangle t^* with probability $\mathbf{Pr}[t = t^*] = \frac{1}{mC(t^*)}$. Since $c = C(t)$, we have that if $t = t^*$, then $\tilde{\tau} = mC(t^*)$. Therefore, the expected value of $\tilde{\tau}$ is

$$\mathbf{E}[\tilde{\tau}] = \sum_{t^* \in \mathcal{T}(G)} mC(t^*) \cdot \mathbf{Pr}[t = t^*] = |\mathcal{T}(G)| = \tau(G).$$

□

[2] This differs from (unweighted) clustering coefficient; see, e.g., [17]

Given an estimate with the right expected value, to obtain good accuracy with a high enough probability, we keep multiple independent copies of the estimate and aggregate them. The following theorem answers the question: how many such estimates are sufficient to obtain an (ε, δ)-approximation?

Theorem 3.3 (Triangle Counting) *Let $0 < \delta, \varepsilon \leq 1$ and $r \geq 1$. There is a streaming algorithm using $O(r)$ space that on an arbitrarily-ordered stream of any graph G, returns an (ε, δ)-approximation to the triangle count in G, provided that $r \geq \frac{6}{\varepsilon^2} \frac{m\Delta}{\tau(G)} \log\left(\frac{2}{\delta}\right)$.*

PROOF. Let $\alpha = \frac{6}{\varepsilon^2} \frac{m\Delta}{\tau(G)} \log\left(\frac{2}{\delta}\right)$. We show that the average of $r \geq \alpha$ independent unbiased estimates from Lemma 3.2 is an (ε, δ)-approximation. For $i = 1, \ldots, r$, let X_i be the value of the i-th estimate. Let $\bar{X} = \frac{1}{r} \sum_{i=1}^{r} X_i$ denote the average of these estimators. Then, by Lemma 3.2, we have $\mathbf{E}[X_i] = \tau(G)$ and $\mathbf{E}[\bar{X}] = \tau(G)$. Further, for $e \in E$, we have $c(e) \leq 2\Delta$, so $X_i \leq 2m\Delta$. Let $Y_i = X_i/(2m\Delta)$ so that $Y_i \in [0, 1]$. By letting $Y = \sum_{i=1}^{r} Y_i$, we have $\mathbf{E}[Y] = r \cdot \tau(G)/(2m\Delta)$. Thus, we have that $\mathbf{Pr}\left[\bar{X} > (1 + \varepsilon)\,\mathbf{E}[X]\right]$ is at most

$$\mathbf{Pr}\left[\textstyle\sum_i Y_i > (1 + \varepsilon)\,\mathbf{E}[Y]\right] \leq e^{-\frac{\varepsilon^2}{3}\mathbf{E}[Y]} \leq \delta/2$$

by Chernoff bound (Theorem 2.1). Similarly, we can show that $\mathbf{Pr}[X < (1 - \varepsilon)\,\mathbf{E}[X]] \leq \delta/2$. Hence, with probability at least $1 - \delta$, the average $\bar{X}$ approximates the true count within $1 \pm \varepsilon$. Since each estimator only takes $O(1)$ space, the total space is $O(r)$. □

3.2.1 A Sharper Space Bound For Triangle Counting

The sufficient condition in the previous theorem is rather conservative. In practice, we tend to observe much better results than the theorem suggests. We now show a sharper bound on the space requirement by being more precise about the spectrum of the $c(\cdot)$ values. Toward this goal, we define a measure that captures the amount of interaction between triangles and non-triangle edges in the graph. The *tangle coefficient* of a graph G, denoted by $\gamma(G)$, is given by

$$\gamma(G) := \frac{1}{\tau(G)} \sum_{t' \in \mathcal{T}(G)} C(t')$$

or equivalently, $\gamma(G) = \frac{1}{\tau(G)} \sum_{e \in E} c(e)s(e)$, where $s(e)$ counts the number of triangles $t' \in \mathcal{T}(G)$ such that the first edge of t' is e. Using the tangle coefficient, we prove the following theorem, which gives a different way to aggregate the results of different unbiased estimators (we briefly discuss the intuition behind tangle coefficient after that):

Theorem 3.4 (Improved Triangle Counting) *Let $\gamma(G)$ denote the tangle coefficient of a graph G. Let $0 < \delta, \varepsilon \leq 1$ and $r \geq 1$. There is a streaming algorithm using $O(r)$ space that on an arbitrarily-ordered stream of any graph G, returns an (ε, δ)-approximation to the triangle count in G, provided that $r \geq \frac{48}{\varepsilon^2} \frac{m\gamma(G)}{\tau(G)} \log\left(\frac{1}{\delta}\right)$.*

PROOF. First, we note that an estimator in Lemma 3.2 has variance at most $m \sum_{t \in \mathcal{T}(G)} C(t) = m \cdot \tau(G)\gamma$. We will run $\alpha = 4/\varepsilon^2 \cdot \gamma m/\tau(G)$ independent parallel copies of such an estimator. Let the average of these estimates be Y. By Chebyshev's inequality, we have $\mathbf{Pr}[|Y - \mathbf{E}[Y]| > \varepsilon \cdot \tau(G)] \leq \frac{1}{4}$. To increase the success probability to $1 - \delta$, we will run $\beta = 12 \ln(1/\delta)$ independent copies of Y estimators and take the median. Hence, our median estimator fails to produce an ε-approximation only if more than $\beta/2$ fails to produce an ε-approximation. In expectation, the number of "failed" estimators is at most $\beta/4$. Therefore, by a Chernoff bound (Theorem 2.1), it fails with probability at most $\exp(-\frac{1^2(\beta/4)}{3}) = \delta$. We conclude that the final estimate is an (ε, δ)-approximation using at most $O(\alpha\beta)$ space. □

Notice that the tangle coefficient γ is at most 2Δ, recovering the original bound we proved. But γ is often much smaller than that, especially in graphs such as power-law graphs, where there are only a few vertices with degree close to Δ and the degrees of rest of the vertices are much smaller than Δ. We can gain more understanding of the tangle coefficient γ by considering the following random process: Fix a stream and pick a random triangle from this graph. If e is the first edge in the stream of this triangle, then the value of γ is the number of edges that are adjacent to e and come after e. In this view, γ can be seen as a measure of how entangled the triangles in this stream are—as our intuition suggests, if the triangles "interact" with many non-triangle triples, we will need more space to obtain an accurate answer.

3.3 Nearly-Linear Time Triangle Counting

Our discussion thus far directly leads to a simple $O(mr)$-time implementation, where r is the number of estimators being maintained; however, this can be too slow for large graphs. We want the algorithm to take time linear in the number of edges m and the number of estimators r.

This section describes a bulk-processing scheme that delivers significantly better performance than the naïve implementation. Bulk processing helps because the estimators need to be updated much less often. It also represents a common usage pattern: in many applications, the algorithm receives edges in bulk (e.g., block reads from disk) or has infrequent queries compared to updates (e.g., users only make queries every once in a while, so the incoming changes can be queued up). We show the following bounds:

Theorem 3.5 (Bulk Processing) *Let $r \leq m$ be the number of estimators. There is an algorithm `bulkTC` for triangle counting that processes a batch of w edges in time $O(r + w)$. Furthermore, at any point in time, the algorithm needs at most $O(r + w)$ space.*

As a corollary, using a batch size of $w = \Theta(r)$, we can process m edges in $\lceil m/w \rceil$ batches in $\Theta(m + r)$ total time using $O(r)$ space. That is, with a (small) constant factor increase in space, we are able to achieve $O(m + r)$ time bound as opposed to $O(mr)$.

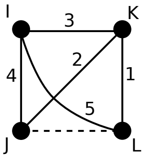
3.3.1 Conceptual Overview

Suppose our estimators have already observed the stream $\mathcal{S} = \langle e_1, \ldots, e_m \rangle$ and there is a batch $B = \langle b_1, \ldots, b_{|B|} \rangle$ of newly-arriving edges. We will devise an algorithm that simultaneously advances the states of all r estimators to the point after incorporating the batch B. Effectively, the states of the r estimators will be as if they have observed the stream $\mathcal{S}$ concatenated with B. The accuracy guarantees from the previous section follow directly because our process simulates playing these edges one by one in the same order.

