**RC25339 (WAT1212-086) December 12, 2012**
**Computer Science**

# IBM Research Report

## Counting and Sampling Triangles from a Graph Stream

**A. Pavan[1], Kanat Tangwongsan[2], Srikanta Tirthapura[1], Kun-Lung Wu[2]**

[1]Iowa State University

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 208
Yorktown Heights, NY 10598
USA

# Counting and Sampling Triangles from a Graph Stream*

A. Pavan* Kanat Tangwongsan† Srikanta Tirthapura* Kun-Lung Wu†

*Iowa State University* and IBM Research†*

**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. When compared with 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.

**Keywords:** graph streams, data streams, triangle counting, cliques, social networks, sampling

## 1 Introduction

Triangle counting has emerged as an important building block in the study of social networks [WF94, New03], identifying thematic structures of networks [EM02], spam and fraud detection [BBCG08], link classification and recommendation [TDM+11], and more. In these applications, streaming algorithms provide an attractive option for real-time processing of live data; they also benefit the analysis of large disk-resident graph data, allowing computations in one or a small number of passes over the data.

In this paper, we address the question of counting and sampling triangles, as well as complete subgraphs, in the adjacency stream model. 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 random triangle from the set of all triangles in a graph; and
(3) counting and sampling cliques of 4 or more vertices ($K_4, K_5, \dots$).

In the *adjacency stream model* [BYKS02, JG05, BFL+06], a graph $G = (V, E)$ is presented as a stream of edges $\mathcal{S} = \langle e_1, e_2, e_3, \dots, e_{|E|}\rangle$. In this notation, $e_i$ denotes the $i$-th edge in the stream order, which is arbitrary and potentially chosen by an adversary. For this graph, let $n = |V|$, $m = |E|$, $\mathcal{T}(G)$ denote set of all triangles, and $\tau(G)$ denote the number of triangles (i.e., $\tau(G) = |\mathcal{T}(G)|$). We will assume that the input graph is simple (no parallel edges, no self-loops, and no multiple copies of the same edge).

Our algorithms are randomized and provide the following notion of probabilistic guarantees: for parameters $\varepsilon, \delta \in [0, 1]$, an $(\varepsilon, \delta)$-approximation for a quantity $X$ is a random variable $\hat{X}$ such that $|\hat{X} - X| \le \varepsilon X$ 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 from a graph stream. Neighborhood sampling is a multi-level inductive random sampling procedure, where a random edge in the stream is first sampled, and then in subsequent steps, an edge with an endpoint in common to the sampled edge(s) is sampled. We show that this simple technique leads to significant improvements in the space complexity of triangle counting and related problems.

— **Counting and Sampling Triangles:** Using neighborhood sampling, we present a one-pass streaming algorithm for triangle counting and triangle sampling. The space complexity of triangle counting is $O(s(\varepsilon, \delta)m\Delta/\tau(G))$ and triangle sampling is $O(m\Delta/\tau(G))$, where $\Delta$ is the maximum degree of any vertex. This significantly improves upon

*E-mail addresses: pavan@cs.iastate.edu, ktangwo@us.ibm.com, snt@iastate.edu, klwu@us.ibm.com

| | Space Complexity | Time Per Element |
|---|---|---|
| Buriol *et al.* [BFL+06] | $O\left(s(\varepsilon,\delta)\frac{mn}{\tau(G)}\right)$ | $O\left(1+s(\varepsilon,\delta)\frac{n\log m}{\tau(G)}\right)$ |
| Jowhari and Ghodsi [JG05] | $s_2=O\left(s(\varepsilon,\delta)\frac{m\Delta^2}{\tau(G)}\right)$ | $\Theta(s_2)$ |
| This Paper (amortized with bulk processing) | $O\left(s(\varepsilon,\delta)\frac{m\Delta}{\tau(G)}\right)$ | $\Theta(1)$ |

Table 1: Performance of streaming algorithms for counting triangles in a graph. The space complexity is expressed in terms of the number of words used, and assumes that vertex identifiers, and counters, including the number of edges and vertices, can be stored in a constant number of words.

prior algorithms for the same problem, as shown in Table 1. We provide a sharper bound for the space complexity of our algorithm in terms of the "tangle coefficient" of the graph, which we define in our analysis. While in this worst case, this results in the space bound we have stated above, in typical cases, it can be smaller.

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

— **Counting and Sampling Cliques:** We extend neighborhood sampling to the problem of sampling and counting the set of all cliques of size $\ell$ in 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 $\ell$-cliques are presented in Section 3. To our knowledge, this is the best space complexity for counting the number of $\ell$-cliques in a graph in the streaming model and improves on prior work due to Kane *et al.* [KMSS12].

— **Experiments:** Our experiments with real-world large 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 (and hence a few MB of memory). This experiment was run on a laptop, with an implementation that did not use parallelism. Our experiments show that in order to get good estimates, far fewer than $\Theta(s(\varepsilon,\delta)m\Delta/\tau(G))$ estimators may be necessary.

**Prior Work:** For triangle counting in adjacency streams, Bar-Yossef *et al.* [BYKS02] present the first algorithms based on reductions to the problem of computing the zeroth and second frequency moments of appropriately defined streams, derived from the edge stream of the graph. Their algorithm on the adjacency stream model takes space $s = O(\frac{s(\varepsilon,\delta)}{\varepsilon}\cdot(mn/\tau(G))^3)$ and $\mathrm{poly}(s)$ time per item. They also present a lower bound showing that in general, the worst case streaming complexity of approximating $\tau(G)$ is $\Omega(n^2)$.

The space and time bounds were subsequently improved. Jowhari and Ghodsi [JG05] present a one-pass streaming algorithm that uses space and per-edge processing time of $O(s(\varepsilon,\delta)m\Delta^2/\tau(G))$. Our algorithm significantly improves upon this algorithm in both space and time complexity. Note that $\Delta$ for large graphs can be significant in practice; for instance, the Orkut graph has a maximum degree of greater than 66,000. They also present a three pass streaming algorithm with space and per-edge processing time of $O\left(s(\varepsilon,\delta)\cdot(1+T_2(G)/\tau(G))\right)$, where $T_2(G)$ is the number of vertex triples in the graph with exactly two edges connecting them. Later, Buriol *et al.* [BFL+06] present algorithms for counting the number of triangles with space complexity $O(s(\varepsilon,\delta)mn/\tau(G))$. If the maximum degree $\Delta$ is small compared with $n$, our algorithm substantially improves upon theirs in terms of space. Another difference is that the algorithm of [BFL+06] needs to know the set of vertices in 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.

On counting cliques, Kane *et al.* [KMSS12] present estimators for the number of occurrences of an arbitrary subgraph $H$ in the stream. When applied to counting cliques on $\ell$ vertices in a graph, their space complexity is $O\left(s(\varepsilon,\delta)\cdot m^{\binom{\ell}{2}}/\tau_\ell^2(G)\right)$ 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.

