GREAT: Generalized Reservoir Sampling based Triangle Counting Estimation over Streaming Graphs

Siyue Wu
College of Computer Science and Software Engineering
Shenzhen University
wusiyue1229@gmail.com

Dingming Wu*
College of Computer Science and Software Engineering
Shenzhen University
dingming@szu.edu.cn

Sinhong Cheuk
College of Computer Science and Software Engineering
Shenzhen University
sinhongcheuk@gmail.com

Tsz Nam Chan
College of Computer Science and Software Engineering
Shenzhen University
edisonchan@szu.edu.cn

Kezhong Lu
College of Computer Science and Software Engineering
Shenzhen University
kzlu@szu.edu.cn

ABSTRACT

The number of triangles of a streaming graph is a crucial metric with various applications, such as network evolution analysis, community detection, and anomaly detection. A practical solution for triangle counting in streaming graphs is the sampling-based approximation. Although a lot of research efforts have been devoted to the fixed-sized memory based algorithms, they suffer from the accuracy and the efficiency issues. To tackle these issues, we first propose the generalized reservoir sampling (GRS), which stores less edges for reducing the computational cost and can still generate uniformly random edge sample in the streaming graph. Then, we propose the GREAT algorithm based on GRS for efficient and accurate triangle counting estimation. To further improve the estimation accuracy, we propose the GREAT$^+$ algorithm for considering the dynamic timestamp interval distribution in real-world streaming graphs so that triangles with short and long timestamp intervals will be sampled following the ground-truth distribution. Extensive evaluations on real datasets demonstrate the efficiency and the accuracy of our algorithms. The relative error of our algorithm GREAT$^+$ is significantly (an order of magnitude) better than the competitors.

PVLDB Reference Format:
Siyue Wu, Dingming Wu, Sinhong Cheuk, Tsz Nam Chan, and Kezhong Lu. GREAT: Generalized Reservoir Sampling based Triangle Counting Estimation over Streaming Graphs. PVLDB, 18(7): 2031 - 2043, 2025.
doi:10.14778/3734839.3734842

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/sinhong-cheuk/GREAT.

*Corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 7 ISSN 2150-8097.
doi:10.14778/3734839.3734842

1 INTRODUCTION

Graph is an important data structure that is used to represent relationships or connections between objects [46]. A triangle is a substructure within a graph consisting of three vertices connected by three edges. A streaming graph a.k.a. an edge stream [2] is an unbounded sequence of edges where each edge arrives at a timestamp. The arrived edges form a dynamic graph. Real-world applications of triangle counting estimation over streaming graphs span several domains. In social networks, users interact dynamically, and triangle counting estimation can help detect real-time communities based on the evolving graph structure [24, 37]. In financial networks, triangle counting estimation can help identify fraud rings or money laundering by detecting clusters of suspicious transactions or individuals that form triangles [29].

In the streaming graph setting, it is infeasible to store the whole graph since the edge stream is unlimited. Moreover, there is no knowledge about the future edges in the stream. Therefore, existing exact algorithms [3, 6, 10] and matrix-based approximation algorithms [41] are not applicable in the streaming graph setting since they require random accesses to the whole graph and need to traverse the graph in a specific way.

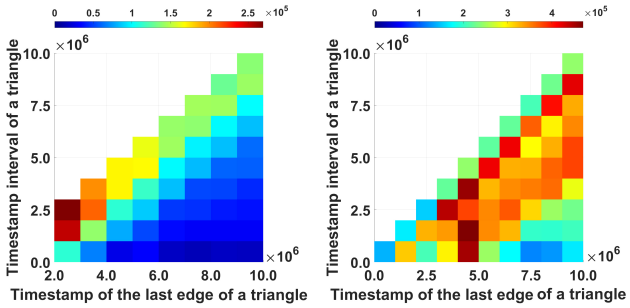
A practical solution for triangle counting in streaming graphs is the sampling-based approximation. It tries to find triangles formed by the sampled edges from the stream and estimates the triangle count based on the probability of discovered triangles. As reviewed in Section 6, existing sampling-based approximation algorithms can be classified into two categories, i.e., fixed probability (FP) based algorithms [1, 27, 36] and fixed-sized memory (FM) based algorithms [18, 39, 50]. The type of FP-based algorithms has the limitations that (i) the space complexity is high since it is related to the number m of the edges in the stream and (ii) it requires some knowledge of the whole edge stream to determine the sampling probability, which may be unavailable. Thus, the FP-based algorithms are not suitable for the streaming graph setting. A lot of research efforts have been devoted to the type of FM-based algorithms that only needs a reservoir of fixed-size k ($k \ll m$) in memory. However, existing FM-based algorithms face challenges related to either accuracy or efficiency.