We will frequently refer to the prefixes of B: To facilitate the discussion, let $B^{(i)} = \langle b_1, \ldots, b_i \rangle$ ($B^{(0)}$ is the empty sequence), and for a sequence of edges A, let $\deg_A(x)$ denote the degree of node x in the graph induced by the edge set A. Furthermore, we define two quantities:

$$c^-(r) = |N(r) \setminus B| \quad \text{and} \quad c^+(r) = |N(r) \cap B|$$

We remember that neighborhood sampling (Section 3.1) maintains for each estimator the following quantities: (1) r_1—a uniformly-chosen random edge from the stream; (2) r_2—a uniform sample from $N(r_1)$; (3) t—a triangle if the edge closing the wedge $r_1 r_2$ is found in $N(r_2)$; and (4) $c = |N(r_1)|$. Conceptually, the batch B can be incorporated as follows:



	I	J	K	L
$\deg_{B^{(1)}}$ (after KL)	$\emptyset$	$\emptyset$	1	1
$\deg_{B^{(2)}}$ (after JK)	$\emptyset$	1	2	1
$\deg_{B^{(3)}}$ (after IK)	1	1	3	1
$\deg_{B^{(4)}}$ (after IJ)	2	2	3	1
$\deg_{B^{(5)}}$ (after IL)	3	2	3	2

Figure 2: A 4-node streaming graph where a batch $B = \langle \mathsf{KL}, \mathsf{JK}, \mathsf{IK}, \mathsf{IJ}, \mathsf{IL} \rangle$ of 5 new edges (solid, labeled with their arrival order) is being added to a stream of 1 edge (dashed) which has arrived earlier—and the corresponding deg values as these edges are added. Degree values that change are circled.

Step 1: *Resample Level-1 edges.* If there were m edges prior to the batch B of size w, for each estimator, keep the current edge with probability $\frac{m}{w+m}$, and with the remaining probability, replace it with an edge uniformly chosen from B. We will also reset c to 0 if the level-1 edge is replaced.

Step 2: *Identify Level-2 candidates and sample from them.* For each estimator, the sample space we want to sample uniformly from is $N(r_1)$. Given the current r_2, we know that $c^-(r_1)$ is the current c (note that if the level-1 was replaced, c was also reset to 0). Hence, we use the following rule to update Level-2 edges:

> With probability $\frac{c^+(r_1)}{c^-(r_1)+c^+(r_1)}$, pick a random edge from $N(r_1) \cap B$; otherwise, keep the current r_2. Moreover, the number of candidate edges c is $c^+(r_1) + c^-(r_1)$.

Step 3: *Detect edges that will close the wedges.* For each estimator with level- 1 and 2 edges r_1 and r_2, we check if an edge in B that comes after r_2 can close the wedge $r_1 r_2$ and update t accordingly.

3.3.2 *An Efficient Implementation*

We give an efficient implementation of the conceptual algorithm. The implementation maintains r estimator states $\mathsf{est}_1, \ldots, \mathsf{est}_r$, where each est_i maintains $\mathsf{est}_i.r_1$, $\mathsf{est}_i.r_2$, $\mathsf{est}_i.t$, and $\mathsf{est}_i.c$ according to neighborhood sampling. When we store an edge, we also keep the position in the stream where it appears.

Implementing Step 1: In one `randInt`$(1, r+w)$ call, we can decide for each estimator whether to retain the current level-1 edge or which new edge in B to select. Thus, in $O(r)$ time, all level-1 edges are properly updated. This is essentially a direct translation of the conceptual algorithm.

Implementing Step 2: Although conceptually rather simple, this step turns out to be the most involved part in the whole algorithm. The challenge is that we cannot afford to explicitly construct the candidate sets (our r estimators can potentially have all different candidate sets). Instead, we need to navigate the input stream while implicitly tracking the r "substreams" simultaneously in space $O(r + w)$—merely a constant factor more than the space needed to keep the estimator states and the newly-arrived edges alone.

The crux of our solution is an observation that characterizes the candidate set of a given level-1 edge (i.e., $N(r_1) \cap B$) in terms of the degrees of the relevant nodes in the prefixes of B. We state the observation first and explain it with examples after that:

Observation 3.6 *For an edge e and a node $w \in e$, the set of edges in B incident to node w that comes after e is*

$$\Gamma^{(e)}(w) = \{b_i \in B : w \in b_i \text{ and } \deg_{B^{(i)}}(w) > \beta^{(e)}(w)\}$$

where $\beta^{(e)}(w) = \deg_{B^{(k)}}(w)$ if e is $b_k \in B$, or $\beta^{(e)}(w) = 0$ otherwise.

Therefore, for an edge $e = \{x, y\}$, the set of edges in B adjacent to e that come after e is given by $N(e) \cap B = \Gamma^{(e)}(x) \cup \Gamma^{(e)}(y)$.

To help understand this observation and subsequent descriptions, imagine adding the edges in B one by one—$b_1, b_2, \ldots, b_w$—and at every step, observe how the degrees of the nodes relevant to B change; see Figure 2 for a running example. In this view, $\beta^{(e)}(w)$ is the degree of the node w at the point when the edge e is added. In our running example, $\beta^{(\mathsf{JK})}(\mathsf{K}) = 2$ and $\beta^{(\mathsf{IK})}(\mathsf{I}) = 1$. Moreover, since $\mathsf{JL} \notin B$, we have $\beta^{(\mathsf{JL})}(\mathsf{L}) = 0$. Now, notice that as an edge $e = \{x, y\}$ is added, exactly two places in the degree vector change—i.e., the degrees of x and y increase by 1 (these are circled). Hence, if $b_j \in B$ comes after $b_i \in B$ and both are incident on a node x, the degree of x when b_j is added is necessarily larger than that when b_i was added—or mathematically, $\deg_{B^{(j)}}(x) > \deg_{B^{(i)}}(x)$. Therefore, in words, Observation 3.6 says that every edge incident on x with a degree of x higher than $\beta^{(e)}(x)$ comes after e in the stream. To give some examples, consider that $\beta^{(\mathsf{IK})}(\mathsf{I}) = 1$, so the edges incident on I that comes after IK are exactly those with $\deg_{B^{(i)}}(\mathsf{I}) > 1$—that is, IJ and IL. As another example, we know that $\beta^{(\mathsf{IK})}(\mathsf{K}) = 3$, so there is no edge incident on K that comes after IK because none of the edges have $\deg_{B^{(i)}}(\mathsf{K}) > 3$.

Algorithm 2: `edgeIter`—a degree-keeping edge iterator

```
▷ deg[] – an array where deg[$u$] contains the degree of vertex
  $u$, initially 0.
for $i = 1$ to $|B|$ do
    $\{x, y\} \leftarrow b_i$
    deg[$x$] = deg[$x$] + 1 and deg[$y$] = deg[$y$] + 1
    notify(EVENTA ($i, \{x, y\}$, deg)).
    notify(EVENTB ($i, \{x, y\}, x$, deg[$x$])).
    notify(EVENTB ($i, \{x, y\}, y$, deg[$y$])).
return deg.
```

To apply the observation, we describe an edge iterator algorithm `edgeIter` (Algorithm 2) that maintains a table of the nodes' degrees. The variable `deg` is initially an empty table and at the end of each iteration i, `deg` contains the degrees of all nodes present in $B^{(i)}$. Moreover, the algorithm generates the following two types of events as it rolls over the batch of edges:

EVENTA ($i, \{x, y\}$, `deg`)—after going over the edge $\{x, y\}$ at index i, the current degree vector ($\deg_{B^{(i)}}$) is `deg`.

EVENTB ($i, \{x, y\}, v, a$)—after going over the edge $\{x, y\}$ at index i, the degree of node v is now updated to a.

These events provide the "cues" needed to track the candidate sets for our estimators.

The implementation of Step 2 consists of three substeps:

▷ ***Step 2a:*** Calculate for each est_i with level-1 edge $\{x, y\}$, the values of $\beta^{(r_1)}(x)$, $\beta^{(r_1)}(y)$, $\deg_B(x)$ and $\deg_B(y)$; these quantities are sufficient to define implicitly the set $N(r_1) \cap B$ according to Observation 3.6. To accomplish this, we temporarily store two additional fields per estimator—$\beta^{(r_1)}(x)$, $\beta^{(r_1)}(y)$—and make a table $L[]$ mapping $L[i]$, $i \in \{1, \ldots, |B|\}$, to a list of estimators whose level-1 edge is b_i (an "inverted index" of the estimators that just replaced their level-1 edges). Building this table takes $O(r)$ time. Initially, set $\beta^{(r_1)}(x)$, $\beta^{(r_1)}(y)$ to 0 for all estimators. Then, we run `edgeIter` and on EVENTA ($i, \{x, y\}$, `deg`), we go over the list $L[i]$ and store in these estimators $\beta^{(r_1)}(x) = \deg_{B^{(i)}}(x) = \texttt{deg}[x]$ and $\beta^{(r_1)}(y) = \deg_{B^{(i)}}(y) = \texttt{deg}[y]$. When `edgeIter` is finished, its return vector is $\deg_B$. This can be implemented in $O(r + w)$ time because the length of lists in L (i.e. $\sum_i |L[i]|$) is r.