**Related Work:** Manjunath *et al.* [MMPS11] 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 bound can be much larger, especially for graphs with a small number of triangles. Recent work on graph sketches by Ahn, Guha, and McGregor [AGM12] also yield algorithms for counting triangles in a graph, with space complexity, whose dependence on $m$ and $n$ is the same as in [BFL$^+$06].

Buriol *et al.* [BFLS07] present algorithms to estimate the clustering index of a graph 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. We note that in the incidence streams model, counting triangles is an easier problem, and there are streaming algorithms [BFL$^+$06] that use space $O\left(s(\varepsilon,\delta)\left(1+T_2(G)/\tau(G)\right)\right)$. In this work, we focus on the adjacency streams model which is a more realistic model for real-time analytics on an evolving graph.

Becchetti *et al.* [BBCG08] 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 of triangles in a graph in the non-streaming setting, for example [SV11, TKMF09]. We do not attempt to survey this literature. An experimental study of algorithms for counting and listing triangles in a graph is presented in [SW05].

**Roadmap:** In Section 2, we present our technique and its use in counting and sampling triangles, followed by extensions to counting and sampling cliques in Section 3, experimental results in Section 4 and extensions to sliding windows in Section 5.

**Preliminaries:** For an edge $e$, let $V(e)$ denotes 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 $N(e_i)$. For $\ell \geq 4$, let $\mathcal{T}_\ell(G)$ denote the set of all $\ell$-cliques of graph $G$, and $\tau_\ell(G)$ the number of $\ell$-cliques. 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 $\texttt{coin}(p)$ which returns heads with probability $p$. We assume this procedure takes constant time.

# 2 Sampling and Counting Triangles

In this section, we present algorithms to sample and count triangles. We begin by describing *neighborhood sampling*, a basic method upon which we build an algorithm for counting triangles (Section 2.2), an efficient implementation for bulk processing (Section 2.3), and an algorithm for sampling triangles (Section 2.4).

## 2.1 Neighborhood Sampling Algorithm for Triangles

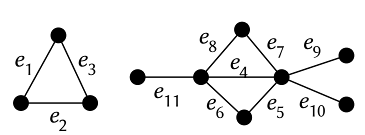


Figure 1: An example graph, where the edges arrive in order $e_1, e_2, \ldots$ in the stream, forming triangles $t_1 = \{e_1, e_2, e_3\}$ and $t_2 = \{e_4, e_5, e_6\}$.

**Overview.** The algorithm first samples a random edge, say $r_1$, from the edge stream; this can be maintained using reservoir sampling. It then samples a random edge from the set of edges in the stream 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. Once such an edge, say $r_2$, has been found, a potential triangle $t$ made up of the two edges $r_1$ and $r_2$ is implicitly defined, and the algorithm waits for the third edge of $t$ to appear in the stream and complete the potential triangle.

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 an algorithm for adjacency streams due to Buriol *et al.* [BFL+06], which also employs random sampling. Like ours, their algorithm 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.

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

- Edge $r_1$, sampled uniformly from among all edges so far. We call this the "level 1 edge".
- Edge $r_2$, sampled uniformly from among all edges in $N(r_1)$, i.e., those edges in the graph stream that are adjacent to $r_1$ and come after $r_1$. We call this the "level 2 edge".
- Counter $c$, equal to $c(r_1) = |N(r_1)|$, i.e, the number of edges that are adjacent to $r_1$ and have appeared after $r_1$.
- A triangle $t$ that is potentially a sample

Presented in Algorithm 1 is the neighborhood sampling algorithm, which is used to maintain the state. Algorithms 2 and 3 build on this algorithm to solve triangle counting and triangle sampling, respectively.

**Algorithm 1:** Algorithm NSAMP-TRIANGLE

```
Initialization: Set r1, r2, t to φ, and c to 0.
Upon getting edge e_i, i ≥ 1 ;
begin
    if coin(1/i) = "head" then
        // e_i is the new sampled edge at level 1.
        r1 ← e_i;
        r2 ← φ; c ← 0; t ← φ;
    else
        if e_i is adjacent to r1 then
            c ← c + 1;
            if coin(1/c) = "head" then
                // e_i is the new sampled edge at level 2.
                r2 ← e_i;
                t ← φ;
            else
                if e_i forms a triangle with r1 and r2 then
                    t ← {r1, r2, e_i}
```

**Lemma 2.1** *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 the $t^*$'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 the neighborhood sampling algorithm produces $t^*$ at the end 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

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

■

## 2.2 Counting Triangles in a Graph

The neighborhood sampling algorithm produces a triangle $t$ with probability $\frac{1}{m \cdot C(t)}$. We show in Algorithm 2 how to build an unbiased estimator from this sampling algorithm. The basic idea is to output a value which counterbiases the probability so that in expectation, the contribution of a triangle is exactly 1. This is easy to achieve because we know both $C(t)$ and the number of edges $m$. The following lemma formally shows that Algorithm 2 gives an unbiased estimator for the number of triangles.

**Algorithm 2:** COUNT-TRIANGLE

Run Algorithm 1 and let $t$ and $c$ be the variables it maintains.
Return $c \cdot m$ if $t$ is defined or 0 otherwise.

**Lemma 2.2** *Let $X$ denote the return value of Algorithm 2 after graph $G$ has been observed. Then* $\mathbf{E}[X] = \tau(G)$.

*Proof:* By Lemma 2.1, the probability that Algorithm 2 samples a particular triangle $t^*$ is precisely $\mathbf{Pr}[t = t^*] = 1/mC(t^*)$. Further, the counting algorithm returns $mC(t^*)$ if $t = t^*$, and 0 if $t$ is empty. Therefore,

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

■

To obtain an accurate estimate, we apply a standard technique which runs multiple copies of an unbiased estimator in parallel and outputs the average. In the following lemma, we give a bound on the number of copies needed to achieve an $(\varepsilon, \delta)$-approximation. In Appendix A, we give a sharper space bound in terms of a measure we term "tangle index," which helps to explain why we usually need less space in practice.

**Theorem 2.3** *For any graph $G$, for parameters $0 \leq \delta, \varepsilon \leq 1$, there is a streaming algorithm that observes the edges of $G$ in an arbitrary order, and returns an $(\varepsilon, \delta)$-approximation for the number of triangles in $G$ using space* $O\left(\frac{1}{\varepsilon^2} \frac{m\Delta}{\tau(G)} \log\left(\frac{1}{\delta}\right)\right)$.

The proof follows from standard concentration bounds and Lemma 2.2; we defer it to Appendix C

## 2.3 A Nearly-Linear Time Algorithm for Triangle Counting

Our discussion thus far directly leads to a simple $O(mr)$-time implementation, where $r$ is the number of copies of estimators we decide to maintain, but this leaves much to be desired in terms of performance. For large graphs, we would ideally like the algorithm to take time linear in the number of edges and the number of estimators.