Accuracy issue. We highlight the limitations of existing FM-based algorithms in terms of accuracy. Most of the algorithms guarantee unbiased estimations, while StreamingTriangle [16] is biased. FURL [18] uses a hash function to get sampled edges that is deterministic. Its accuracy strongly relies on the hash function, which is low in our experiments. Algorithms TRIÈST-B [38, 39],TRIÈST-I [38, 39], and WRS [23, 31] are the state-of-the-art FM-based algorithms for triangle counting estimation in streaming graphs. Previous studies [38, 50] have proven that TRIÈST-I is more accurate than TRIÈST-B. Recent studies [32–34, 48, 49, 49, 51, 51, 53, 54] extend TRIÈST-I and TRIÈST-B to address duplicated edges, deleted edges in the stream, or to adapt to a distributed setting. Thus, the variances of the estimated results of these algorithms are the same as TRIÈST-B and TRIÈST-I.

However, the best known algorithms TRIÈST-I and WRS ignore the dynamic timestamp intervals of the triangles in real-world streaming graphs and their sampled triangles do not follow the real timestamp interval distribution. The timestamp interval of a triangle is defined as the difference between the timestamp of the last arrival edge and that of the first arrival edge in the triangle. Figure 1 shows how the timestamp intervals of triangles change with timestamps in real datasets Wikipedia and Yahoo (used in our experiments). Both axes are divided into 10 sub-ranges. Each colored square represents the number of triangles that are discovered in a sub-range of x-axis and whose timestamp intervals fall in a sub-range of y-axis. In this figure, we have three observations. (O_1) The timestamp interval distribution of the triangles is non-uniform, which means that at any timestamp, the number of discovered triangles varies with different timestamp intervals. (O_2) At any timestamp, the triangles discovered are not always biased to short timestamp intervals. (O_3) The timestamp interval distribution varies over time. TRIÈST-I generates a uniform sample at each timestamp, which is inconsistent with observations O_1 and O_3, resulting in inaccurate estimations. Algorithm WRS has the assumption that the timestamp intervals of the majority of triangles are short and tends to discover triangles with short timestamp intervals, which is not consistent with observations O_2 and O_3, so that WRS produces inaccurate estimations.

To accurately estimate triangle counts in real-world streaming graphs, a desired algorithm should be able to generate edge samples such that the discovered triangles follow a dynamic, non-uniform timestamp interval distribution.

Efficiency issue. We show the shortcomings of existing FM-based estimation algorithms in terms of amortized time complexity. The FM-based estimation algorithms are developed based on the reservoir sampling [20], where the sampled edges are stored in a reservoir. The computational cost of this type of algorithm is dominated by updating the sample graph in the reservoir and counting triangles in the sample graph. As shown in Table 1, FURL-0B [18] has the highest amortized time complexity of updating the sample graph, whereas the other algorithms are fast. Regarding the amortized time complexity of counting triangles, RFES-I [50] is the slowest. The reservoirs in algorithms TRIÈST-I, WRS, and TS-triangle are always full, so that the computational cost of counting triangles is proportional to the reservoir size k, which is a considerable cost. Although StreamingTriangle and TRIÈST-B only need to count triangles when an edge is sampled, leading to amortized time complexity less than $O(k)$, their estimations are inaccurate [38]. Therefore, compared with the triangle counting cost, the sample graph updating cost can be ignored. The amortized time complexity of triangle counting in the best known accurate algorithms TRIÈST-I and WRS is $O(k)$.