▷ ***Step 2b:*** Pick the level-2 edges, each in terms of an EVENTB. With the information collected in 2a, Observation 3.6 fully defines the sample space for level-2 edges in terms of EVENTB's. To illustrate, consider that in Figure 2, the neighborhood of level-1 edge IK is $N(\mathsf{IK}) = \{\mathsf{IJ}, \mathsf{IL}\}$. The edge IJ creates two EVENTB's that we will use to identify it: EVENTB $(4, \{\mathsf{I}, \mathsf{J}\}, \mathsf{I}, 2)$ and EVENTB $(4, \{\mathsf{I}, \mathsf{J}\}, \mathsf{J}, 2)$. Likewise, the edge IL generates two EVENTB's: EVENTB $(5, \{\mathsf{I}, \mathsf{L}\}, \mathsf{I}, 3)$ and EVENTB $(5, \{\mathsf{I}, \mathsf{L}\}, \mathsf{L}, 2)$.

More generally, consider a level-1 edge $r_1 = \{x, y\}$. There are $a = \deg_B(x) - \beta^{(r_1)}(x)$ edges in $\Gamma^{(r_1)}(x)$, namely the edges corresponding to EVENTB $(*, *, x, \beta^{(r_1)}(x) + 1), \ldots,$ EVENTB $(*, *, x, \deg_B(x))$, where $*$ denotes a wildcard match. Similarly, there are $b = \deg_B(y) - \beta^{(r_1)}(y)$ edges in $\Gamma^{(r_1)}(y)$, namely the edges corresponding to EVENTB $(*, *, y, \beta^{(r_1)}(y) + 1), \ldots,$ EVENTB $(*, *, y, \deg_B(y))$.

Thus, for each estimator, we have $c^-(r_1) = c$ (inherited from Step 1) and $c^+(r_1) = a + b$. Let $\varphi = \texttt{randInt}(1, c^- + c^+)$, where $c^- = c^-(r_1)$ and $c^+ = c^+(r_1)$ and translate φ as follows:

Algorithm 3: Translating edge numbers into events

```
if ($\varphi \le c^-$) then keep existing $r_2$
else if ($\varphi \le c^- + a$) then
    pick the edge that causes EVENTB$(*, *, x, \beta^{(r_1)}(x) + \varphi - c^-)$
else
    pick the edge that causes EVENTB$(*, *, y, \beta^{(r_1)}(y) + \varphi - c^- - a)$
```

▷ ***Step 2c:*** Convert the selection into actual edges. Each estimator has decided which new edge, if any, is replacing the current level-2 edge but it is given in terms of an EVENTB event. To convert these events into actual level-2 edges, we create a (hash) table $P[]$ mapping an ordered-pair (x, d) to a list of estimators that subscribe to the event EVENTB $(*, *, x, d)$ from the previous step. Then, we run `edgeIter` and on EVENTB, we consult the table P to see which estimators, if any, selected this edge. Step 2c requires $O(r + w)$ time and P consumes at most $O(r)$ space since $\sum_{\ell \in P} |\ell| \le r$.

Implementing Step 3: We keep a (hash) table Q that maps an edge needed to complete a triangle to the estimator(s) that needs it. This table can be populated by by looking at our estimator states. Once this table is constructed, we simply have to go through the edges in B and check whether b_i is present in Q and if so, whether it comes after the level-2 edge that expects it. This is easy to check because we store with every edge its position in the stream. Hence, Step 3 takes at most $O(r + w)$ time since Q can contain at most r entries.

In sum, we have shown that each of these steps can be implemented in $O(r + w)$ time using at most $O(r + w)$ space. Hence, the algorithm `bulkTC` satisfies Theorem 3.5, as promised.

3.4 Sampling Triangles

We now turn to the problem of maintaining $k \ge 1$ triangle(s) uniformly-sampled with replacement from a graph stream. First, we show how to sample one triangle uniformly with a reasonable success probability. Following that, we apply it to sample $k \ge 1$ triangles and outline how to adapt the efficient implementation (Section 3.3) to this case.

Our starting point is the neighborhood sampling algorithm, which maintains a random triangle. The problem, however, is that this sample is not necessarily uniform. But this is easy to fix, as described in the following lemma:

Lemma 3.7 *Let G be a streaming graph. Let t and c be the values the neighborhood sampling algorithm maintains. Define*

$$\texttt{unifTri}(G) = \begin{cases} t & \textit{with prob. } \frac{c}{2\Delta} \\ \emptyset & \textit{otherwise.} \end{cases}$$

Then, $\texttt{unifTri}(G)$ produces a triangle (i.e. $\texttt{unifTri}(G) \neq \emptyset$) with probability at least $\frac{\tau(G)}{2m\Delta}$. Furthermore, if it returns a triangle, each triangle in $\mathcal{T}(G)$ is equally likely to be produced.

PROOF. By Lemma 3.1, neighborhood sampling picks a particular triangle t^* with probability $\mathbf{Pr}[t = t^*] = 1/mC(t^*)$. Further, note that $C(t^*) \le 2\Delta$. Therefore, if neighborhood sampling picks the triangle t^*, the probability that t^* is passed on as our output is $\frac{1}{mC(t^*)} \cdot \frac{C(t^*)}{2\Delta} = \frac{1}{2m\Delta}$, where we note that $c = C(t^*)$ and the normalization factor $\frac{c}{2\Delta} \le 1$. Finally, since the events of different triangles being returned are all disjoint from each other, the probability that some triangle is returned by the algorithm is $\frac{\tau(G)}{2m\Delta}$. □

We extend this to sample k triangles by running multiple copies of `unifTri` and picking any k triangles at random. Below, we calculate the number of copies sufficient for producing k triangles with probability at least $1 - \delta$.

Theorem 3.8 *For $0 < \delta < 1$, $r \ge 1$, and $k \ge 1$, there is an algorithm $\texttt{unifTri}(G, k)$ with space requirement $O(r)$ that on any stream graph G with $\tau(G) > 0$, returns k triangles uniformly-chosen with replacement from G that succeeds with probability at least $1 - \delta$, as long as $r \ge \frac{4mk\Delta \ln(e/\delta)}{\tau(G)}$.*

PROOF. We run $r \ge \alpha = \frac{4mk\Delta \ln(e/\delta)}{\tau(G)}$ independent copies of `unifTri` from Lemma 3.7 and randomly pick k triangles these copies yield. Thus, it succeeds as long as at least k copies yield a triangle. To analyze this, let X be the number of copies that yield a triangle and $p = 2m\Delta/\tau(G)$ be a sampler's success probability (Lemma 3.7). Therefore, X is a Binomial random variable with mean αp because these are independent samplers succeeding with probability p. Hence, $\texttt{unifTri}(G, k)$ succeeds with probability $\mathbf{Pr}[X \ge k] = 1 - \mathbf{Pr}[X < k]$, where $\mathbf{Pr}[X < k] \le \exp(-\frac{1}{2\alpha p}(\alpha p - k)^2) \le \delta/e \le \delta$ by Chernoff bounds (Theorem 2.1). We conclude that the algorithm succeeds with probability at least $1 - \delta$ and consumes $O(r)$ space because each copy takes constant space. □

The algorithm $\texttt{unifTri}(G, k)$ can also benefit from the bulk-processing implementation from Section 3.3. Since Theorem 3.5 implements neighborhood sampling for bulk arrival, the states it keeps are readily consumable by our `unifTri` algorithm (Lemma 3.7). Hence, together with the theorem above, we have that as long as we keep $r \ge \frac{4mk\Delta \ln(e/\delta)}{\tau(G)}$, we can maintain k uniformly-sampled triangles in the same space/time bounds as Theorem 3.5; that is, $O(m + r)$ total time and $O(r + w)$ space.

