When a Graph is not so Simple: Counting Triangles in Multigraph Streams

Article · October 2013

Source: arXiv

Madhav Jha

15 PUBLICATIONS **374** CITATIONS

SEE PROFILE

Ali Pinar

Sandia National Laboratories

141 PUBLICATIONS **3,437** CITATIONS

SEE PROFILE

When a Graph is not so Simple: Counting Triangles in Multigraph Streams[*]

Madhav Jha[†] C. Seshadhri[‡] Ali Pinar[§]

Abstract

Estimating the number of triangles in a graph given as a stream of edges is a fundamental problem in data mining. The goal is to design a single pass space-efficient streaming algorithm for estimating triangle counts. While there are numerous algorithms for this problem, they all (implicitly or explicitly) assume that the stream does not contain duplicate edges. However, data sets obtained from real-world graph streams are rife with duplicate edges. The work around is typically an extra unaccounted pass (storing all the edges!) just to "clean up" the data. Can we estimate triangle counts accurately in a single pass even when the stream contains repeated edges? In this work, we give the first algorithm for estimating the triangle count of a multigraph stream of edges.

Keywords: triangle counting, streaming graphs, clustering coefficient, birthday paradox, streaming algorithms, multigraphs.

1 Introduction

The abundance of triangles has been observed in many applications of networks, such as social interaction, computer communications, financial transactions, proteins, or ecology. This abundance is noted as a critical feature that distinguishes real graphs from random graphs. In social sciences, triangle counts are used as a guide to understand graphs [Col88, Por98, Bur04, WDC10]. Triangle counts are also used in some graph mining applications such as spam detection [EM02] and finding common topics on the WWW [BBCG08]. Frequency of triadic patterns is a standard part of motif detection in bioinformatics [MSI$^+$02]. Triangles are also used in modeling and characterizing real-world networks [SKP12, DPKS12].

Many massive graphs come from modeling an continuing set of interactions between entities of a system. People call each other on the phone, exchange emails, or co-author a paper; computers exchange messages; animals come in the vicinity of each other; companies trade with each other. Each such interaction is modeled as an edge in the graph. These interactions (edges) manifest as a *stream of edges*. The edges appear "one at a time" with timestamps and the final graph is an accumulation of the observed edges. In all these examples, interactions are repeated events (e.g., multiple phone calls or emails; many papers co-authored together etc.), hence the graph is naturally a *multigraph*. (A multigraph allows multiple, also called parallel, edges between vertices, while a *simple* graph has no parallel edges.)

There are algorithmic methods to deal with such massive graphs, such as random sampling [SW05, TKMF09, SPK13], MapReduce paradigm [SV11, Pla12], distributed-memory parallelism [AKM12, CBB$^+$11], adopting external memory [CGG$^+$95, AGS10], and multithreaded parallelism [BHKK07]. *All of these methods start with preprocessing the data to remove the duplicate edges*, a potentially expensive operation.

1.1 Triangle counting in multigraph streams

In this paper we study triangle counting in multigraph streams. We first formalize the setting we study. The input multigraph is given as a sequence of edges $e_1, e_2, \ldots, e_m$. Some of the edges may be repeated, meaning that we may have, for example, $e_1 = e_{100} = e_{125} = (u, v)$. All edges are undirected. The final multigraph G is obtained by taking all the edges $e_1, e_2, \ldots, e_m$. Most graph analyses are performed on the underlying simple graph G', which is obtained by deleting all multiple copies of edges. Our aim is to estimate the number of triangles in G', denoted by T. We also wish to estimate the *transitivity* (sometimes called global clustering coefficient), which is $3T/W$, where W is the number of wedges in G'.

Our aim is to do this using a *small space streaming algorithm* that makes a *single pass* over the stream $e_1, e_2, \ldots, e_m$. At any timestamp t, such an algorithm

[*] This work was funded by the DOE ASCR Complex Interconnected Distributed Systems (CIDS) program and Sandia's Laboratory Directed Research & Development (LDRD) program. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

[†] Sandia National Laboratories, Livermore, CA, mjha@sandia.gov

[‡] Sandia National Laboratories, CA, scomand@sandia.gov

[§] Sandia National Laboratories, CA, apinar@sandia.gov

retains a very small random subset of the edges seen so far $(e_1, e_2, \ldots, e_t)$. This is called the "sketch" and is updated rapidly as new edges appear. Using the sketch and some auxiliary data structures, the algorithm computes an accurate estimate for the number of triangles for the graph seen so far. The size of the data structures is orders of magnitude smaller than the size of the graph. Because of the single pass and small space, the algorithm cannot revisit edges that it has forgotten. Furthermore, it cannot always determine if the new edge, e_t, has already appeared before.