(a) Wikipedia. **(b) Yahoo.**

Figure 1: Dynamic timestamp intervals of triangles.

Table 1: Amortized time complexity of FM-based algorithms

Algorithm	Amortized time complexity per edge (u, v).	
	Counting	Updating
StreamingTriangle [16]	$O\left((k + s_w) \cdot p_{uv}^{(s)}\right)$	$O\left(p_{uv}^{(s)}\right)$
TRIÈST-B [38]	$O\left(k \cdot p_{uv}^{(s)}\right)$	$O\left(p_{uv}^{(s)}\right)$
TRIÈST-I [38]	$O(k)$	$O\left(p_{uv}^{(s)}\right)$
WRS [23, 31]	$O(k)$	$O(1)$
TS-triangle [53]	$O(k)$	$O\left(p_{uv}^{(s)}\right)$
FURL-0B [18]	$O(k)$	$O\left(p_{uv}^{(s)} \cdot k \cdot \log k\right)$
RFES-I [50]	$O\left((1 + 2 \cdot d_{avg}) \cdot k\right)$	$O\left(p_{uv}^{(s)}\right)$
GREAT^I	$O\left(\left(1 - \frac{\alpha}{2}\right) \cdot k - \frac{1}{2}\right)$	$O\left(p_{uv}^{(s)}\right)$
GREAT^{II}	$O(k - \mathbb{C})$	$O\left(p_{uv}^{(s)}\right)$
GREAT$^+$	$O(k - \mathbb{C})$	$O\left(p_{uv}^{(s)}\right)$

* s_w is the reservoir size for storing wedges. k is the reservoir size. d_{avg} is the average degree of the vertices in the reservoir. $p_{uv}^{(s)}$ is the sampling probability of edge (u, v). α is the edge removing probability. $\mathbb{C}$ is a positive value.

To tackle the accuracy and the efficiency issues, we first propose the generalized reservoir sampling (GRS) that stores less edges (controlled by parameter α) in the reservoir so that the triangle counting cost is reduced by searching in a smaller sample graph. Next, we propose the Generalized Reservoir Sampling based Triangle counting estimation algorithm (GREAT) that belongs to the type of FM-based algorithms. It adopts the GRS and has a new way for computing the probabilities of discovered triangles, so that it is efficient and can still provide unbiased estimations. Compared with the best known FM-based algorithms, our GREAT algorithm has lower amortized time complexity (see Section 3.3) and smaller variance (see Section 3.4).

By considering the dynamic timestamp intervals of the triangles in real-world streaming graphs, shown in Figure 1, we propose algorithm GREAT$^+$ that improves GREAT by an adaptive strategy. The proposed adaptive strategy automatically adjusts the parameter α in GREAT, so that the timestamp interval distribution of the sampled triangles in the reservoir can adapt to the evolving timestamp interval distribution of the streaming graph. In this way, triangles

with short and long timestamp intervals will be sampled following the ground-truth distribution, resulting in accurate estimations. According to our accuracy analysis, the simplified variance of GREAT$^+$ is smaller than that of GREAT.

Finally, extensive experiments are conducted on four real-world streaming graph datasets to evaluate the performance of our algorithms. The results demonstrate that the proposed algorithm, GREAT, achieves comparable accuracy to its competitors while delivering a significant improvement in efficiency. Moreover, the relative error of our enhanced algorithm, GREAT$^+$, is an order of magnitude lower than that of the competitors, while maintaining efficiency on par with state-of-the-art algorithms.

The rest of the paper is organized as follows: Section 2 presents the problem definition and the framework of triangle counting estimation. Section 3 proposes the generalized reservoir sampling based triangle counting estimation algorithm GREAT. Section 4 improves the GREAT algorithm by an adaptive strategy. Extensive evaluations are conducted in Section 5. Section 6 reviews relevant algorithms and we conclude in Section 7.