The result in this section is motivated by the observation that in many real-world applications, the algorithm receives edges in bulk (e.g., block reads from disk, HTTP `PUT` requests). We show that it is possible to achieve significantly better performance through bulk processing. The basic intuition is that with bulk processing, we update the estimators much less often. As one example, by processing in bulks of $\Theta(r)$ edges, the total running time becomes $\Theta(m + r)$ using $O(r)$ space—at the expense of a constant factor more space, we are able to achieve $O(m + r)$ time bound as opposed to $O(mr)$. More precisely, we obtain the following bounds:

**Theorem 2.4** *Let $w$ be a block size and $r$ be the number of estimators. There is an algorithm for triangle counting that on input a graph with $m$ edges, runs in total time $O(m(1 + r/w))$ using space $O(r + w)$.*

We obtain this bound by developing a routine to process a block $B = \langle b_1, \ldots, b_{|B|}\rangle$ of edges in roughly $O(|B|+r)$ time. Our goal is to advance the states of all $r$ estimators to the point after including the block $B$, simulating the effects of playing these edges one by one faithfully. In the interest of space, we can only highlight a few key ingredients here and give detailed descriptions in Appendix B.

The challenge in fastforwarding over the block $B$ lies in maintaining level-2 edges, finding the completed triangles, and keeping track of the counters; maintaining the level-1 edges is relatively straightforward. There are two important ingredients in our solution: The first ingredient is a data structure that maintains the degrees of about $O(r)$ nodes; these degrees only count the edges in $B$. More specifically, let $R$ be the set of the endpoints of all level-1 edges. As we make a pass through $b_1, b_2, \ldots, b_{|B|}$, we maintain for each $v \in R$, a counter $\lambda_v$ which contains the number of edges in this block so far that is incident on $v$. This is useful for updating the counter $c$ and for determining which level-2 edge to pick. As an example, for each estimator, its $c$ value is increased by the difference between the $\lambda$ values of the endpoints when its level-1 edge is encountered and the $\lambda$ values at the end of the pass. This whole computation requires one pass and $O(|B| + |R|) = O(|B| + r)$ time and space using a (hash) table which maps level-1 edges to their corresponding estimator states and a (hash) table storing the degree values.

The second ingredient is the observation that the event "*the edge that increments the $\lambda_x$ to a certain value $z$*" uniquely identifies an edge in $B$. Given the degree information obtained previously, we can describe our random selection of a level-2 edge in this form—and by keeping a (hash) table mapping (vertex, desired-restricted-degree) to the corresponding estimator, a second pass through the data allows us to find level-2 replacement edges. Since the table contains at most $O(r)$ entries, the space and time usage for this pass is $O(|B| + r)$.

Finally, in the second pass, we also keep another (hash) table for queries of the form: "*if an edge $e$ is seen, we have a complete triangle for estimator $X$.*" This table is updated as we discover a new level-2 edge. The size of this table is, again, at most $O(r)$. Hence, each block can be processed in *two* passes in $O(|B| + r)$ space and time.

## 2.4 Sampling a Triangle

We now present an algorithm that picks a triangle of the graph stream uniformly at random. As discussed previously, the neighborhood sampling algorithm does not necessarily return a uniform sample. To fix it, we resort to a simple normalization procedure.

**Algorithm 3:** Algorithm SAMPLE-TRIANGLE: a query arrives for a random triangle

Run Algorithm 1 and let $t$ and $c$ be the variables it maintains.
Return the triangle $t$ with probability $c/2\Delta$; else return "Failed".

**Lemma 2.5** *If a query for a random triangle is posed after observing a graph $G$, each triangle in $\mathcal{T}(G)$ is equally likely to be returned by Algorithm 3. Further, the probability that a triangle is returned by Algorithm 3 is $\frac{\tau(G)}{2m\Delta}$, where $\Delta$ is the maximum degree of any vertex in $G$.*

*Proof:* The proof is similar to the proof of Lemma 2.2. By 2.1, the probability that Algorithm 2 samples a particular triangle $t^*$ is precisely $\mathbf{Pr}[t = t^*] = 1/mC(t^*)$. Further, note that $C(t^*) \leq 2\Delta$.Therefore, if the triangle $t^*$ was returned by Algorithm 1, 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 $\frac{c}{2\Delta} \leq 1$. 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 $\frac{\tau(G)}{2m\Delta}$. ∎

The following establishes the main theorem on triangle sampling.

**Theorem 2.6** *Assuming that $\tau(G) > 0$, for a parameter $\delta$, $0 < \delta < 1$, there is a streaming algorithm with space complexity $\Theta\left(\frac{m\Delta\log(1/\delta)}{\tau(G)}\right)$ that observes graph $G$ as an adjacency stream and returns a random triangle in $G$ with probability at least $1 - \delta$.*

*Proof:* Let $\alpha = \left(\frac{2m\Delta\ln(1/\delta)}{\tau(G)}\right)$. The streaming algorithm runs $\alpha$ copies of Algorithm 3. This algorithm fails to return a triangle only if all these $\alpha$ copies fail to produce a triangle; otherwise, it can return any of them. Therefore, the overall failure probability is at most $(1-\beta)^\alpha \le e^{-\beta\alpha} \le \delta$, where $\beta = \frac{\tau(G)}{2m\Delta}$ is the success probability in Lemma 2.5.The space complexity follows since the space taken by each copy of the algorithm is a constant number of words. ■

**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 [BFL+06]*? Note that $T_2(G)$ 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.

**Lemma 2.7** *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 $\omega(1 + T_2(G)/\tau(G))$.*

The proof uses a reduction from the index problem from communication complexity and is deferred to Appendix C in the interest of space. We note that the same proof technique also applies to triangle sampling.

# 3 Counting and Sampling Cliques

We extend the neighborhood sampling technique to counting and sampling from the set of cliques of size greater than 3. 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 $G$. We partition $\mathcal{T}_4(G)$ into two classes, `Type I` cliques and `Type II` cliques, defined based on the order in which the edges of the clique appear in the stream. Let $\kappa^*$ be a 4-clique and let $f_1, f_2, \cdots, f_6$ be the order in which the edges of $\kappa^*$ appear in the stream. We say that $\kappa^*$ is a `Type I` clique if $f_2$ and $f_1$ share a common vertex, otherwise $\kappa^*$ is a `Type II` clique. Clearly, every clique in $\mathcal{T}_4(G)$ is either a `Type I` clique or a `Type II` clique. Let $\mathcal{T}_4^1(G)$ denote the set of `Type I` cliques in $G$ and $\mathcal{T}_4^2(G)$ denote the set of `Type II` cliques; let $\tau_4^1(g)$ and $\tau_4^2(G)$ denote the corresponding cardinalities. We present two estimators, one for $\tau_4^1(G)$ and the other for $\tau_4^2(G)$. The sum of the two estimators will be an estimator for $\tau_4(G)$. Due to space constraints all proofs of this section appear in Appendix D.