3.5 Transitivity Coefficient

Another important triadic graph measure is the *transitivity coefficient*, proposed by Newman, Watts, and Strogatz [15]:

$$\kappa(G) = \frac{3\tau(G)}{\zeta(G)},$$

where $\zeta(G) = \sum_{u \in V(G)} \binom{\deg(u)}{2}$ is the number of connected triplets in the graph G. Since we already have an algorithm for triangle counting, we will estimate the transitivity coefficient by designing an algorithm for estimating $\zeta(G)$. To start, we express $\zeta(G)$ in terms of a quantity related to what neighborhood sampling tracks:

Claim 3.9 *If $c(e)$ is the number of edges incident on e that arrive after e in the stream, then*

$$\zeta(G) = \sum_{e \in E} c(e)$$

PROOF. Let $\mathbb{S} = \{\{e, f\} \subseteq E(G) : e \neq f \text{ and } |e \cap f| = 1\}$ be the set of unordered pairs of edges that share a common vertex, so $|\mathbb{S}| = \zeta(G)$. Further, let $\mathbb{T} = \{(p, q) : p \in E(G) \text{ and } q \in N(p)\}$, so $|\mathbb{T}| = \sum_{e \in E} c(e)$. We prove the claim by giving a bijection between $\mathbb{S}$ and $\mathbb{T}$, showing that $|\mathbb{S}| = |\mathbb{T}|$. For this, we map each $(p, q) \in \mathbb{T}$ to $\{p, q\} \in \mathbb{S}$. It is routine to check that this mapping is well-defined and is bijective. □

Using this relationship, we can estimate $\zeta(G)$ as follows:

Lemma 3.10 *Let c be the quantity that neighborhood sampling tracks and m be the total number of edges seen so far. Define*

$$\tilde{\zeta} = m \times c.$$

Then, $\mathbf{E}\left[\tilde{\zeta}\right] = \zeta(G)$.

PROOF. Fix an edge $e \in E$. Neighborhood sampling picks this edge as its level-1 edge with probability $1/m$ and when this edge is picked, the value c is $c(e)$, so $\tilde{\zeta} = m \times c(e)$. Thus, $\mathbf{E}\left[\tilde{\zeta}\right] = \sum_e \frac{1}{m} c(e) \cdot m = \zeta(G)$, which follows from Claim 3.9. □

Like in triangle counting, we boost the accuracy and success probability of the above estimator by running multiple copies and taking the average of the estimates. By Chernoff bounds (applied similarly to Theorem 3.3), we have the following lemma:

Lemma 3.11 *Let* $0 \leq \delta, \varepsilon \leq 1$. *There is an algorithm that observes a graph stream G and returns an* (ε, δ)*-approximation for* $\zeta(G)$ *using space* $O(\frac{1}{\varepsilon^2} \frac{m\Delta}{\zeta(G)} \log(1/\delta))$.

We put these together as follows: Run the triangle counting algorithm simultaneously with the ζ-estimation algorithm. In particular, run the algorithm from Theorem 3.3 to compute $\tau'(G)$, a $(\varepsilon/3, \delta/2)$-approximation of $\tau(G)$, and the algorithm from Lemma 3.11 to compute $\zeta'(G)$, a $(\varepsilon/3, \delta/2)$-approximation of $\zeta(G)$—and return $\kappa'(G) = 3\tau'(G)/\zeta'(G)$. Note that the total space used by this algorithm is bounded by $O(\frac{1}{\varepsilon^2} \frac{m\Delta}{\zeta(G)} \log(1/\delta))$. By our choice of parameters and an application of union bound it follows that $\kappa'(G) \in (1 \pm \varepsilon)\kappa(G)$. Thus, we have the following theorem:

Theorem 3.12 *Let* $\varepsilon \leq 1, \delta > 0$ *and* $r \geq 1$. *There is an* $O(r)$*-space* (ε, δ)*-approximation streaming algorithm for transitivity coefficient, provided* $r \geq \frac{K}{\varepsilon^2} \frac{m\Delta}{\zeta(G)} \log(1/\delta))$ *for a fixed constant K.*

3.6 A Lower Bound

It is natural to ask whether better space bounds are possible for triangle sampling and counting. In particular, *can we meet the space complexity of* $O(1+T_2(G)/\tau(G))$*, which is the space-complexity of an algorithm for triangle counting in the incidence stream model* [5]? Here, $T_2(G)$ is the number of vertex triples with exactly two edges in them. We show that this space bound is not possible in the adjacency stream model, separating it from the incidence stream model:

Theorem 3.13 *There exists a graph* G^* *and an order of arrival of edges, such that any randomized streaming algorithm that can estimate the number of triangles in* G^* *with a relative error of better than* $1/2$ *must have space complexity* $\omega(1 + T_2(G)/\tau(G))$.

PROOF. We use a reduction from the Index problem from communication complexity: Alice is given a bit vector $x \in \{0, 1\}^n$ and Bob is given an index $k \in \{1, 2, \ldots, n\}$; Bob wants to compute x_k, the bit in the k-th position in x. It is known that in the model where Alice can send exactly one message to Bob, the communication cost of any (randomized) protocol is $\Omega(n)$ bits (see [11, Chapter 4.2]).

Let $\mathcal{A}$ be a streaming algorithm that estimates the number of triangles. We can use this algorithm to solve the Index problem as follows. Given a bit vector $x \in \{0, 1\}^n$, Alice constructs a graph G^* on $3(n+1)$ vertices with the vertex set $\{a_0, \cdots, a_n\} \cup \{b_0, \cdots, b_n\} \cup \{c_0, \cdots, c_n\}$; the edges are as follows: she forms a triangle on a_0, b_0, c_0—and for each $i \in \{1, 2, \ldots, n\}$, she places the edge (a_i, b_i) if and only if $x_i = 1$. Alice processes this graph using $\mathcal{A}$ and sends the state of the algorithm to Bob, who continues the algorithm using the state sent by Alice, and adds the two edges (b_k, c_k) and (c_k, a_k), where k is the index requested. By querying the number of triangles in this graph with relative error of smaller than $1/2$, Bob can distinguish between the following cases: (1) G^* has two triangles, and (2) G^* has one triangle. In case (1), $x_k = 1$ and in Case (2), $x_k = 0$, and hence, Bob has solved the Index problem.

It follows that the memory requirement of the streaming algorithm an Alice's end must be $\Omega(n)$ bits. Note that the graph G^* sent by Alice has no triples with two edges between them, and hence, $O(1 + T_2(G^*)/\tau(G^*)) = O(1)$. □

4. EXPERIMENTS

In this section, we empirically study the accuracy and the runtime performance of the proposed triangle counting algorithm on real-world, as well as synthetic, graphs and in comparison to existing streaming algorithms.

We implemented the version of the algorithm which processes edges in batches and also the other state-of-the-art algorithms for adjacency streams due to Buriol et al. [5], and Jowhari and Ghodsi [9]. Our bulk-processing algorithm follows the description in Section 3.3, with the following optimizations: First, we combine Steps 2c and 3 to save a pass through the batch. Second, level-1 maintenance can be further optimized by noticing that in later stages, the number of estimators that end up updating their level-1 edges is progressively smaller. Deciding which estimator gets a new edge boils down to generating a binary vector where p is the probability that a position is 1 (as time progresses, p becomes smaller and smaller). We can generate this by generating a few geometric random variables representing the gaps between the 1's in the vector. Since we expect only a p-th fraction of the estimators to be updated, this is more efficient than going over all the estimators. Our implementation uses GNU's STL implementation of collections, including unordered_map for hash maps; we have only performed basic code optimization.

4.1 Experimental Setup

Our experiments were performed on a 2.2 Ghz Intel Core i7 laptop machine with 8GB of memory, but our experiments only used a fraction of the available memory. The machine is running Mac OS X 10.7.5. All programs were compiled with GNU g++ version 4.2.1 (Darwin) using the flag -O3. We measure and report wall-clock time using gettimeofday.

We use a collection of popular social media graphs, obtained from the publicly available data provided by the SNAP project at Stanford [12]. We present a summary of these datasets in Figure 3. We remark that while these datasets stem from social media, our algorithm does not assume any special property about them. Since these social media graphs tend to be power-law graphs, we complement the datasets with a randomly-generated synthetic graph (called "Synthetic ∼ d-regular") that has about the same number of nodes and edges as our biggest real dataset (Orkut)—but the nodes have degrees between 42 and 114.