1.2 The multigraph issue Practical streaming algorithms for massive graphs is an increasingly important subject. We refer the reader to a recent tutorial on network sampling at KDD 2013 [HAN13]. There have been numerous streaming algorithms specifically for triangle counting [BFL+06, TPT13, PTTW13], with the state of the art arguably being recent previous work of the authors [JSP13]. All these results ignore the problem of duplicate edges in multigraphs. This is a convenient and standard assumption that allows for algorithmic progress. But this avoids a practical difficulty in processing a real-world graph stream. For example, the classic Enron email dataset is really a multigraph with 790K edges, while the underlying simple graph has 167K edges. Similarly, a DBLP co-authorship graph recently collected is actually a multigraph with 8.9M edges, but the underlying simple graph has only 5.1M edges.

We believe that for streaming algorithms to be actually useful in practice, the problem of multiple edges must be dealt with. We stress that all previous streaming algorithms break down when given multiple edges, and there are almost no attempts on triangle counting in this setting. Previous work on multigraph mining explicitly states triangle counting of streaming multigraphs as an open problem [CM05].

Triangle counting becomes quite tricky in multigraphs. Consider edges a, b, and c that form a triangle. These edges may appear in the multigraph stream in many different ways. For example, these edges could come as $a, a, \ldots, b, b, \ldots, c, c, \ldots$, or $a, b, c, a, b, c, a, b, c, \ldots$. (Observe how this is *not* an issue for simple graphs.) In the first case, if the algorithm does not sample the edge a from the first stretch of a, then the triangle will not be found. But in the latter case, it appears that random sampling is more likely to find the triangle. For an unbiased estimate of T, the algorithm should detect each triangle with the same probability, and the multigraph poses serious problems for such estimates. The stream might have some complicated pattern of a, b, c, but we must count the triangle only once regardless of this pattern. Because of the small space, it is not possible to store enough of the history to determine if a triangle has already been detected before.

1.3 Our Contributions Previous work of the authors [JSP13] gives a practically (and provably) accurate, small-space algorithm for counting triangles in a simple graph stream. The main contribution of this work is extending this algorithm for multigraphs.

- **Sampling with multiple edges and unbiasing estimates:** The major problem with previous work is that random sampling in multigraph stream creates biases leading to incorrect estimates. Furthermore, the same triangle may appear numerous times (in many different ways) in the stream, but we must count it exactly once. We handle all these issues and provide a *provably* correct algorithm.

- **Simplicity of multigraph "fix":** Our hash-based sampling and unbiasing methods to deal with multigraphs is extremely simple to implement. This leads to a practical algorithm that deals with multigraphs. We freely admit that the algorithmic change looks quite incremental over the basic algorithm of [JSP13]. But this simple change handles the rather challenging problem of multigraphs, and is one of the first provably practical algorithms in this setting.

- **Empirical studies that show effectiveness in practice:** We have conducted detailed experiments on real and artificially generated data sets to show that our proposed algorithm performs well in practice. By using the orders of magnitude smaller storage than the full graph, we were able to predict the transitivity and the number of triangles accurately. We showed that the algorithm rapidly converges to the true values with increasing storage. Moreover, our experiments also showed that the algorithm is robust to different orderings of the stream.

1.4 Related Work The most closely related work from the perspective of computing on mulitgraph streams are the works of [VSGB05] and [CM05]. As mentioned earlier, [CM05] explicitly mentions the question of counting subgraphs in multigraph as directions for future work.

There is significant history on triangle counting in various settings, and we simply refer to reader to the references and discussion in [SPK13, JSP13]. There is much work on triangle counting in graph streams [JG05, BFL+06, AGM12, KMSS12, TPT13, PTTW13, JSP13]. As mentioned earlier, all this work focuses on simple graphs.

2 Idealized algorithm

In this section, for the ease of theoretical analysis, we present an idealized version of our algorithm. Our algorithm is based on detecting closure of wedges. In [JSP13], the notion of a *future closed wedge* was introduced. A wedge w formed by edges e_{t_1} and e_{t_2} is a future closed wedge if there exists an edge e_{t_3} with $t_3 > \max\{t_1, t_2\}$ such that edges e_{t_1}, e_{t_2} and e_{t_3} form a triangle. Observe that for a simple graph, each triangle contains exactly one future closed wedge, and hence the number of these is exactly T. The algorithm of [JSP13] provides a streaming algorithm to estimate the number of future closed wedges.

However, in multigraph streams, the number of future closed wedges of a triangle depends on the ordering of the stream. Hence, estimating the number of these wedges does not give an accurate triangle count. Our main insight is a new notion called *last future closed* wedge. Given a triangle $\tau = \{a, b, c\}$ formed by edges a, b, and c, let $last(\tau) \in \tau$ be the last occurrence of an edge of τ. Then the wedge opposite $last(\tau)$, namely, $\tau \setminus last(\tau)$ is the *last future closed* wedge. By definition, every triangle has precisely 1 last future closed wedge. Observe that the last future closed wedge for a triangle depends on the stream order. Nonetheless, at any fixed point in the stream, the number of such wedges is equal to the number of triangles.