## 3.1 Neighborhood Sampling for 4-Cliques

Our first algorithm, described in Algorithm 4, is concerned with sampling `Type I` cliques. This algorithm maintains two sets, a sample of up to three edges, $r_1, r_2, r_3$, and a potential clique, $\kappa_1$. Each of the $r_i$s is also included in $\kappa_1$, which could also contain additional edges required for completing the clique.

**Lemma 3.1** *Consider a `Type I` clique $\kappa^*$ 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 $\kappa_1$ equals $\kappa^*$ is $\frac{1}{m \cdot c(f_1) \cdot c(f_1, f_2)}$.*

Next we describe a neighborhood sampling algorithm, Algorithm 5 that processes `Type II` cliques. This algorithm is simpler that Algorithm 4, and maintains a sample consisting of two edges $r_1$ and $r_2$, and a potential clique $\kappa_2$.

**Lemma 3.2** *Consider a `Type II` clique $\kappa^*$. After Algorithm 5 has processed the entire graph, the probability that $\kappa_2$ is equal to $\kappa^*$ is $\frac{1}{m^2}$*

**Counting 4-cliques.** We obtain following results for counting 4-cliques. Let $\eta = \max\{m\Delta^2, m^2\}$.

**Algorithm 4:** NSAMP-Type I

**Initialization:** Set $r_1, r_2, r_3$ to $\phi$, $\kappa_1$ 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 $\kappa_1$.

**Algorithm 5:** NSAMP-Type II

**Initialization:** $r_1 = \phi, r_2 = \phi$; $\kappa_2 = \emptyset$.
**Edge** $e_i$ **arrives**;
If (`coin`$(1/i)$ = "head") set $r_1 \leftarrow e_i$;
If (`coin`$(1/i)$ = "head") set $r_2 \leftarrow e_i$;
**if** *(Neither coin returned "head")* **then**
  **if** $e_i$ *is adjacent to both* $r_1$ *and* $r_2$ **then**
    $\kappa_2 = \kappa_2 \cup \{e_i\}$;
**else**
  **if** $V(r_1) \cap V(r_2) = \emptyset$ **then**
    $\kappa_2 = \{r_1, r_2\}$; **else** $\kappa_2 = \{r_1\}$;

**Theorem 3.3** *There is a* $O\left(s(\varepsilon,\delta)\frac{\eta}{\tau_4(G)}\right)$*-space bounded streaming algorithm that observes a graph* $G$ *and returns a* $(\varepsilon,\delta)$ *approximation of the number of* $4$*-cliques in* $G$*.*

**Theorem 3.4** *There is a* $O\left(s(\varepsilon,\delta)\frac{\eta_\ell}{\tau_\ell(G)}\right)$*-space bounded algorithm that returns an* $(\varepsilon,\delta)$ *approximation of the number of* $\ell$*-cliques in a stream, where* $\eta_\ell = \max \cup_{\alpha=1}^{\lfloor \ell/2 \rfloor}\{m^\alpha \Delta^{\ell-2\alpha}\}$*.*

For instance, if $\ell$ was even, and $\Delta = O(\sqrt{m})$, then the above algorithm for $\ell$-cliques takes space $O\left(s(\varepsilon,\delta)\frac{m^{\ell/2}}{\tau_\ell(G)}\right)$

**Sampling 4-Cliques.** We show the following analogous results for sampling cliques.

**Theorem 3.5** *Assume that* $\tau_4(G) > 0$*, and let* $\eta = \max\{m\Delta^2, m^2\}$*. For every* $\delta$*,* $0 < \delta < 1$*, there is a streaming algorithm with space complexity* $O\left(\frac{\eta \log(1/\delta)}{\tau_4(G)}\right)$ *that observes a graph* $G$ *with* $m$ *edges and returns a random 4-clique in* $G$ *with probability at least* $1-\delta$*.*

**Theorem 3.6** *Assume that* $\ell$ *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 an* $\ell$*-clique from the graph, chosen uniformly at random from the set of all* $\ell$*-cliques in the graph, where* $\eta_\ell = \max \cup_{\alpha=1}^{\lfloor \ell/2 \rfloor}\{m^\alpha \Delta^{\ell-2\alpha}\}$*.*

# 4 Experiments

We empirically study the proposed triangle counting algorithm. We implemented the version of the algorithm which processes edges in batches, and also the other state-of-the-art algorithms due to [BFL+06, JG05].

## 4.1 Experimental Setup

**Datasets and Environment.** Our study uses a collection of popular social media graphs, obtained from the publicly available data provided by the SNAP project at Stanford [Les]. We present a summary of these datasets in Table 2. We remark that while these datasets stem from social media, our algorithm does not assume any special property about them. Our experiments were performed on a 2.2 Ghz Intel Core i7 laptop machine with 8GB of memory, but our experiments used no more than a few MB of RAM. 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`, which has enough resolution for our experiments. Our prototype implementation uses GNU's STL implementation of collections, including `unordered_map` for hash maps; we did not attempt to optimize our code.

Our algorithm is randomized and may behave differently on different runs. Thus, for robustness, we perform *five* trials and gather the following statistics: **(1)** the minimum, mean, and maximum 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 error, which we believe to give an accurate picture of how well the algorithm performs. For completeness, we also report the min/max deviation values, but we note that as we perform more trials, the minimum becomes smaller and the maximum becomes larger, so they are not robust.

| **Dataset** | $n$ | $m$ | $\Delta$ | $\tau$ | $m\Delta/\tau$ |
|---|---|---|---|---|---|
| Amazon | 334,863 | 925,872 | 1,098 | 667,129 | 1,523.85 |
| DBLP | 317,080 | 1,049,866 | 686 | 2,224,385 | 323.78 |
| Youtube | 1,134,890 | 2,987,624 | 57,508 | 3,056,386 | 56,214.20 |
| LiveJournal | 3,997,962 | 34,681,189 | 29,630 | 177,820,130 | 5,778.89 |
| Orkut | 3,072,441 | 117,185,083 | 66,626 | 633,319,568 | 12,328.02 |

Table 2: A summary of the datasets used in our experiments, showing for every dataset the number of nodes ($n$), the number of edges ($m$), the maximum degree ($\Delta$), the number of triangles in the graph ($\tau$), and the ratio $m\Delta/\tau$.

**Baseline.** We implemented the state-of-the-art algorithms for adjacency streams due to Buriol et al. [BFL+06], and Jowhari and Ghodsi [JG05]. Our implementation of Buriol et al.'s algorithm follows the description of the optimized version in their paper, which achieves $O(m + r)$ running time for $m$ edges and $r$ estimators, through certain approximations. However, we were unable to obtain meaningful results from these algorithms on the datasets we consider: The estimators due to Buriol et al.fail 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 (see Section 2.1 for a related discussion); this behavior is consistent with Buriol et al.'s findings for their adjacency-stream algorithm. In the case of Jowhari-Ghodsi's algorithm, its $O(mr)$ running time is too slow even on a modest dataset. Instead, we directly compare our results with the true count and focus our study on the scalability of the approach as the graph size and the number of estimators increase. For most datasets, the exact triangle count is provided by the source; in other cases, we compute the exact count using an algorithm developed as part of the Problem-Based Benchmark Suite [SBF+12].