2 PRELIMINARIES

2.1 Problem Definition

Streaming graph [2] (a.k.a. edge stream) Σ is an unbounded sequence of edges, where each edge (u, v) has a unique timestamp t_{uv} that is the arrival time of edge (u, v). At any timestamp t, graph $G^{(t)}$ consists of the edges arrived at timestamp t and earlier. Let unordered triple (u, v, w) denote the triangle formed by three edges (u, v), (v, w), (u, w). Let Δ be the set of all triangles in graph $G^{(t)}$ and Δ_u be the set of all local triangles of any vertex u in graph $G^{(t)}$.

Problem statement. We study the problem of estimating the global and the local triangle counts in insertion-only streaming graphs, following the *assumptions* in the most prior work [23, 26, 39] that (i) the unbounded edge stream cannot be physically stored, (ii) memory budget is limited, i.e., at most k edges can be stored in memory, and (iii) the edge stream is processed in single pass, i.e., the edges is processed one by one in the order of their arrival time. Formally,

- **Given:** an edge stream Σ and a memory budget k.
- **Estimate:** at any timestamp t, the global triangle count $\widehat{\tau}$ and the local triangle count $\widehat{\tau}_u$ of any vertex u in graph $G^{(t)}$.
- **Goal:** minimizing estimation errors.

Although we consider the insertion-only streaming graph in this paper, the proposed algorithms can be easily extended to handle the edge stream containing deleted and duplicated edges [33, 38, 45]. We leave this extension as the future work, due to space limitations.

2.2 Reservoir Sampling based Framework

Most existing triangle counting estimation algorithms [18, 32–34, 36, 38, 45, 48–51, 53, 54] adopt traditional reservoir sampling [20] since it satisfies the three assumptions in our problem statement (Section 2.1) and can generate a uniformly random sample of the edge stream. Figure 2 shows the reservoir sampling based framework for triangle counting estimation. The framework processes the edges from the input stream one by one and includes the following two components:

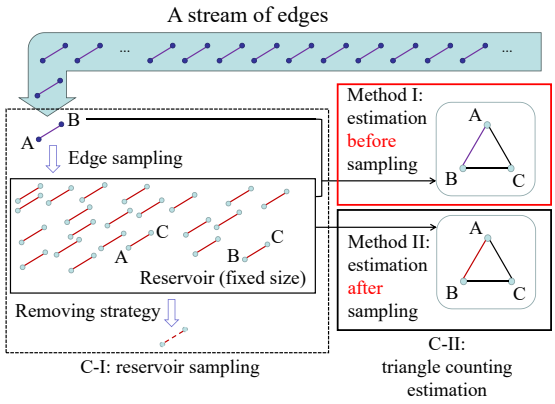


Figure 2: Reservoir sampling based framework.

C-I: reservoir sampling. It adopts traditional reservoir sampling. A reservoir S that can store at most k edges is maintained in memory. For edge (u, v) that arrives at timestamp t_{uv}, it is sampled with probability k/t_{uv}. The sampled (selected) edges are stored in the reservoir. When the reservoir is full, one random edge in the reservoir is removed to free space for the next sampled edge. At any timestamp, the edges in the reservoir construct a uniformly random sample of all arrived edges. In other words, the edges in the reservoir are not temporally biased.

C-II: triangle counting estimation. For each edge from the input stream, we say one triangle is discovered, if it can form a triangle with two edges in the reservoir. At any timestamp, global and local triangle counts are estimated according to all the discovered triangles so far and the corresponding triangle discovering probabilities. In the literature, there are two methods of estimation. Method I estimates triangle counts before edge sampling. It takes the current edge into account no matter whether it will be stored in the reservoir or not. In other words, the current edge may contribute to discovered triangles even though it may not be selected in the edge sampling process after. Method II estimates triangle counts after edge sampling. In this way, only the edges sampled and stored in the reservoir will contribute to discovered triangles.

Previous studies [38, 50] have proven that Method I is more accurate than Method II. Therefore, our algorithms follow the framework using Method I for triangle counting estimation.