Algorithm 1 estimates the number of last future closed wedges. Based on ideas in [JSP13], it maintains a small uniform sample (called *edge-sample*) of the set (not multiset) of edges of the graph seen so far. In addition, the algorithm tracks every wedge formed by edges in *edge-sample* for *closure*. Specifically, for every wedge w, the algorithm maintains a Boolean value F_w that is 1 if and only if w is the *last future closed* wedge.

Algorithm 1 takes as input a sampling rate parameter α which is the probability with which it includes an edge of the graph in the sample. Concretely, for every edge e in the stream, the algorithm computes a hash value $hash(e)$ which is uniformly distributed in $[0, 1]$. Therefore, to select an edge e with probability α, it suffices to include it in the sample if and only if $hash(e) \leq \alpha$. Observe that the number of occurrences of an edge does not affect its inclusion in *edge-sample*. Moreover, the behavior of the algorithm is deterministic across all occurrences of an edge. In particular, if the first occurrence of an edge is not included in the sample, then none of the later occurrences will.

How do we track the last future closed wedge? An edge not only witnesses the fact that a wedge may be *future closed*, but also certifies that another wedge is *no longer the last future closed wedge*. In other words, it invalidates the *last future closed wedge* status of every wedge containing the edge. This is the main bias correction step of the algorithm and is implemented in Step 11.

The bias correction is deceptively simple. Consider some wedge w formed by edges in *edge-sample*. If the new edge e_t closes w, then we set $F_w = 1$. If e_t happens to be an edge of w, we simply set $F_w = 0$. (Otherwise, F_w does not change.) This is enough for the bias correction, and together with the hash-based sampling, gives a streaming algorithm for multigraphs.

Algorithm 1: IdealCountTriangles (α)

```
1  Initialize edge-sample as an empty set.
2  foreach edge $e_t$ in the stream do
3      Let $x \leftarrow \mathsf{hash}(e_t)$ be a random value in $[0, 1]$.
4      if $x \leq \alpha$ and $e_t \notin$ edge-sample then
5          Insert $e_t$ in edge-sample.
6      foreach wedge $w$ in edge-sample do
7          Let $w = \{\{u, v\}, \{u, w\}\}$.
8          if $e_t$ is the closing edge $\{v, w\}$ then
9              Set $F_w$ to 1.
10         else if $e_t \in \{\{u, v\}, \{u, w\}\}$ then
11             Reset $F_w$ to 0.  // bias-correction
12 Output $F = \frac{1}{\alpha^2} \sum_w F_w$ where the sum is over
   wedges $w$ in edge-sample.
```

Theorem 2.1. (Main) *Fix some parameter $\alpha \in (0, 1)$. Let F be the output of Algorithm 1 when run on the stream of edges of the underlying (simple undirected) graph G with T triangles. Then $\mathbf{E}[F] = T$.*

Proof. We first extend the definition of F_w to every wedge w in graph G. We define $F_w = 0$ if F_w is never assigned (set or reset) during the invocation of the algorithm. This happens precisely if one of the edges of w is not sampled. Then $F = \frac{1}{\alpha^2} \sum_w F_w$ where the sum is over all wedges w in G. For every edge e in G, let $t_{max}(e)$ be the maximum value of t such that $e_t = e$. Fix a triangle $\tau = \{a, b, c\}$ formed by edges a, b and c and assume (by relabeling if required) that c is the last edge to appear in the stream among a, b, and c. In other words, $t_{max}(c) > \max\{t_{max}(a), t_{max}(b)\}$. Since $\{a, b\}, \{b, c\}, \{c, a\}$ are wedges, it makes sense to talk about $F_{\{a,b\}}$, etc.

Lemma 2.1. $F_{\{b,c\}} = F_{\{c,a\}} = 0$. *Moreover, $F_{\{a,b\}} = 1$ if and only if both edges a and b are in edge-sample.*

Proof. Consider the moment $t = t_{max}(c)$ when $e_t = c$. If wedge $\{b, c\}$ is not in *edge-sample*, then by definition, $F_{\{b,c\}}$ is 0. On the other hand, if wedge $\{b, c\}$ is in *edge-sample*, then by Step 11 of Algorithm 1, the value of

$F_{\{b,c\}}$ is reset to 0. No subsequent change is made to this value. An identical argument shows the same for $F_{\{c,a\}}$. Finally, $F_{\{a,b\}}$ is set to 1 at this moment if and only if wedge $\{a, b\}$ is in *edge-sample*, and once again, this value is not changed subsequently.