## 4.2 Accuracy

The first set of experiments aims to study the accuracy 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 $\Delta$ and the number of triangles $\tau$.

| **Dataset** | $r = 1\text{K}$ | | $r = 128\text{K}$ | | $r = 1\text{M}$ | | 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 |

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.

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 $r$ $(1024, 131072, 1047576)$. 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. 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 $9.78$ million, but we already get this accuracy using $1$ million estimators.

## 4.3 Performance

In the second set of experiments, we selected two graphs—Youtube and LiveJournal—to study runtime performance and accuracy in more detail. The goal of these experiments is to understand how the number of estimators affects the running time, as well as the accuracy.

First, we consider running time. It is clear from Figure 2 that 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 $r$.

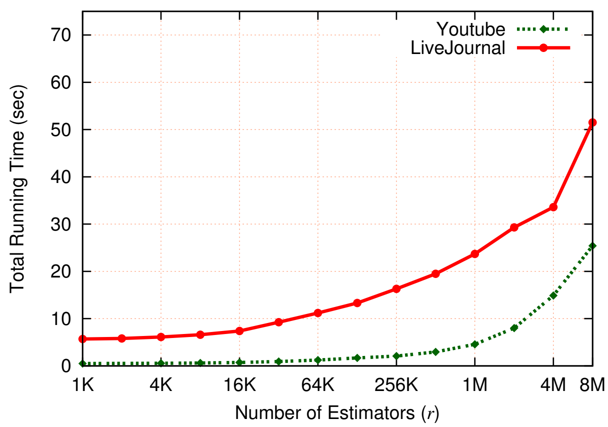


Figure 2: Running time (**in seconds**) as the number of estimators $r$ is varied ($r = 1\text{K}, 2\text{K}, \ldots, 8\text{M}$). The $x$-axis is in log scale.

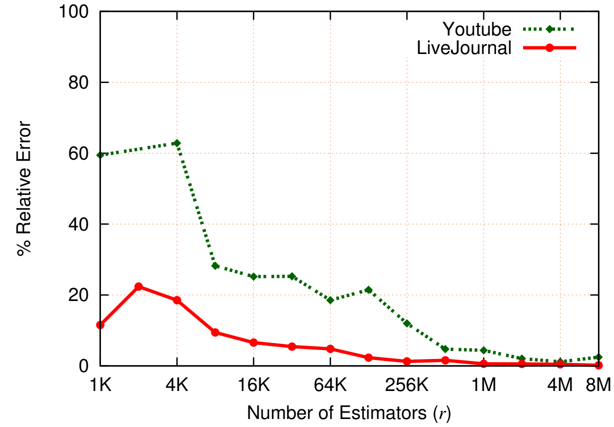


Figure 3: Mean deviation (**in percentage**) of our estimates as the number of estimators $r$ is varied ($r = 1\text{K}, 2\text{K}, \ldots, 8\text{M}$). The $x$-axis is in log scale.

As for accuracy, $(\varepsilon, \delta)$-type approximation algorithms are difficult to experiment with, in general: the observed error could stem from the controlled error term $\varepsilon$ or it could stem from the unlikely event that happens with probability $\delta$ where we have no control over the error. The general trend from Figure 3 is that the mean deviation decreases as we increase the number of estimators.

# 5 Extensions

Our basic algorithms for sampling and counting can be extended to the case the case of sliding windows. For simplicity, we consider sequence-based sliding window, where the scope of relevant data is restricted to the $w$ most recent edges. Thus, we would like to estimate the number of triangles in the graph induced by the most recent $w$ edges. There are several algorithms to sample an element from a sliding window [BOZ09, BDM02, Haa, ZLY+05, GL08] that can be adapted for use here. For simplicity, we use the algorithm from [BDM02]. Recall that the neighborhood sampling algorithm maintains two edges $r_1$—a randomly chosen edge, and $r_2$—a randomly chosen edge from $N(r_1)$. At time instance $t$, let $\langle e_{t-w+1}, e_{t-w+2}, \cdots, e_{t-1}, e_t\rangle$ 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 $e_{\ell_1}$ is the edge such that $\rho(\ell_1) = \min\{\rho(t-w+1), \cdots \rho(t)\}$. For $2 \le i \le k$, $e_{\ell_i}$ 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 $e_{\ell_i}$ lies in the current window, any neighbor that arrives after it will also be in the current window. Thus all of $r_2^i$s also belong to the current window. We chose $r_1$ to be $e_{\ell_1}$ and $r_2$ to $r_2^1$. When $r_1$ falls out of window, we remove it from $S$, and update $r_1$ to $e_{\ell_2}$ 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. It is easy to see that that the expected size of the set $S$ is $\Theta(\log w)$ [BDM02]. Thus, the total expected space used by the algorithm increases by a factor of $\log w$.

# 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 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 or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

# References


[AGM12] Kook Jin Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Proc. ACM Symposium on Principles of Database Systems (PODS)*, pages 5–14, 2012. 3

[BBCG08] 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. 1, 3

[BDM02] Brian Babcock, Mayur Datar, and Rajeev Motwani. Sampling from a moving window over streaming data. In *Proc. ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 633–634, 2002. 11

[BFL+06] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *Proc. ACM Symposium on Principles of Database Systems (PODS)*, pages 253–262, 2006. 1, 2, 3, 4, 7, 9, 10

[BFLS07] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, and Christian Sohler. Estimating clustering indexes in data streams. In *Proc. European Symposium on Algorithms (ESA)*, pages 618–632, 2007. 3

[BOZ09] Vladimir Braverman, Rafail Ostrovsky, and Carlo Zaniolo. Optimal sampling from sliding windows. In *PODS*, pages 147–156, 2009. 11

[BYKS02] Ziv Bar-Yossef, Ravi 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. 1, 2

[EM02] Jean-Pierre Eckmann and Elisha 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. 1

[GL08] Rainer Gemulla and Wolfgang Lehner. Sampling time-based sliding windows in bounded space. In *SIGMOD Conference*, pages 379–392, 2008. 11

[Haa] P. J. Haas. Data stream sampling: Basic techniques and results. In M. Garofalakis, J. Gehrke, and R. Rastogi, editors, *Data Stream Management: Processing High Speed Data Streams*, pages 223–233. Springer. 11

[JG05] Hossein Jowhari and Mohammad Ghodsi. New streaming algorithms for counting triangles in graphs. In *Proc. 11th Annual International Conference Computing and Combinatorics (COCOON)*, pages 710–716, 2005. 1, 2, 9, 10