The algorithms considered are randomized and may behave differently on different runs. For robustness, we perform *five* trials with different random seeds and report the following statistics: **(1)** the mean deviation (relative error) values from the true answer across the trials, **(2)** the median wall-clock overall runtime, and **(3)** the median I/O time. Mean deviation is a well accepted measure of

Dataset	n	m	Δ	τ	$m\Delta/\tau$
Amazon	335K	926K	549	667,129	761.9
DBLP	317K	1.0M	343	2,224,385	161.9
Youtube	1.13M	3.0M	28,754	3,056,386	28,107.1
LiveJournal	4.00M	34.7M	14,815	177,820,130	2,889.4
Orkut	3.07M	117.2M	33,313	633,319,568	6164.0
Syn. ˜*d*-reg	3.07M	121.4M	114	848,519,155	16.3

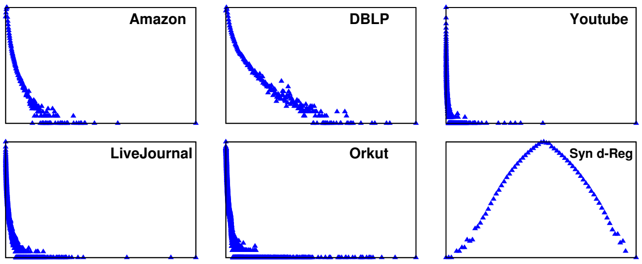


Figure 3: A summary of the datasets used in our experiments: the left panel shows for every dataset the number of nodes (n), the number of edges (m), the maximum degree (Δ), the number of triangles in the graph (τ), and the ratio $m\Delta/\tau$; the right panel shows for each dataset, a plot of frequency (vert. axis in log scale) vs. degree (linear scale).

error, which we believe to give an accurate picture of how well the algorithm performs. For some experiments, we also report the min/max deviation values for completeness, but we note that as more trials are performed, the minimum becomes smaller and the maximum becomes larger, so they are not robust.

4.2 Baseline Study

The state-of-the-art algorithms for adjacency streams are due to Buriol et al. [5], and Jowhari and Ghodsi [9]; we discuss their performance in turn.

Our implementation of Buriol et al.'s algorithm follows the description of the optimized version in their paper, which achieves roughly $O(m+r)$ running time for m edges and r estimators through certain approximations. Even though the algorithm is fast, it fails to find a triangle most of the time, resulting in low-quality estimates, or producing no estimates at all—even when using millions of estimators on the large graphs that we consider (see Section 3.1 for a related discussion); this is consistent with Buriol et al.'s findings about the adjacency stream algorithm [5]. Hence, we do not report further results on this algorithm.

Our implementation of Jowhari-Ghodsi's algorithm follows the description in their paper, which achieves $O(mr)$ total running time. It is, however, too slow for extensive experiments on the datasets in Figure 3. Hence, we use smaller datasets to compare with our algorithm:

— *Syn* 3-*reg* is a 3-regular graph with $n = 2,000$ nodes; $m = 3,000$ edges, and a max. degree of $\Delta = 3$, containing $\tau = 1,000$ triangles.

— *Hep-Th* is an arXiv Hep-Th collaboration network [12] with $n = 9,877$ nodes; $m = 51,971$ edges; $\Delta = 130$ max. degree with $\tau = 90,649$ triangles.

The results are shown in Tables 1 and 2. On the synthetic 3-regular graph, which has very small $m\Delta/\tau = 9$, both algorithms give accurate estimates yielding better than 92% accuracy even with only $r = 1,000$ estimators. Both yield progressively better estimates with more estimators with JG being more accurate. However, our bulk-processing algorithm is at least 10x faster. On the *Hep-Th* graph, which has $m\Delta/\tau = 74.53$, neither algorithms is able to produce a reasonable estimate across 5 runs with $r = 1,000$ or $10,000$. But with $r = 100,000$ estimators, our algorithm's error drops to below 1% while we see no improvements from JG. Again, the bulk-processing algorithm is at least 10x faster. Importantly, we note that for the same value of r (number of estimators), the JG algorithm uses considerably more space than our algorithm, since our algorithm uses constant space per estimator while the JG algorithm uses up to $O(\Delta)$ space per estimator.

Algorithm	$r = 1,000$		$r = 10,000$		$r = 100,000$	
	MD	Time	MD	Time	MD	Time
JG [9]	7.20	0.04	2.08	0.44	0.27	5.26
Ours	4.28	0.004	1.52	0.01	0.93	0.07

Table 1: The accuracy (mean deviation **in percentage**) and processing time (**in seconds**) of Jowhari-Ghodsi(JG)'s algorithm in comparison to our algorithm on a synthetic 3-regular graph (*Syn.* 3-*reg*) as the number of estimators is varied.

Algorithm	$r = 1,000$		$r = 10,000$		$r = 100,000$	
	MD	Time	MD	Time	MD	Time
JG [9]	79.33	0.71	86.86	7.17	86.66	86.02
Ours	92.69	0.05	81.25	0.08	0.68	0.17

Table 2: The accuracy (mean deviation **in percentage**) and processing time (**in seconds**) of Jowhari-Ghodsi(JG)'s algorithm in comparison to our algorithm on *Hep-Th* as the number of estimators is varied.

These findings suggest that *existing algorithms for adjacency streams can deliver accurate results but require more estimators, memory, and runtime for obtaining results of similar quality as our algorithm, and hence are impractical for large graphs.* In the remaining experiments, we directly compare our results with the true count to assess accuracy and study the scalability of the approach as the graph size and the number of estimators increase.

4.3 Accuracy, Speed, and Memory Usage

This set of experiments aims to study the accuracy, speed, and memory usage of our estimates on different datasets. Our theoretical results predict that as the number of estimators r increases, so does the accuracy. We are interested in verifying this prediction, as well as studying the dependence of the accuracy on parameters such as the number of edges m, maximum degree Δ and the number of triangles τ.

For each dataset, we run 5 trials of the bulk-processing algorithm with $r = 1024$, 131072, and 1048576. The number of estimators (r) controls the memory requirement for keeping the states. Our implementation uses 36 bytes per estimator; therefore, regardless of the datasets, the memory usage for the estimators is as follows:

	$r = 1\text{K}$	$r = 128\text{K}$	$r = 1\text{M}$
Memory	36K	4.5M	36M

In addition, when a batch of edges arrives, the bulk-processing algorithm needs a working space of about 3x the batch size to process the batch; this space is thrown away after it completes the batch. We further discuss the effects of batch size in Section 4.5.

Dataset	r = 1K		r = 128K		r = 1M		I/O
	min/mean/max dev.	Time	min/mean/max dev.	Time	min/mean/max dev.	Time	
Amazon	1.60 / 6.28 / 12.45	0.41	0.11 / 0.84 / 1.52	1.06	0.08 / 0.25 / 0.40	3.72	0.26
DBLP	8.04 / 18.28 / 36.53	0.45	0.08 / 0.50 / 0.97	1.08	0.07 / 0.19 / 0.42	3.90	0.28
Youtube	12.56 / 59.45 / 79.76	1.25	9.37 / 21.46 / 38.49	2.39	1.75 / 4.42 / 10.18	5.26	0.79
LiveJournal	0.24 / 11.53 / 29.76	15.00	1.41 / 2.35 / 4.02	23.10	0.19 / 0.60 / 1.45	33.40	10.00
Orkut	4.61 / 31.93 / 58.93	52.40	2.13 / 4.69 / 12.69	75.20	1.48 / 3.55 / 5.93	103.00	33.40
Syn. ~d-regular	1.26 / 7.58 / 13.57	53.70	0.00 / 0.37 / 0.81	64.80	0.01 / 0.24 / 0.53	73.00	34.50

Table 3: The accuracy (min/mean/max deviation **in percentage**), median total running time (**in seconds**), and I/O time (**in seconds**) of our bulk algorithm across five runs as the number of estimators r is varied.

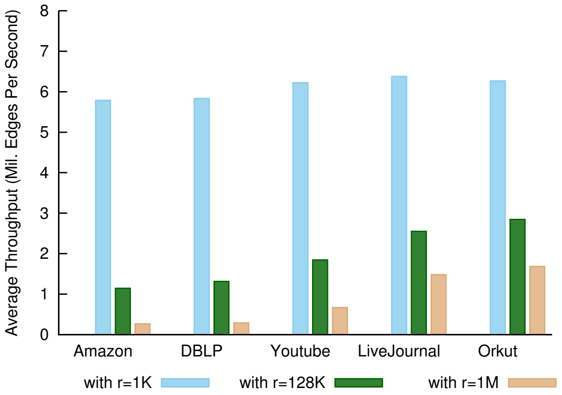


Figure 4: Average throughput of the streaming algorithm (**in million edges per second**) on different real-world datasets as the number of estimators is varied.