From the above lemma, it follows that $\mathbf{E}[F_{\{b,c\}}] = \mathbf{E}[F_{\{c,a\}}] = 0$, while $\mathbf{E}[F_{\{a,b\}}] = \alpha^2$, the probability that both edges a and b are sampled. Therefore, the sum of expectation of F_w over all three wedges w of the triangle $\tau = \{a, b, c\}$ is $\sum_{w\in\tau} \mathbf{E}[F_w] = \alpha^2$. Observe this is true for any fixed triangle τ of G. For any wedge w that does not participate in a triangle, F_w is obviously zero. Let $\mathcal{T}$ denote the set of triangles. By linearity of expectation,

$$\begin{aligned}\mathbf{E}[F] = \frac{1}{\alpha^2}\mathbf{E}[\sum_w F_w] &= \frac{1}{\alpha^2}\sum_{\tau\in\mathcal{T}}\sum_{w\in\tau}\mathbf{E}[F_w] \\ &= \frac{1}{\alpha^2}\cdot\alpha^2 T = T\end{aligned}$$

3 Implementing Algorithm 1.

In this section, we detail our efforts to implement the ideal algorithm of the previous section. Here, we heavily build on the algorithm of [JSP13]. For completeness, we give the full algorithm in Algorithm 2. The main distinction from the idealized algorithm is that the value F_w is not maintained for every wedge w in *edge-sample*. Instead, the value F_w is tracked only for a small sample of wedges sampled from wedges in *edge-sample*. This random pool of wedges (sometimes referred to as the wedge reservoir) is a fixed size array called *wedge-sample*. The corresponding F_w values are stored in a Boolean array of the same size called *isClosed*. Thus, the main data structures maintained by Algorithm 2 are (i) a set of edges (*edge-sample*), (ii) an array of wedges (*wedge-sample*) and (iii) a Boolean array *isClosed*. The capacities of these data structures are bounded by input parameters s_e (for the first one) and s_w (for the last two). The other key value maintained by the algorithm is *tot_wedges* giving the number of total wedges formed by edges in *edge-sample*. Next we describe key steps of the algorithm.

1. **(Maintaining uniform edge sample.)** As in the idealized algorithm, we use a hash function which hashes the edge to a uniform value in $[0, 1]$. We sample the edge only if its hash value is at most α. The hash function that we use is Murmur3 [A. 08].

2. **(Maintaining uniform wedge sample.)** As in [JSP13], this is maintained by doing a reservoir sampling on the set of wedges in *edge-sample* without explicitly maintaining this set. The main trick here is that the value of *tot_wedges* together with the set of newly formed wedges involving the current edge e_t is enough to simulate reservoir sampling over the set of wedges. See [JSP13] for details.

3. **(Bias correction.)** This is the trickiest part of the implementation. In principle, we follow the same steps as the idealized algorithm. Steps 10-15 of Algorithm 2 correspond to Steps 6-11 of Algorithm 1.

4. **(Keeping size of *edge-sample* in check.)** If size of *edge-sample* reaches s_e, we roughly throw away half the edges from *edge-sample*. Specifically, every edge in *edge-sample* is evicted with probability 1/2 and the sampling rate α is reduced by 1/2, as well. This is implemented in Steps 3-9.

4 Experiments.

We implemented the proposed algorithm in `C++` and performed experiments with a large set of graphs.

4.1 Real datasets

We conducted experiments on the DBLP co-authorship and Enron email networks.

DBLP Co-authorship network: One advantage of our approach is that it allows working directly with the raw data. That is, without making an extra pass to remove duplicate edges. To this end, we downloaded the raw XML data from DBLP [Ley] and used our algorithm to estimate the number of triangles *without any preprocessing*. The dataset consists of metadata entry of papers on dblp with each entry describing the paper title and list of authors. Observe that a list of authors of the form $\{a, b, c\}$ gives rise to three edges (a, b), (b, c) and (c, a). Also, observe that the dataset inherently creates multiple edges. (Example, consider two different papers with authors $\{a, b, c\}$ and $\{a, b, d\}$.) Our algorithm estimates triangle count without removing these duplicate edges and by only making a single pass over these edges.

To evaluate our algorithm, we did process the data to compute exact values of the network statistics. The network consists of $1.26M$ vertices and $8.97M$ edges of which there are only about $5M$ *distinct* edges. We ran our algorithm with $s_e = s_w = 30K$ and found it reporting pretty accurate answers. In particular, our algorithm reported 0.1733 as the transitivity estimate where the true value (of the underlying simple graph) is 0.1743. Similarly, our triangles estimate $(11, 299, 160)$ was within 1.5% relative error of the true count of triangles $(11, 299, 160)$ in the underlying simple graph. We mention that considering multiplicity of edges, the