[KMSS12] Daniel M. Kane, Kurt Mehlhorn, Thomas Sauerwald, and He Sun. Counting arbitrary subgraphs in data streams. In *Proc. International Colloquium on Automata, Languages, and Programming (ICALP)*, pages 598–609, 2012. 2

[KN97] Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, 1997. 15

[Les] Jure Leskovec. Stanford large network dataset collection. `http://snap.stanford.edu/data/index.html`. Accessed Dec 5, 2012. 9

[MMPS11] Madhusudan Manjunath, Kurt Mehlhorn, Konstantinos Panagiotou, and He Sun. Approximate counting of cycles in streams. In *Proc. European Symposium on Algorithms (ESA)*, pages 677–688, 2011. 3

[New03] M. E. J. Newman. The structure and function of complex networks. *SIAM REVIEW*, 45:167–256, 2003. 1

[SBF+12] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. Brief announcement: the problem based benchmark suite. In *Proc. ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 68–70, 2012. 10

[SV11] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *Proc. 20th International Conference on World Wide Web (WWW)*, pages 607–614, 2011. 3

[SW05] Thomas Schank and Dorothea 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. 3

[TDM+11] Charalampos E. Tsourakakis, Petros Drineas, Eirinaios Michelakis, Ioannis Koutis, and Christos Faloutsos. Spectral counting of triangles via element-wise sparsification and triangle-based link recommendation. *Social Netw. Analys. Mining*, 1(2):75–81, 2011. 1

[TKMF09] Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos 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. 3

[WF94] S. Wasserman and K. Faust. *Social Network Analysis*. Cambridage University Press, 1994. 1

[ZLY+05] L. Zhang, Z. Li, M. Yu, Y. Wang, and Y. Jiang. Random sampling algorithms for sliding windows over data streams. In *11th Joint International Computer Conference,*, pages 572–575, 2005. 11

# A Appendix: Improved Space Bound For Triangle Counting

In this section, we prove a sharper bound on the space needed to obtain an accurate estimate. We will define a measure that captures in the amount of interaction between triangles and non-triangle edges in the graph. The *tangle coefficient* of a graph $G$, denoted by $\gamma$ is given by

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

This is equivalent to $\gamma = \frac{1}{\tau(G)} \sum_{e \in E} c(e)s(e)$, 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 (we provide some discussions about the tangle coefficient after that):

**Theorem A.1 (Improved Triangle Counting)** *Let $\gamma$ be the tangle coefficient of $G$. There is an $(\varepsilon, \delta)$-approximation to the triangle counting problem that requires $O(1/\varepsilon^2 \cdot m\gamma/\tau(G) \cdot \log(\frac{1}{\delta}))$.*

*Proof:* (Sketch) Each estimator has variance $m \sum_{t \in \mathcal{T}(G)} C(t) = m\tau(G)\gamma$. 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

$$\mathbf{Pr}\Big[|Y - \mathbf{E}[Y]\,| > \varepsilon \cdot \tau(G)\Big] \leq \frac{1}{4}.$$

To boost the success probability up to $1 - \delta$, we will run $\beta = 12\ln(1/\delta)$ independent copies of $Y$ estimators and take the median. Note that our median estimator fails to produce an $\varepsilon$-approximation only if more than $\beta/2$ fails to produce an $\varepsilon$-approximation. In expectation, the number of "failed" estimators is at most $\beta/4$. Therefore, by a standard Chernoff bound, we fail with probability at most

$$\mathbf{Pr}[\textsf{FAILED}] \leq e^{-\frac{1^2(\beta/4)}{3}} = \delta,$$

proving our final estimate is an $(\varepsilon, \delta)$-approximation using space at most $\alpha\beta$. ∎

Clearly, $\gamma$ is at most $2\Delta$, recovering the original bound we proved. We can gain more understanding of the quantity $\gamma$ 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 $\gamma$ is the number of edges that incident on $e$ that comes after it. In this view, $\gamma$ 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.

# B Bulk-Processing Edges

Our algorithms take advantage of a procedure `unif` for generating a uniform random number from the interval $[0, 1]$ as well as a procedure `randInt`$(n)$ for drawing a number uniformly at random from $\{0, \ldots, n-1\}$. We assume these take constant time.

We now give detailed descriptions; the bookkeeping, though conceptually simple, turns out to be quite messy. For a block $B$, computing the level-1 edge after incorporating $B$ is straightforward. Let $i_B$ be the total number of edges so far (including the edges in $B$). An edge in $B$ will replace an existing level-1 edge with probability $b/i_B$—and when this happens, each edge in $B$ has an equal probability of being a replacement edge. Thus, with one call to `unif`, we can compute the level-1 edge after incoporating $B$, so in $O(r)$ time, we can determine the level-1 edges for all $r$ estimators we maintain. This can be further optimized by noticing that in later stages, the number of estimators that end up updating their level-1 edges are progressively smaller. In this case, our task boils down to generating a binary vector where the probability $p$ that a position is 1 is small. We can generate this vector by generating a few geometric random variables representing the gap between 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. Using this technique, the cost of updating level-1 across the whole stream is at most $O(r \log(m/w))$.

The challenge in fastforwarding over the block $B$ lies in maintaining level-2 edges, finding the completed triangles, and keeping track of the counters. But these, too, can be maintained efficiently if we maintain some additional information and perform an extra pass through the block.

We first address the problem of maintaining the count $c$; the information we gather here will also be useful for updating level-2 edges. At this point, we have determined the level-1 edges for all $r$ estimators. Let $R$ be the set of the endpoints of these edges. We will maintain an integer counter for each node in $R$, requiring $O(r)$ space. Further, we augment the state of each estimator $\mathsf{est}$ with the following fields: $\mathsf{est}.d_1, \mathsf{est}.d_2$.

*Pass 1.* We will make a pass through $B$. As we go through the edges $b_1, b_2, \ldots, b_{|B|}$, we keep a count $\lambda_w$ of the edges incident on $w$ so far. For an estimator $\mathsf{est}$ with level-1 edge $e = \{x, y\}$, we record $\mathsf{est}.d_1 = \lambda_x$ and $\mathsf{est}.d_2 = \lambda_y$ when seeing $e$ in $B$ and when we finish the pass, the count $\mathsf{est}.c$ is incremented by $\lambda_x - \mathsf{est}.d_1 + \lambda_y - \mathsf{est}.d_2$, where $\lambda_x, \lambda_y$ are the values at the end of the pass. So far, we require one pass and $O(|B| + |R|) = O(|B| + r)$ time and space using a (hash) table which maps level-1 edges to their corresponding estimator states and a (hash) table storing the degree values.