Table 3 shows the median total running time, accuracy (showing min., mean, and max. relative errors in percentage), and median I/O time of our algorithm across five runs as we vary the number of estimators r. We show the I/O time for each dataset as it makes up of a non-negligible fraction of the total running time. Several things are clear from this experiment in terms of accuracy: First, *our algorithm is accurate with only a modest number of estimators.* In all but the Youtube dataset, the algorithm achieves less than 5% mean deviation using only about 100 thousand estimators. Furthermore, the accuracy significantly improves as we increase the number of estimators r to 1M. Second, *a high degree graph with few triangles needs more estimators.* Consistent with the theoretical findings, Youtube and Orkut, which have the largest $m\Delta/\tau(G)$ values, need more estimators than others to reach the same accuracy. Third, but perhaps most importantly, *in practice, far fewer estimators than suggested by the pessimistic theoretical bound is necessary to reach a desired accuracy.* For example, on Orkut, using $\varepsilon = 0.0355$, the expression $s(\varepsilon, \delta)m\Delta/\tau$ is at least 4.89 million, but we already get this accuracy using 1 million estimators. Finally, we note that across the datasets, *the mean deviation decreases as we increase the number of estimators.* This is desirable and to be expected; as the proof suggests, the aggregate becomes progressively sharper as we increase the number of estimators.

Power-law or not. Our real-world datasets stem from social media, which tend have a power-law degree distribution. It is natural to wonder how the scheme performs on non-power-law graphs. For this, we experimented with a synthetic graph of roughly the same size as Orkut. The result on the synthetic graph suggests that as long as $m\Delta/\tau$ is small, our algorithm is capable of producing accurate results—regardless of the power-law nature. In more "regular" graphs, we believe that $m\Delta/\tau$ will be a better predictor of accuracy because the degree variance is smaller.

Having discussed the accuracy and memory usage trends, we now turn to the throughput of the algorithm. Figure 4 shows the average throughput of our algorithm for different real-world datasets as the number of estimators is varied. These numbers represent the throughput we obtain on average for processing a stream of edges of a particular length. Since the I/O rate is faster than the processing rate and we factor out the I/O time, these are the highest "sustained" rate the algorithm can achieve on that stream. As expected, the throughput decreases as the number of estimators increases (which yields more accuracy). Furthermore, for a fixed r (and using, say, $w = 8r$), the throughput increases with the stream's length: since the total time to process a stream of length m is proportional to $m+r$, the throughput is proportional to $\frac{m}{m+r} = \frac{1}{1+r/m}$, which increases with m. We focus on $r = 128$K and $r = 1$M as these are likely the values one is likely to use to obtain an accurate estimate. With $r = 128$K, the algorithm is capable of handling more than 1 million edges/second—and with $r = 1$M, for sufficiently long streams, it can process more than 1.2 million edges/second.

4.4 Effects of The Number of Estimators

In this set of experiments, we selected two graphs—Youtube and LiveJournal—to study the algorithm's behaviors in more detail.

Figure 5 (left) shows the median running time (in seconds) as r geometrically increases from $r = 1024$ to $r = 4$M; the horizontal axis (x-axis) is in log scale. This translates into the average throughput curves in the middle panel. As can be seen, the running time increases with the number of estimators r, as expected. The theory predicts that the running time is $O(m + r)$; that is, it scales linearly with r. This is hard to confirm visually since we do not know how much of it is due to the $O(m)$ term; however, in both cases, we are able to compute a value t_0 such that the running times minus t_0 scale roughly linearly with r, suggesting that the algorithm conforms to the $O(m + r)$ bound.

We also study the mean deviation as r changes. Figure 5 (right) shows the mean deviation values for the same two datasets as well as the error bound indicated by Theorem 3.3 using $\delta = 1/5$. Ignoring the first several data points where using a small number of estimators results in low accuracy and high fluctuations, we see that in general—though not a strict pattern—the error decreases with the number of estimators. Furthermore, consistent with our earlier observations, the bound in Theorem 3.3 is conservative; we do better in practice.

4.5 Effects of Batch Size

In this section, we discuss the effects the batch size have on memory consumption and the processing throughput. When a batch of edges arrives, the bulk-processing algorithm needs a working space of about 3x the space for storing the arriving edges to process the batch this space is thrown away after it completes each batch.

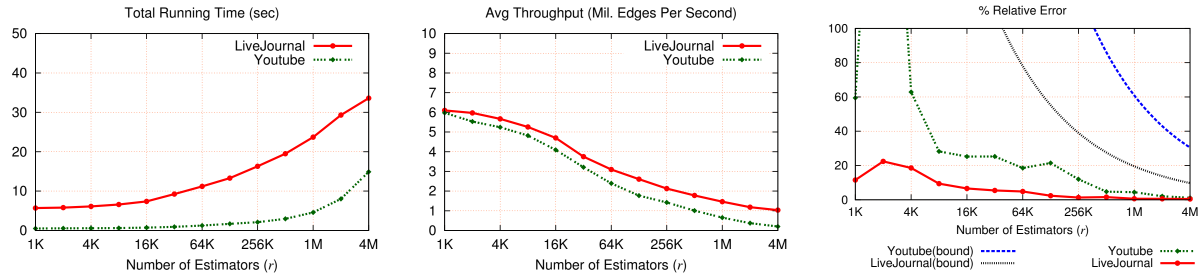


Figure 5: (Left) running time (**in seconds**), (center) average throughput (**in million edges per second**), (right) mean deviation (**in percentage**) as as the number of estimators r is varied ($r = 1\text{K}, 2\text{K}, \ldots, 4\text{M}$). The x-axis is in log scale.

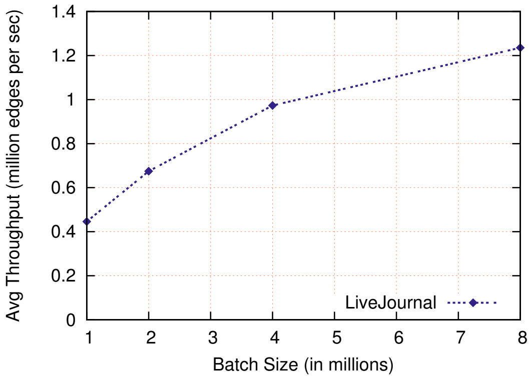


Figure 6: The throughput of the bulk-processing algorithm on LiveJournal as the batch size is varied.

As an example, using a batch size of $w = 8\text{M}$ and 1M estimators, the implementation would have a high water mark around 164MB. Furthermore, we study the effects on the throughput by performing an experiment on LiveJournal graph with $r = 1\text{M}$, varying the batch size between 1M and 8M. The results are shown in Figure 6, suggesting that the throughput increases with the batch size. This agrees with the running time bound we derived in Theorem 3.5: the average time to process an edge in a stream of length m is proportional to $1 + \frac{r}{m} + \frac{w}{m} + \frac{1}{w}$.

5. EXTENSIONS

In this section, we demonstrate the versatility of the neighborhood sampling technique by using it to derive a space-efficient streaming algorithm for counting higher-order cliques, as well as applying it to the problem of triangle counting in a sliding window model. These results are mostly of theoretical interest.

5.1 Counting and Sampling Cliques

For ease of presentation, we focus on 4-cliques. Let $\mathcal{T}_4(G)$ denote the set of all 4-cliques in G, and $\tau_4(G)$ the number of 4-cliques in G. The most crucial ingredient of our triangle counting algorithm is the neighborhood sampling subroutine. Following the same recipe for estimating 4-cliques, our first step would be to design a neighborhood sampling algorithm for 4-cliques. Given a proper neighborhood sampling algorithm, it is easy to obtain an unbiased estimator, and taking the average of multiple copies of the unbiased estimator yields a good estimate for the number of 4-cliques.

Extending neighborhood sampling to 4-cliques requires extra care, however. In the case of triangles, we first sample an edge r_1, and then samples a neighbor from $N(r_1)$, and waits for the arrival of the third edge that completes the triangle. A natural extension to 4-cliques is to sample an edge r_1, sample r_2 from $N(r_1)$, and sample r_3 from the set of neighbors of r_1 and r_2 that arrive after r_2—then, wait for the remaining edges to complete the 4-clique. Unfortunately, this strategy will miss many 4-cliques: Consider a 4-clique $\kappa^* = \{f_1, f_2, f_3, f_4, f_5, f_6\}$ whose edge arrive in that order. If f_1 and f_2 do not share a vertex, the above strategy will never sample κ^*. However, we would like our neighborhood sampling algorithm to pick each 4-clique with a non-zero probability.

We overcome this difficulty by partitioning the set of 4-cliques into two classes—`Type I` and `Type II`—and design two neighborhood sampling algorithms to estimate their numbers separately. The type of a 4-clique is determined by the order its edges appear in the stream. Let $f_1, \ldots, f_6$ be the edges of a 4-clique κ^* arriving in that order. We say that κ^* is a `Type I` clique if f_2 and f_1 share a common vertex; otherwise κ^* is a `Type II` clique. Thus, every clique is, by definition, either `Type I` or `Type II`. Let $\mathcal{T}_4^1(G)$ be the set of `Type I` cliques in G and $\mathcal{T}_4^2(G)$ be the set of `Type II` cliques; let $\tau_4^1(G)$ and $\tau_4^2(G)$ denote the corresponding sizes.