Algorithm 2: CountTriangles(s_e, s_w)

```
Let $\alpha = 0.5$. Initialize *wedge-sample* as an empty array of size $s_w$ and *edge-sample* as an empty set.
foreach edge $e_t$ in the stream do
    /* If edge-sample size limit is reached, probabilistically remove half the edges. Decrease sampling rate $\alpha$ to $\alpha/2$. */
    if $|edge\text{-}sample| \geq s_e$ then
        Set $\alpha$ to $\alpha/2$.
        foreach edge $e$ in edge-sample do
            Let $x \leftarrow \mathsf{hash}(e)$.
            if $x > \alpha$ then
                Remove $e$ from edge-sample.
                Update tot_wedges.

    /* Set/Reset isClosed based on $e_t$. */
    for $i \in 1, \ldots, s_w$ do
        Let $\{\{u, v\}, \{u, w\}\} \leftarrow wedge\text{-}sample[i]$
        if $e_t$ is the closing edge $\{v, w\}$ then
            Set $isClosed[i]$ to 1.
        else if $e_t \in \{\{u, v\}, \{u, w\}\}$ then
            Reset $isClosed[i]$ to 0.

    /* Insert $e_t$ with probability $\alpha$ by deterministically hashing $e_t$ to a uniform value in $[0, 1]$ */
    Let $x \leftarrow \mathsf{hash}(e_t)$.
    if $x > \alpha$ or $e_t$ already in edge-sample then
        Proceed to the next edge in the stream.
    Insert $e_t$ in edge-sample. Update tot_wedges.
    Determine $\mathcal{N}_t$ (wedges involving $e_t$ in edge-sample) and let $new\_wedges = |\mathcal{N}_t|$.

    /* Refresh (probabilistically) every wedge sample with a new wedge */
    for $i \in 1, \ldots, s_w$ do
        Pick a random number $y$ in $[0, 1]$
        if $y \leq new\_wedges/tot\_wedges$ then
            Pick uniform random $w \in \mathcal{N}_t$.
            $wedge\text{-}sample[i] \leftarrow w$.
            $isClosed[i] \leftarrow$ false.
Let $\rho$ be the fraction of values in isClosed which are set to 1. Output $3 \cdot \rho$ as the transitivity estimate. Output $\alpha^{-2} \cdot tot\_wedges \cdot \rho$ as the estimate for triangles.
```

graph has a whooping $90M$ triangles.

Enron email network: We also experimented with the Enron email dataset [ENR] which consists of email exchanges of employees of Enron. Being a communication network, it naturally contains many repeated edges (e.g. links corresponding to multiple email exchanges between the same pair of individuals). To give exact numbers, the email network consists of $19,133$ vertices and $790,871$ edges. There are however only $167,273$ *distinct* edges. The underlying simple network has $996,306$ triangles with the transitivity value of 0.1058. The number of triangles in the multigraph is about 372 times larger: $370,721,337$. The transitivity value of the multigraph is also much higher: 0.39386.

Our estimate with $30K$ edges and $30K$ wedges was 0.1193 for transitivity (only 0.013 absolute error) and $1,104,100$ for the number of triangles (about 10% relative error).

4.2 Other networks For a thorough empirical study, we have extended our data set to include networks obtained from the SNAP database [SNA13]. The vital statistics of all the simple graphs are provided in Tab. 1. In this table, $|V|$, $|E_s|$, W, T, and κ correspond to the number of vertices, number of edges, number of wedges, number of triangles, and the transitivity, respectively.

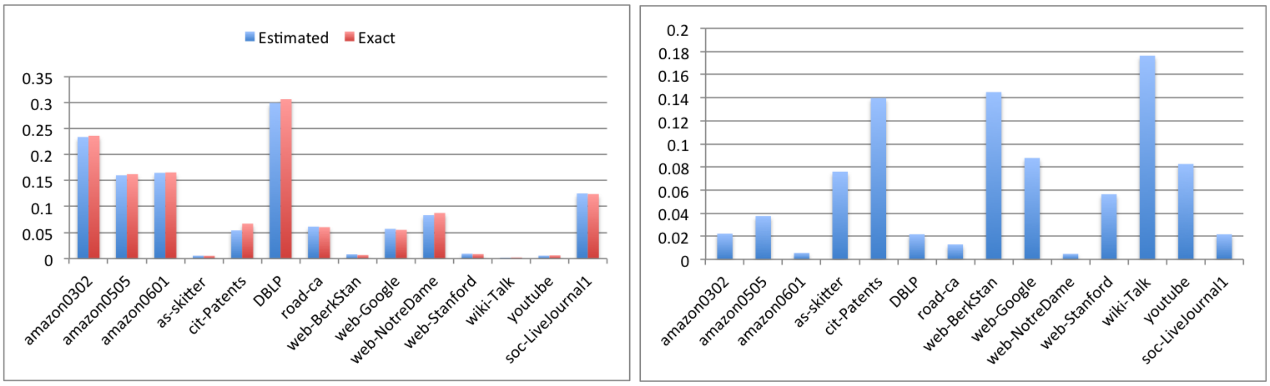
To generate multi graphs form these simple graphs, we artificially injected multiple edges in the dataset as follows. For every edge e in the dataset, we flipped a coin and based on the coin toss decided to either replicate the edge or leave it as it is. More precisely, for every edge independently, with one-third probability, we replicated the edge many times and with two-third probability left the edge as it is. When selected for replication, the edge was replicated x times where x was chosen equiprobably from $2, 4, 8, 16$, and 32.