Conveniently, the $\lambda$ values are also useful for updating level-2 edges. Consider an estimator $\mathsf{est}$ with level-1 edge $e = \{x, y\}$. The information we gather in the first pass provides us with clues about which edge to pick as a level-2 edge. We know that the number of edges in $B$ incident to $x$ that come after $e$ is exactly $\alpha = \lambda_x - \mathsf{est}.d_1$. Similarly, the number of edges in $B$ incident to $y$ that come after $e$ is exactly $\beta = \lambda_y - \mathsf{est}.d_2$. Therefore, the probability that an edge from $B$ will replace the current level-2 edge is exactly $(\alpha + \beta)/\mathsf{est}.c$. At this point, we can number the edges incident on $x$ with numbers $\{0, 1, \ldots, \alpha - 1\}$ and the edges incident on $y$ with numbers $\{0, 1, \ldots, \beta - 1\}$. Picking a random level-2 edge boils down to flipping a coin to decide whether to replace the orignal edge and if so, pick a random number between 0 and $\alpha + \beta - 1$ corresponding the replacement edge.

Our task now is simply to recognize the selected edge when we perform the second pass. This is where the second ingredient comes in: the event "*the edge that increments the $\lambda_x$ to a certain value $z$*" uniquely identifies an edge in $B$. Given the degree information obtained previously, we can describe our random selection of a level-2 edge in this form. Once these decisions are made, we store them a (hash) table mapping (vertex, desired-restricted-degree) to the corresponding estimator

*Pass 2.* A second pass through the data allows us to recognize level-2 replacement edges as well as their corresponding estimators that need to be updated. Since the table contains at most $O(r)$ entries, the space and time usage for this pass is $O(|B| + r)$. The second pass has another important role: we can recognize the edges that will complete the triangles partially formed by level- 1 and 2 edges. For this, we keep a (hash) table for queries of the form: "*if an edge $e$ is seen, we have a complete triangle for estimator $X$.*" This table is updated as we discover a new level-2 edge, and it is important that we perform our pass sequentially in the same order as we did in pass 1. Note that the size of this table is, again, at most $O(r)$. Hence, each block requires two passes, which can be completed in $O(|B| + r)$ space and time.

We can be more precise about our runtime bounds. Let $C_1$ and $C_2$ be constants independent of $r$. Using the optimized level-1 processing, the running time of our bulk-processing algorithm is at most

$$C_1 r(2 + H_{m/w}) + C_2(1 + \tfrac{m}{w})(w + r).$$

# C Omitted Proof from Section 2

We first state a measure concentration bound that will be used in the proofs.

**Theorem C.1 (A Chernoff Bound)** *Let $\lambda > 0$ and $X = X_1, \ldots, X_n$, where each $X_i$, $i = 1, \ldots, n$, is independently distributed in $[0, 1]$. Then,*

$$\mathbf{Pr}\Big[X \geq (1 + \lambda)\,\mathbf{E}[X]\Big] \leq e^{-\frac{\lambda^2}{2+\lambda}\cdot\mathbf{E}[X]} \quad \textit{and} \quad \mathbf{Pr}\Big[X \geq (1 - \lambda)\,\mathbf{E}[X]\Big] \leq e^{-\frac{\lambda^2}{2}\cdot\mathbf{E}[X]}.$$

*Proof of Theorem 2.3:* Let $\alpha = \frac{6}{\varepsilon^2}\frac{m\Delta}{\tau(G)}\log\left(\frac{2}{\delta}\right)$. We show that the average of $\alpha$ independent unbiased estimators from Algorithm 2 is an $(\varepsilon, \delta)$-approximation. For $i = 1, \ldots, \alpha$, let $X_i$ be the value returned by the $i$-th estimator. Let $\bar{X} = \frac{1}{\alpha}\sum_{i=1}^{\alpha} X_i$ denote the average of these estimators. Then, as a direct consequence of Lemma 2.2, we have $\mathbf{E}[X_i] = \tau(G)$ and $\mathbf{E}\left[\bar{X}\right] = \tau(G)$. Further, for $e \in E$, we have $c(e) \leq 2\Delta$, so know $X_i \leq 2m\Delta$. Let $Y_i = X_i/(2m\Delta)$ so that $Y_i \in [0,1]$. By letting $Y = \sum_{i=1}^{\alpha} Y_i$, we have $\mathbf{E}[Y] = \alpha\tau(G)/(2m\Delta)$; thus, by Chernoff bound (Theorem C.1),

$$\mathbf{Pr}\left[\bar{X} > (1+\varepsilon)\,\mathbf{E}[X]\right] = \mathbf{Pr}[\textstyle\sum_i Y_i > (1+\varepsilon)\,\mathbf{E}[Y]] \leq e^{-\frac{\varepsilon^2}{3}\mathbf{E}[Y]} = e^{-\frac{\varepsilon^2}{3}\frac{\alpha\cdot\tau(G)}{2m\Delta}} = \delta/2.$$

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 $O(\alpha)$. ■

*Proof of Lemma 2.7:* 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 $k \in \{1, 2, \ldots, n\}$, and 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 a randomized protocol is $\Omega(n)$ bits (see Chapter 4.2 in [KN97]).

Suppose there is a streaming algorithm $\mathcal{A}$ 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 $\{a_0, a_1, \cdots, a_n\} \cup \{b_0, \cdots, b_n\} \cup \{c_0, \cdots, c_n\}$. Alice places three edges among $a_0, b_0, c_0$ to form a triangle. For each $i \in \{1, 2, \ldots, n\}$, Alice places the edge $(a_i, b_i)$ if and only if $x_i$ equals 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 $(c_k, a_k)$. By querying the number of triangles in this graph with relative error of smaller that $0.5$, 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 used by the streaming algorithm at Alice must be $\Omega(n)$ bits. Note that the graph $G^*$ sent by Alice has no triples with two edges between them, and hence $O(1 + T_2(G^*)/\tau(G^*)) = O(1)$. We note that a similar proof applies to triangle sampling also. ■

# D Omitted Proofs from Section 3

Before we present proofs, we extend the notion of neighborhood to a set of edges. Let $f_1$ and $f_2$ be two edges of $\mathcal{S}$ so that $f_2$ arrives after $f_1$. Let $f$ be an edge that is adjacent to both $f_1$ and $f_2$. Note that such an edge may not exist. The neighborhood of $f_1$ and $f_2$, denote $N(f_1, f_2)$, is the following set:

$$N(f_1, f_2) = \{e \in N(f_1) \mid e \text{ arrives after } f_2\} \cup N(f_2) - \{f\}.$$

Let $c(f_1, f_2)$ denote the cardinality of $N(f_1, f_2)$.

*Proof of Lemma 3.1.:* Since $\kappa^*$ is a `Type I` clique, $f_1$ 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 $f_2$. We 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 $\kappa$ equals $\kappa^*$ when the following events are all true: (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 $N(r_1)$, $\Pr[\mathcal{E}_2 \mid \mathcal{E}_1] = \frac{1}{c(f_1)}$. Finally note that $r_3$ is chosen uniformly at random from $N(r_1, r_2)$. Thus $\Pr[\mathcal{E}_3 \mid \mathcal{E}_1, \mathcal{E}_2] = \frac{1}{c(f_1, f_2)}$. Thus the probability that $\kappa$ equals $\kappa^*$ is $\frac{1}{m} \cdot \frac{1}{c(f_1)} \cdot \frac{1}{c(f_1, f_2)}$, as desired.