For sampling purposes, the basic distinction between `Type I` and `Type II` cliques is as follows: Let $f_1, \ldots, f_6$ be the edges of a 4-clique κ^* arriving in that order. If κ^* is a `Type I` clique, since f_1 and f_2 share a vertex, three of the first four edges (f_1, f_2 and one of f_3 or f_4) determine all four vertices of the clique. If κ^* is a `Type II` clique then the first two edges (f_1, f_2) determine all four vertices of the clique.

5.1.1 *Neighborhood Sampling for 4-Cliques*

Let f_1 and f_2 be two edges of $\mathcal{S}$, where f_2 arrives after f_1. Let F be the set of edges adjacent to both f_1 and f_2 (possibly empty). The neighborhood of f_1 and f_2, denoted by $N(f_1, f_2)$, is

$$N(f_1, f_2) = \{e \in N(f_1) \mid e \text{ comes after } f_2\} \cup N(f_2) - F,$$

and let $c(f_1, f_2)$ denote the size of $N(f_1, f_2)$.

Neighborhood sampling for `Type I` cliques. As mentioned earlier, we need three edges to determine all vertices of a `Type I` clique. Thus, our sampling algorithm maintains a sample of three edges over four vertices and looks for edges that complete the 4-clique. Our algorithm maintains following states:

- level-1 edge r_1, level-2 edge r_2, level-3 edge r_3—r_1 is uniformly sampled from all the edges seen so far; r_2 is drawn at random from $N(r_1)$; r_3 is drawn at random from $N(r_1, r_2)$.
- counters c_1 and c_2—c_1 tracks the size of $N(r_1)$ and c_2 tracks the size of $N(r_1, r_2)$.
- potential clique κ_1—edges of a potential clique that can be formed using edges r_1, r_2, and r_3.

We use the standard reservoir sampling technique to pick r_1, r_2 and r_3. We present a more detailed description in Algorithm 4.

Algorithm 4: NSAMP-Type I

```
Initialization: Set $r_1, r_2, r_3, \kappa_1$ to $\emptyset$, and $c_1, c_2$ to 0.
Edge $e_i$ arrives;
if coin$(1/i)$ = "head" then
    $r_1 = e_i$;  $c_1 = 0$;  $\kappa_1 = \{r_1\}$;
else
    if $e_i$ is adjacent to $r_1$ then
        Increment $c_1$
        if coin$(1/c_1)$ = "head" then
            $r_2 = e_i$;  $c_2 = 0$;  $\kappa_1 = \{r_1, r_2\}$;
        else
            if $e_i$, $r_1$, and $r_2$ form a triangle then
                $\kappa_1 = \kappa_1 \cup \{e_i\}$;
            else
                if $e_i$ is adjacent to $r_1$ or $r_2$ then
                    $c_2 = c_2 + 1$;
                    if coin$(1/c_2)$ = "head" then
                        $r_3 = e_i$;  $\kappa_1 = \kappa_1 \cup \{r_3\}$
                    else
                        If $e_i$ is an edge connecting two end
                        points of $r_1, r_2$ or $r_3$, then add $e_i$ to $\kappa_1$.
```

Lemma 5.1 *Consider a* `Type I` *clique κ^* and let $f_1, \ldots, f_6$ be its edges in the order they appeared in the stream. After Algorithm 4 has processed the entire graph, the probability that κ_1 equals κ^* is $\frac{1}{m \cdot c(f_1) \cdot c(f_1, f_2)}$.*

PROOF. Since κ^* is `Type I`, f_1 and f_2 are adjacent to each other, and they together fix 3 vertices of the clique. The edge f_3 is adjacent to either one or both of f_1 and f_2. Consider the case when f_3 is adjacent to only one of f_1 or f_2, but not both; the other case can be handled similarly. Note that κ equals κ^* if the following events all hold: (1) $\mathcal{E}_1$: f_1 equals r_1, (2) $\mathcal{E}_2$: f_2 equals r_2, (3) $\mathcal{E}_3$: f_3 equals r_3.

Since r_1 is chosen uniformly at random among all possible m edges, the probability of $\mathcal{E}_1$ is $1/m$. Since r_2 is an edge that is chosen uniformly at random from $N(r_1)$, $\mathbf{Pr}[\mathcal{E}_2 \mid \mathcal{E}_1] = \frac{1}{c(f_1)}$. Finally, note that r_3 is chosen uniformly at random from $N(r_1, r_2)$. Thus, $\mathbf{Pr}[\mathcal{E}_3 \mid \mathcal{E}_1, \mathcal{E}_2] = \frac{1}{c(f_1, f_2)}$. Thus, the probability that κ equals κ^* is $\frac{1}{m} \cdot \frac{1}{c(f_1)} \cdot \frac{1}{c(f_1, f_2)}$, as desired. □

Neighborhood sampling for `Type II` **cliques.** Remember that to determine the vertices of a `Type II` clique we only need two edges. The neighborhood sampling algorithm randomly chooses two edges r_1 and r_2 so that $V(r_1) \cap V(r_2) = \emptyset$ and looks for remaining edges that can form a clique using r_1 and r_2. In the interest of space, we omit the formal description of the algorithm, which mirrors that of `Type I` algorithm, and prove the following lemma:

Lemma 5.2 *There is a neighborhood sampling algorithm, that maintains a state named κ_2, with the following property: For every* `Type II` *clique κ^*, after neighborhood sampling algorithm has processed the entire graph, the probability $\mathbf{Pr}[\kappa_2 = \kappa^*] = \frac{1}{m^2}$.*

PROOF. Suppose that the edges of κ^* in the stream order were $f_1, f_2, \ldots, f_6$ respectively. It is easy to see that $\kappa_2 = \kappa^*$ if and only if at the end of observation, $r_1 = f_1$ and $r_2 = f_2$. Then, since the events $r_1 = f_1$ and $r_2 = f_2$ are independent and each has a probability of $1/m$ of being true, the probability $\mathbf{Pr}[\kappa_2 = \kappa^*] = 1/m^2$, as claimed. □

5.1.2 Estimating 4-cliques

Using these two neighborhood sampling algorithms, we can derive unbiased estimators for $\tau_4^1(G)$ and $\tau_4^2(G)$, as follows:

Lemma 5.3 *Let c_1, c_2 and κ_1 be the values that Algorithm 4 maintains after observing m edges. Define a random variable X to be $c_1 \cdot c_2 \cdot m$ if κ_1 is a 4-clique, and 0 otherwise. Then, $\mathbf{E}[X] = \tau_4^1(G)$.*

PROOF. First, we observe that X is nonzero only when κ_1 is a `Type I` clique. For a `Type I` clique κ^*, let $f_1, f_2, f_3, f_4, f_5, f_6$ be the edges in the order of arrival. Now note that if $\kappa = \kappa^*$, it must be true that $c_1 = c(f_1)$ and $c_2 = c(f_1, f_2)$. Hence,

$$\mathbf{E}[X] = \sum_{\kappa^* \in \mathcal{T}_4^1(G)} \mathbf{E}[X \mid \kappa = \kappa^*] \cdot \mathbf{Pr}[\kappa = \kappa^*] = \sum_{\kappa^* \in \mathcal{T}_4^1(G)} \mathbf{Pr}[\kappa = \kappa^*] \cdot m c_1 c_2$$
$$= \sum_{\kappa^* \in \mathcal{T}_4^1(G)} \mathbf{Pr}[\kappa = \kappa^*] \cdot m \cdot c(f_1) \cdot c(f_1, f_2) = \tau_4^1(G),$$

where we have used Lemma 5.1 to establish that $\mathbf{Pr}[\kappa = \kappa^*] = (m \cdot c(f_1) \cdot c(f_1, f_2))^{-1}$. □

By a similar argument, Lemma 5.2 implies the following lemma:

Lemma 5.4 *Let κ_2 be the value that the* `Type II` *neighborhood sampling algorithm keeps after observing m edges. Define a random variable Y to be m^2 if κ_2 is a 4-clique, and 0 otherwise. Then, $\mathbf{E}[Y] = \tau_4^2(G)$.*

Let $\eta = \max\{m\Delta^2, m^2\}$ and recall that $s(\varepsilon, \delta) = \frac{1}{\varepsilon^2}\log(1/\delta)$. Putting these ingredients together, we have the following theorem:

Theorem 5.5 *There is an $O(r)$-space bounded streaming algorithm that observes a graph stream G and returns an (ε, δ)-approximation of the number of 4-cliques in G, provided that $r \geq K \cdot s(\varepsilon, \delta)\frac{\eta}{\tau_4(G)}$ for a fixed constant $K > 0$.*

PROOF. Let X be the average of r independent unbiased estimators from Lemma 5.3 and Y be the average of r independent unbiased estimators from Lemma 5.4. Our algorithm returns $X + Y$ as an estimate for the number of 4-cliques. The correctness of the algorithm, space and accuracy bounds follow from Lemma 5.3 and Lemma 5.4 and applying a Chernoff bound (similarly to Theorem 3.3) on the resulting estimates. We omit details. □

Higher-order cliques: We can further generalize our algorithm to estimate the number of ℓ-cliques.