We applied our algorithm to these multi graphs to estimate the transitivity and the number of triangles in the underlying simple graphs. The results of our experiments are presented in Fig. 1. The results show that our algorithm always estimates the transitivity very accurately. The number of triangles however, can be off. This is because the number of triangles is estimated by multiplying the transitivity with the estimate for the number of wedges. For graphs where the transitivity is low and the the number of wedges is high, this multiplication amplifies tiny errors in transitivity into large differences in the number of triangles.

4.3 Convergence of estimates In this set of experiments, we show that our algorithm converges to the

Table 1: Properties of the graphs used in the experiments

Graph	$\lvert V\rvert$	$\lvert E_s\rvert$	W	T	κ
amazon0302	262K	900K	9.1M	718K	0.236
amazon0505	410K	2439K	73M	3951K	0.162
amazon0601	403K	2443K	72M	3987K	0.166
as-skitter	1696K	11095K	16022M	28770K	0.005
cit-Patents	3775K	16519K	336M	7515K	0.067
DBLP	317K	1049K	21M	2240K	0.3064
roadNet-CA	1965K	2767K	6M	121K	0.060
web-BerkStan	685K	6649K	27983M	64691K	0.007
web-Google	876K	4322K	727M	13392K	0.055
web-NotreDame	326K	1090K	305M	8910K	0.088
web-Stanford	282K	1993K	3944M	11329K	0.009
wiki-Talk	2394K	4660K	12594M	9204K	0.002
youtube	1158K	2990K	1474M	3057K	0.006
livejournal	5284K	48710K	7519M	310877K	0.124



(a) Transitivity

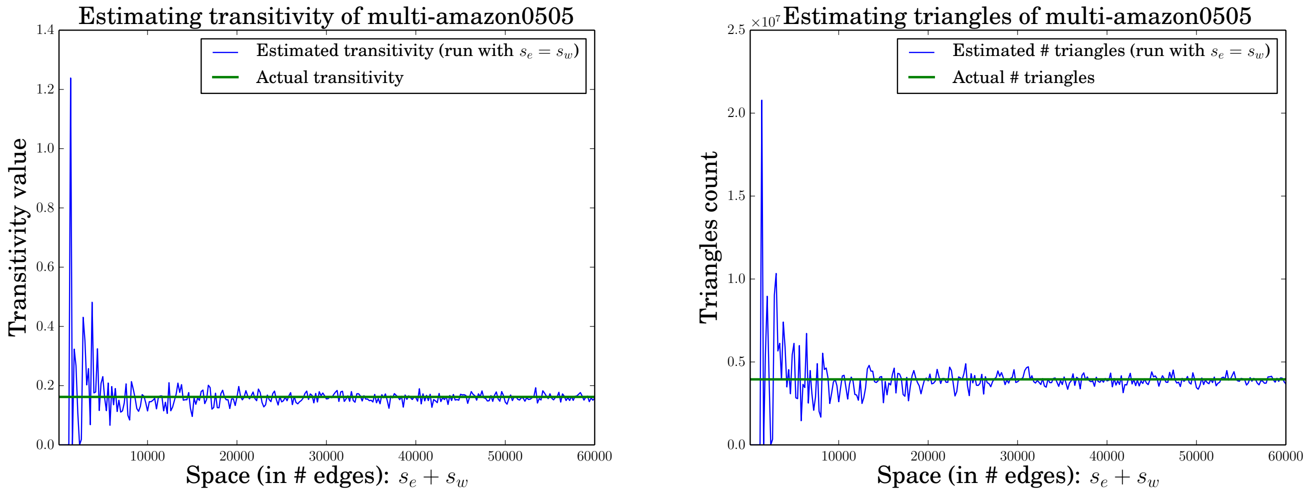
(b) Triangles

Figure 1: Output of a single run of CountTriangles on a variety of real datasets with 25K edge reservoir and 25K wedge reservoir. The plot on the left gives the estimated transitivity values (labelled streaming) alongside their exact values. The plot on the right gives the relative error of CountTriangles's estimate on triangles T.

true value as we increase the space. For this purpose, we run our algorithm on amazon0505 graph (after converting it into a multigraph as described in the previous item) by gradually increasing the space ($r_e + r_w$) available to the algorithm. For convenience, we keep the size of edge reservoir and wedge reservoir the same. Fig. 2 displays the estimates for the transitivity and the number of triangles in our experiments. As the figure shows, the estimates converge to the true value as the available memory increases. The estimates oscillate for smaller sizes at first, but stabilize after about 10K edges. After that the return in improved accuracy for increased storage starts diminishing. We have observed similar trends in other data sets.