■

*Proof of Lemma 3.2:* Suppose that the edges of $\kappa^*$ in the stream order were $f_1, f_2, \ldots, f_6$ respectively. We note that $\kappa_2 = \kappa^*$ if and only if at the end of observation, $r_1 = f_1$ and $r_2 = f_2$ in Algorithm 5. To see these, note that if at the end of observation, $r_1 = f_1$ and $r_2 = f_2$, then $\kappa_2 = \kappa^*$, since the edges $f_3, \ldots, f_6$ will certainly be added to $\kappa_2$ by the algorithm. The converse can also be easily seen to be true. The proof follows due to the face that the events $r_1 = f_1$ and $r_2 = f_2$ are independent, and each has a probability of $1/m$ of being true. ■

Before we present proof Theorem 3.3, we will first describe algorithms that estimate the number of `Type I` and `Type II` cliques.

---

**Algorithm 6:** COUNT-Type I

---

Run Algorithm 4 and let $\kappa_1$ and $c_1, c_2$ be the variables it maintains.
If $\kappa_1$ is a 4-Clique, then return $c_1 \cdot c_2 \cdot m$, else return 0.

---

The following states that Algorithm 6 returns an unbiased estimator form $\tau_4^1(G)$.

**Lemma D.1** *Let $X$ denote the random variable returned by Algorithm 6 after the graph $G$ has been observed. Then* $\mathbf{E}[X] = \tau_4^1(G)$.

*Proof:* Note that Algorithm 6 returns a non-zero value only when $\kappa_1$ is a `Type I` clique. For a `Type I` clique $\kappa^*$ let $f_1, f_2, f_3, f_4, f_5, f_6$ be the edges in the order of arrival. First, we note that if $\kappa = \kappa^*$, it must be true that $c_1 = c(f_1)$ and $c_2 = c(f_1, f_2)$.

$$\begin{aligned}
\mathbf{E}[X] &= \sum_{\kappa^* \in S_1} \mathbf{E}[X \mid \kappa = \kappa^*] \cdot \Pr[\kappa = \kappa^*] = \sum_{\kappa^* \in \mathcal{T}_4^1(G)} \Pr[\kappa = \kappa^*] \cdot m \cdot c_1 \cdot c_2 \\
&= \sum_{\kappa^* \in \mathcal{T}_4^1(G)} \Pr[\kappa = \kappa^*] \cdot m \cdot c(f_1) \cdot c(f_1, f_2) \\
&= \sum_{\kappa^* \in \mathcal{T}_4^1(G)} \frac{1}{m} \cdot \frac{1}{c(f_1)} \cdot \frac{1}{c(f_1, f_2)} \cdot m \cdot c(f_1) \cdot c(f_1, f_2) (\text{ By Lemma 3.1}) \\
&= \tau_4^1(G)
\end{aligned}$$

■

The next algorithm estimates the number of `Type II` cliques.

---

**Algorithm 7:** COUNT-Type II

---

Run Algorithm 5 and let $\kappa_2$ be the variable it maintains.
If $\kappa_2$ is a 4-Clique, then return $m^2$, else return 0.

---

**Lemma D.2** *Let $X$ denote the random variable returned by Algorithm 7. Then* $\mathbf{E}[X] = \tau_4^2(G)$.

The proof of the above lemma is similar to the proof of Lemma D.1, we omit the proof. Now we are ready to describe the algorithm to estimate the number of 4-cliques and prove Theorem 3.3

*Proof of Theorem 3.3:* Algorithm 8 is our estimator of number of 4-cliques. The correctness of the algorithm, space and accuracy bounds follow from Lemma D.1 and Lemma D.2 and applying a Chernoff bound on the resulting estimates.

■

By running $O(Dm \log(1/\delta))$ repetitions of Algorithm 9, we obtain Theorem 3.5, where $D = \max\{\Delta^2, m\}$.

**Algorithm 8:** COUNT-CLIQUES

**Result**: An $(\varepsilon, \delta)$ estimate for $\tau_4(G)$

$N \leftarrow \max\{6m\Delta^2, m^2\}\frac{\log(1/\delta)}{\tau_4(G)\varepsilon^2}$;

$X_1 \leftarrow 0, X_2 \leftarrow 0$;

Run $N$ independent copies of Algorithm 6 and of Algorithm 7;

Set $X_1$ to the mean of all values returned by Algorithm 6;

Set $X_2$ to the mean of all values returned by Algorithm 7;

Return $X_1 + X_2$.

**Algorithm 9:** Sample from $\mathcal{T}_4(G)$

Run in parallel Algorithms 4, and 5. Let $\kappa_1, c_1, c_2$ be the state maintained by Algorithm 4 and $\kappa_2$ be the state maintained by Algorithm 5.

**Query arrives for a random Clique**

Uniformly at random pick $b \in \{1, 2\}$;

If $b = 1$, then if $\kappa_1$ is a 4-clique, return $\kappa_1$ with probability $\frac{c_1 c_2}{D}$.

If $b = 2$, then if $\kappa_2$ is a 4-clique, return $\kappa_2$ with probability $m/D$.

Return "Fail"

**Lemma D.3** *For any 4-clique in G, say $\kappa^*$, the probability that Algorithm 9 returns $\kappa^*$ is $\frac{1}{2mD}$. The probability that the algorithm returns a 4-clique is $\frac{\tau_4(G)}{2mD}$.*

*Proof:* Suppose that $\kappa^*$ was a `Type I` clique, and suppose its first two edges in the stream order were $f_1, f_2$ respectively. Algorithm 9 returns $\kappa^*$ if: (1) $\mathcal{E}_1$:$b$ is chosen to be 1 by the algorithm, and (2) $\mathcal{E}_2$:$\kappa^*$ is chosen by $\kappa_1$, and (3) $\mathcal{E}_3$:$\kappa^*$ is finally returned by th algorithm in the final step, with prob. $c(f_1)c(f_1, f_2)/m\Delta$.

We know from Lemma 3.1 that $\Pr[\mathcal{E}_2] = \frac{1}{mc(f_1)c(f_1,f_2)}$. Thus,

$$\Pr[\mathcal{E}_1 \cap \mathcal{E}_2 \cap \mathcal{E}_3] = \frac{1}{2} \cdot \frac{1}{mc(f_1)c(f_1, f_2)} \cdot \frac{c(f_1)c(f_1, f_2)}{D} = \frac{1}{2mD}$$

It is possible to similarly show that the probability of returning each `Type II` clique is also $\frac{1}{2mD}$. Since the events of returning different cliques are all disjoint, the probability that Algorithm 9 returns some 4-clique is $\frac{\tau_4(G)}{2mD}$. ■