3 GENERALIZED RESERVOIR SAMPLING BASED TRIANGLE COUNTING ESTIMATION

As discussed before, existing reservoir sampling based triangle counting estimation algorithms suffer from the efficiency and the accuracy issues. To efficiently and accurately estimate triangle counts in streaming graphs, we first propose the generalized reservoir sampling (GRS) in this Section 3.1 that keeps less edges in the reservoir for reducing the computational cost. Then, we propose a triangle counting estimation algorithm based on GRS named GREAT in Section 3.2. The analyses of amortized time complexity and accuracy are provided in Sections 3.3 and 3.4, respectively.

3.1 Generalized Reservoir Sampling (GRS)

Recall that traditional reservoir sampling maintains a reservoir of size k and adopts the removing strategy that one random edge in

the reservoir is removed when the reservoir is full. The proposed generalized reservoir sampling (GRS) uses a new removing strategy that the edges in the reservoir are removed with probability α, $1/k < \alpha < 1$. GRS removes more than one edges when the reservoir is full. Traditional reservoir sampling is a special case of GRS when setting $\alpha = 1/k$ (in this case, the expected number of edges removed is $\alpha \cdot k = 1$). Since the average number of edges retained in the GRS reservoir is lower than that in traditional reservoir sampling, the computational cost of finding triangles for each incoming edge is reduced.

Algorithm 1 GRS

Input: reservoir S, edge (u, v), round r, edge counter t.

1: **if** $t \leq k$ **then** ▷ computational round 0.
2: Add edge (u, v) to S;
3: $p_{uv}^{(s)} \leftarrow 1$; $r_{uv} \leftarrow r$;
4: **else** ▷ computational round r ($r > 0$).
5: **if** S is full **then** ▷ The reservoir is full.
6: The edges in S are removed with probability α;
7: $r \leftarrow r + 1$;
8: **end if**
9: Generate a random number $p \in (0, 1)$;
10: Compute $p_{uv}^{(s)}$;
11: **if** $p < p_{uv}^{(s)}$ **then**
12: Add edge (u, v) to S;
13: $r_{uv} = r$;
14: **end if**
15: **end if**

We partition the computational process of GRS into several rounds. Computational round 0 is the initial round in GRS, which begins with an empty reservoir and concludes once the reservoir is full. Computational round r (> 0) starts at the end of round $r - 1$, proceeds with edge removal operations, and ends once the reservoir is refilled with sampled edges. Then, we define r_{uv} as the computational round in which edge (u, v) arrives. Algorithm 1 shows the pseudo code of GRS. Lines 1–3 denote computational round 0, where each arrived edge is sampled with probability 1. Lines 5–8 detail the edge removal operations. Lines 9–14 illustrate any computational round r (> 0), where the sampling probability of edge (u, v) is set to $p_{uv}^{(s)}$.

3.2 GRS based Triangle Counting Estimation Algorithm GREAT

This section proposes the Generalized REservoir sAmpling based Triangle counting estimation algorithm (GREAT) that estimates global and local triangle counts based on GRS.

First, we introduce how to calculate the probabilities of the discovered triangles in GRS in Lemma 3.1.

Lemma 3.1. *Given an edge (u, v) from the stream in computational round r, the probability of discovering triangle (u, v, w) in GRS is calculated as*

$$P_{uvw} = p_{uw}^{(s)} \cdot p_{vw}^{(s)} \cdot (1 - \alpha)^{2r - r_{uw} - r_{vw}}. \quad (1)$$

Proof. According to GRS, edge (u, w) is sampled with probability $p_{uw}^{(s)}$ in computational round r_{uw}. Then, in computational round r ($\geq r_{uw}$), the probability of edge (u, w) staying in the reservoir is

$$p_{uw}^{(r)} = p_{uw}^{(s)} \cdot (1 - \alpha)^{r - r_{uw}}. \quad (2)$$

Similarly, in computational round r, the probability of edge (v, w) staying in the reservoir is $p_{vw}^{(r)} = p_{vw}^{(s)} \cdot (1 - \alpha)^{r - r_{vw}}$. Hence, for the current edge (u, v) in computational round r, the probability of discovering triangle (u, v, w) formed by edge (u, v) and two edges in the reservoir is $P_{uvw} = p_{uw}^{(r)} \cdot p_{vw}^{(r)} = p_{uw}^{(s)} \cdot p_{vw}^{(s)} \cdot (1-\alpha)^{2r - r_{uw} - r_{vw}}$. □