4.4 Sensitivity to the stream order To understand how we generate different edge orderings of a multigraph, consider the following operations. Given a stream of edges σ, let $Repeat10(\sigma)$ denote the stream obtained by replacing each edge e of σ by 10 copies of e. Likewise, let $RepeatVariable(\sigma)$ denote the stream obtained by replacing each edge of the stream by i copies where i is 1 with probability 2/3 and is distributed uniformly in $\{2, 4, 8, 16, 32\}$ with the remaining probability. Finally, $Permute(\sigma)$ means randomly permuting the stream while $BlockPermute(\sigma)$ means breaking the stream into blocks of equal size (5000) and then only permuting the blocks.

Let σ_0 be the original stream of edges of amazon0505 dataset. Then the first order (Order 1) is the stream $\sigma_1 = Repeat10(\sigma_0)$ while the second order (Or-



(a) Convergence of the transitivity estimate

(b) Convergence of triangles estimate

Figure 2: Concentration of estimate on multi-amazon0505: We plot the transitivity and triangles estimate as a function of total space ($s_e + s_w$) and observe that they converge to their true values. In our experiment, we keep $s_e = s_w$.

der 2) is the stream $\sigma_2 = BlockPermute(\sigma_1)$. The third order (Order 3) is the stream $\sigma_3 = RepeatVariable(\sigma_0)$ while the fourth order (Order 4) is the stream $\sigma_4 = Permute(\sigma_3)$. Finally, the fifth order (Order 5) is the stream $\sigma_5 = RepeatVariable(\sigma_0)$ and the last order (Order 6) is the stream $\sigma_6 = BlockPermute(\sigma_5)$.

The results of our experiments are presented in Fig. 3. In the figures, the first column always corresponds to the true value, and the remaining 6columns correspond to the estimates of our algorithm based on streams ordered as described above. It can be seen that the estimates of the algorithm are not sensitive to the ordering used and always accurate. Note that the 6 orders are not merely random orders. they have been designed to expose a sensitivity of the algorithm to the stream order. Yet, the results show that the algorithms robust to stream orderings.

5 Conclusions

We have described a streaming algorithm to compute the number of triangles and the transitivity (global clustering coefficient) of a multigraph (a graph with parallel edges). This new algorithm extends our previous work that described a streaming algorithm that required the graph to be simple. Our new method adopts a randomized hash function to identify repeated edges. However, a randomized hash function by itself is not sufficient as the order of the sequence may affect the prediction. To correct for this problem, we use an unbiasing technique that makes sure only one wedge per triangle is being considered for closure.

We provide experimental results that show that the proposed techniques work well in practice. We were able to analyze the DBLP co-authorship network by directly processing the raw data, without explicitly constructing a graph. We estimate the transitivity as 0.1733, when the true value is 0.1743 and the number of triangles as 11, 299, 160 when the true vale is 11, 460, 675. We have also experimented with artificially generated data sets, and various stream orders, and always achieved accurate estimations.

Acknowledgements

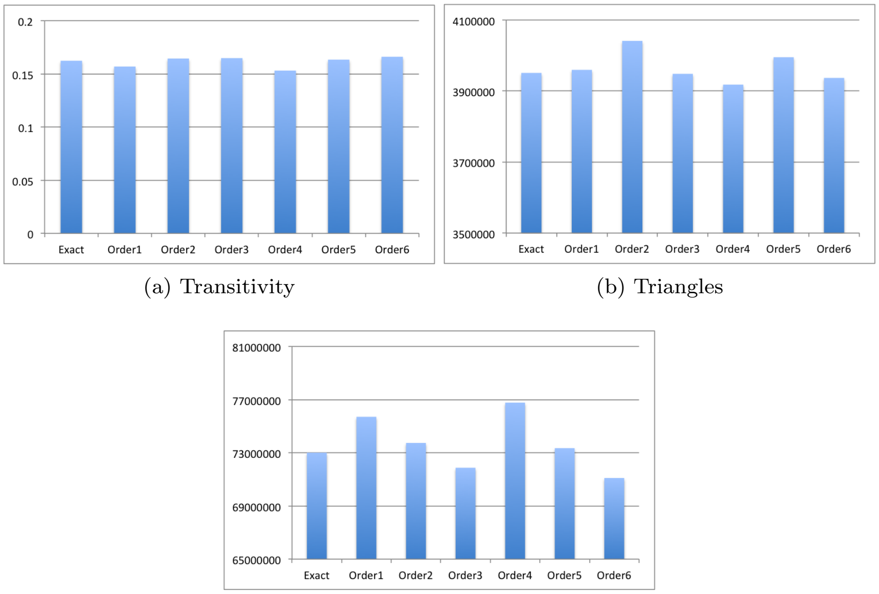
We thank Ashish Goel for suggesting the use of hash-function based reservoir sampling. This was a key step towards the development of the final algorithm.

References

[A. 08] A. Appleby. Murmur hash, 2008. Available at https://sites.google.com/site/murmurhash/.

[AGM12] Kook J. Ahn, Sudipto Guha, and Andrew McGregor. Graph sketches: sparsification, spanners, and subgraphs. In *Principles of Database Systems*, pages 5–14, 2012.