Theorem 5.6 *There is a $O(r)$-space bounded algorithm that returns an (ε, δ) approximation of the number of ℓ-cliques in a stream, where $\eta_\ell = \max \cup_{\alpha=1}^{\lfloor \ell/2 \rfloor}\{m^\alpha \Delta^{\ell-2\alpha}\}$, provided that $r \geq K \cdot s(\varepsilon, \delta)\frac{\eta_\ell}{\tau_\ell(G)}$ for a fixed constant $K > 0$.*

Similar ideas also lead to the following results for sampling higher-order cliques:

Theorem 5.7 *Assume that ℓ is a constant and $\tau_\ell(G) > 0$. There is an $O\left(\frac{\eta_\ell}{\tau_\ell}\log(1/\delta)\right)$-space bounded algorithm that observes a graph G and returns a random ℓ-clique of G with probability at least $1 - \delta$, where $\eta_\ell = \max \cup_{\alpha=1}^{\lfloor \ell/2 \rfloor}\{m^\alpha \Delta^{\ell-2\alpha}\}$.*

5.2 Sliding Windows

We extend the basic triangle counting algorithm to the setting of sequence-based sliding windows. Here, the graph of interest is restricted to the most recent w edges and we want to estimate the number of triangles in this graph. We show how to implement neighborhood sampling in a sliding-window setting. Recall that the neighborhood sampling algorithm maintains two edges r_1—a randomly chosen edge, and r_2—a randomly chosen edge from $N(r_1)$. In a sliding-window model, one has to ensure that r_1 is an edge chosen uniformly at random from the most recent w edges. This can be done using any algorithm that sampled an element from a sliding window (for instance, using [2]). At time t, let $e_{t-w+1}, \ldots, e_{t-1}, e_t$ denote the last w edges seen. For each edge e_i, we pick a random number $\rho(i)$ chosen uniformly between 0 and 1. We maintain a chain of samples $S = \{e_{\ell_1}, e_{\ell_2} \cdots, e_{\ell_k}\}$ from the current window. The first edge e_{ℓ_1} is the edge such that $\rho(\ell_1) = \min\{\rho(t-w+1), \cdots \rho(t)\}$. For $2 \leq i \leq k$, e_{ℓ_i} is the edge such that $\rho(\ell_i) = \min\{\rho(\ell_{i-1}+1), \cdots \rho(t)\}$. For each $e_{\ell_i} \in S$, we also maintain a random adjacent neighbor r_2^i from $N(e_{\ell_i})$. Note that the second edge can be chosen using standard reservoir sampling, because, if e_{ℓ_i} lies in the current window, any neighbor that arrives after it will also be in the current window. Thus, all of r_2^is also belong to the current window. We chose r_1 to be e_{ℓ_1} and r_2 to r_2^1. When r_1 falls out of window, we remove it from S, and update r_1 to e_{ℓ_2} and r_2 to r_2^2, and so on. This will ensure that r_1 is always a random edge in the current window and r_2 is a random neighbor of r_1 from the current window, and the rest of the analysis follows. Therefore, the expected size of the set S is $\Theta(\log w)$ [2]. Thus, the total expected space used by the algorithm increases by a factor of $O(\log w)$. We arrive at the following theorem:

Theorem 5.8 (Sliding Window) *Let $\gamma(G)$ denote the tangle coefficient of a graph G. Let $0 < \delta, \varepsilon \leq 1$, $r \geq 1$, and $w \geq 1$. In the sequence-based sliding window model with window size w, there is a streaming algorithm using $O(r \log w)$ space that on a sliding-window stream of any graph G, returns an (ε, δ)-approximation to the triangle count in G, provided that $r \geq \frac{48}{\varepsilon^2} \frac{m\gamma(G)}{\tau(G)} \log\left(\frac{1}{\delta}\right)$.*

6. CONCLUSION

In this paper, we presented a new space-efficient algorithm for approximately counting and sampling triangles, and more generally, constant-sized cliques. We showed significant improvements in the space and time complexity over previous work for these fundamental problems. The experimental findings show that even with our improvements, maintaining an approximate triangle count in a graph stream remains compute-intensive: the experiments were CPU-bound even when we were streaming from a laptop's hard drive. It is therefore natural to wonder if our scheme can be parallelized to achieve a higher throughput. In a follow-up work, we show that neighborhood sampling is amendable to parallelization. We have implemented a (provably) cache-efficient multicore parallel algorithm for approximate triangle counting where arbitrarily-ordered edges arrive in bulk [20].

Acknowledgments. This research was in part sponsored by the U.S. Defense Advanced Research Projects Agency (DARPA) under the Social Media in Strategic Communication (SMISC) program, Agreement Number W911NF-12-C-0028. The research of Pavan was sponsored in part by NSF grant 0916797, and of Tirthapura was sponsored in part by NSF grants 0834743 and 0831903. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Defense Advanced Research Projects Agency, National Science Foundation, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

7. REFERENCES

[1] K. J. Ahn, S. Guha, and A. McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proc. ACM Symposium on Principles of Database Systems (PODS)*, pages 5–14, 2012.

[2] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 633–634, 2002.

[3] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 623–632, 2002.

[4] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Proc. ACM Conference on Knowledge Discovery and Data Mining (KDD)*, pages 16–24, 2008.

[5] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting triangles in data streams. In *Proc. ACM Symposium on Principles of Database Systems (PODS)*, pages 253–262, 2006.

[6] L. S. Buriol, G. Frahling, S. Leonardi, and C. Sohler. Estimating clustering indexes in data streams. In *Proc. European Symposium on Algorithms (ESA)*, pages 618–632, 2007.

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

[8] M. Jha, C. Seshadhri, and A. Pinar. From the birthday paradox to a practical sublinear space streaming algorithm for triangle counting. *CoRR*, abs/1212.2264, 2012.

[9] H. Jowhari and M. Ghodsi. New streaming algorithms for counting triangles in graphs. In *Proc. 11th Annual International Conference Computing and Combinatorics (COCOON)*, pages 710–716, 2005.

[10] D. M. Kane, K. Mehlhorn, T. Sauerwald, and H. Sun. Counting arbitrary subgraphs in data streams. In *Proc. International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 598–609, 2012.

[11] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[12] J. Leskovec. Stanford large network dataset collection. http://snap.stanford.edu/data/index.html. Accessed Dec 5, 2012.

[13] M. Manjunath, K. Mehlhorn, K. Panagiotou, and H. Sun. Approximate counting of cycles in streams. In *Proc. European Symposium on Algorithms (ESA)*, pages 677–688, 2011.

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

[15] M. E. J. Newman, D. J. Watts, and S. H. Strogatz. Random graph models of social networks. *PNAS*, 99(1):2566–2572, 2002.

[16] R. Pagh and C. E. Tsourakakis. Colorful triangle counting and a mapreduce implementation. *Inf. Process. Lett.*, 112(7):277–281, 2012.

[17] T. Schank and D. Wagner. Approximating clustering coefficient and transitivity. *Journal of Graph Algorithms and Applications*, 9(2):265–275, 2005.

[18] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Workshop on Experimental and Efficient Algorithms (WEA)*, pages 606–609, 2005.

[19] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proc. 20th International Conference on World Wide Web (WWW)*, pages 607–614, 2011.

[20] K. Tangwongsan, A. Pavan, and S. Tirthapura. Parallel triangle counting in massive streaming graphs. In *CIKM*, 2013.

[21] C. E. Tsourakakis, P. Drineas, E. Michelakis, I. Koutis, and C. Faloutsos. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Netw. Analys. Mining*, 1(2):75–81, 2011.

[22] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 837–846, 2009.

[23] S. Wasserman and K. Faust. *Social Network Analysis*. Cambridge University Press, 1994.

[24] D. J. Watts and S. H. Strogatz. Collective dynamics of "small-world" networks. *Nature*, 393:440–442, 1998.