[AGS10] L. Arge, M.T. Goodrich, and N. Sitchinava. Parallel external memory graph algorithms. In *Parallel Distributed Processing Symposium (IPDPS)*, pages 1 – 11, april 2010.



(a) Transitivity

(b) Triangles

(c) Wedges

Figure 3: Affects of the stream order on the accuracy of the estimations on graph amazon0505.

[AKM12] S. M. Arifuzzaman, M. Khan, and M. Marathe. Patric: A parallel algorithm for counting triangles and computing clustering coefficients in massive networks. Technical Report 12-042, NDSSL, 2012.

[BBCG08] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient semi-streaming algorithms for local triangle counting in massive graphs. In *Knowledge Data and Discovery (KDD)*, pages 16–24, 2008.

[BFL$^+$06] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *Principles of Database Systems*, pages 253–262, 2006.

[BHKK07] J.W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *Parallel and Distributed Processing Symposium (IPDPS)*, pages 1 –14, march 2007.

[Bur04] Ronald S Burt. Structural holes and good ideas. *American Journal of Sociology*, 110(2):349–399, 2004.

[CBB$^+$11] Dhruva R. Chakrabarti, Prithviraj Banerjee, Hans-J. Boehm, Pramod G. Joisha, and Robert S. Schreiber. The runtime abort graph and its application to software transactional memory optimization. In *International Symposium on Code Generation and Optimization*, pages 42–53, 2011.

[CGG$^+$95] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Symposium on Discrete Algorithms (SODA)*, pages 139–149, 1995.

[CM05] Graham Cormode and S. Muthukrishnan. Space efficient mining of multigraph streams. In Chen Li, editor, *PODS*, pages 271–282. ACM, 2005.

[Col88] James S. Coleman. Social capital in the creation of human capital. *American Journal of Sociology*, 94:S95–S120, 1988.

[DPKS12] N. Durak, A. Pinar, T. G. Kolda, and C. Seshadhri. Degree relations of triangles in real-world networks and graph models. In *CIKM'12*, 2012.

[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 (PNAS)*, 99(9):5825–5829, 2002.

[ENR] Enron email dataset. Available at https://www.cs.cmu.edu/~enron/.

[HAN13] Mohammad Al Hasan, Nesreen Ahmed, and Jennifer Neville. Network sampling: Methods and applications. In *Proceedings of ACM SIGKDD*, 2013. http://www.cs.purdue.edu/homes/neville/courses/kdd13-tutorial.html.

[JG05] H. Jowhari and M. Ghodsi. New streaming algorithms for counting triangles in graphs. In *Computing and Combinatorics Conference (COCOON)*, pages 710–716, 2005.

[JSP13] Madhav Jha, C. Seshadhri, and Ali Pinar. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '13, pages 589–597, New York, NY, USA, 2013. ACM.

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

598–609, 2012.

[Ley] M. Ley. Digital bibliography & library project. Available at http://www.informatik.uni-trier.de/~ley/db/.

[MSI$^+$02] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[Pla12] Todd Plantenga. Inexact subgraph isomorphism in mapreduce. *Journal of Parallel and Distributed Computing*, (0), 2012.

[Por98] Alejandro Portes. Social capital: Its origins and applications in modern sociology. *Annual Review of Sociology*, 24(1):1–24, 1998.

[PTTW13] A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. Counting and sampling triangles from a graph stream. In *International Conference on Very Large Databases (VLDB)*, 2013.

[SKP12] C. Seshadhri, Tamara G. Kolda, and Ali Pinar. Community structure and scale-free collections of Erdös-Rényi graphs. *Physical Review E*, 85(5):056109, May 2012.

[SNA13] SNAP. Stanford network analysis project, 2013. Available at http://snap.stanford.edu/.

[SPK13] C. Seshadhri, Ali Pinar, and Tamara G. Kolda. Fast triangle counting through wedge sampling. In *Proceedings of the SIAM Conference on Data Mining*, 2013.

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

[SW05] Thomas Schank and Dorothea Wagner. Approximating clustering coefficient and transitivity. *Journal of Graph Algorithms and Applications*, 9:265–275, 2005.

[TKMF09] Charalampos E. Tsourakakis, U. Kang, Gary L. Miller, and Christos Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *Knowledge Data and Discovery (KDD)*, pages 837–846, 2009.

[TPT13] Kanat Tangwongsan, A. Pavan, and Srikanta Tirthapura. Parallel triangle counting in massive streaming graphs. In *ACM Conference on Information & Knowledge Management (CIKM)*, 2013.

[VSGB05] Shobha Venkataraman, Dawn Xiaodong Song, Phillip B. Gibbons, and Avrim Blum. New streaming algorithms for fast detection of superspreaders. In *NDSS*. The Internet Society, 2005.

[WDC10] B. F. Welles, A. Van Devender, and N. Contractor. Is a friend a friend?: Investigating the structure of friendship networks in virtual worlds. In *CHI-EA'10*, pages 4027–4032, 2010.