Then, having the probability of each discovered triangle, the estimated global and the estimated local triangle counts are calculated as follows. Let $\widehat{\Delta}$ be the set of discovered triangles and $\widehat{\Delta}_u$ be the set of discovered local triangles of vertex u. The global triangle count is estimated as

$$\widehat{\tau} = \sum_{(u,v,w) \in \widehat{\Delta}} \frac{1}{P_{uvw}}. \quad (3)$$

The local triangle count of vertex u is estimated as

$$\widehat{\tau}_u = \sum_{(u,v,w) \in \widehat{\Delta}_u} \frac{1}{P_{uvw}}. \quad (4)$$

Algorithm TriangleCountingEst (Algorithm 2) shows the pseudo code of GRS based triangle counting estimation. In the reservoir, each vertex u has a hashset that stores the neighbors of u. It first tries to find triangles formed by the given edge (u, v) and two edges in the reservoir, i.e., finding the common neighbors of vertices u and v in the reservoir (line 1). The size of the common neighbor set W is the number of discovered triangles for edge (u, v). Then, the probability of each discovered triangle (u, v, w) is computed according to Lemma 3.1 (line 3). In the end, according to Equations 3 and 4, the global the local triangle counts are updated using the probabilities of the discovered triangles (lines 4–7).

Algorithm 2 TriangleCountingEst

Input: reservoir S, edge (u, v), round r.

1: $W \leftarrow S.\textbf{getNeighbor}(u) \cap S.\textbf{getNeighbor}(v)$;
2: **for** each vertex $w \in W$ **do**
3: $P_{uvw} = p_{uw}^{(s)} \cdot p_{vw}^{(s)} \cdot (1 - \alpha)^{2r - r_{uw} - r_{vw}}$; ▷ Lemma 3.1;
4: $\widehat{\tau}_u \leftarrow \widehat{\tau}_u + p_{uvw}^{-1}$;
5: $\widehat{\tau}_v \leftarrow \widehat{\tau}_v + p_{uvw}^{-1}$;
6: $\widehat{\tau}_w \leftarrow \widehat{\tau}_w + p_{uvw}^{-1}$;
7: $\widehat{\tau} \leftarrow \widehat{\tau} + p_{uvw}^{-1}$;
8: **end for**

Lately, the proposed algorithm (GREAT) follows the framework in Figure 2, which uses Algorithm 1 in component C-I and Algorithm 2 in component C-II. Following existing algorithms [30, 39, 50], GREAT adopts Method I that performs estimations before edge sampling in the framework since this method has lower variances than Method II. Algorithm 3 shows the pseudo code of algorithm GREAT. For each arrived edge (u, v), GREAT first calls TriangleCountingEst (Algorithm 2) that updates the estimated triangle counts (line 6), and then calls GRS (Algorithm 1) that performs edge sampling and maintains the reservoir (line 7).

In algorithm GREAT, we investigate the following two ways (P-I and P-II) of calculating the edge sampling probability $p_{uv}^{(s)}$ in GRS, leading to two specialized algorithms: GREAT^{I} that employs probability P-I and GREAT^{II} utilizes probability P-II. According

Algorithm 3 GREAT

Input: an edge stream Σ, GRS parameter α, a reservoir S of size k.
Output: the estimated global triangle count $\widehat{\tau}$, the estimated local triangle count $\widehat{\tau}_u$ of vertex u, $\forall u \in V$.

```
S ← ∅; τ̂ ← 0; τ̂_u ← 0, ∀u ∈ V;
Edge counter t ← 0;
computational round r ← 0.
while the next edge (u, v) from Σ do
    t ← t + 1;
    TriangleCountingEst(S, (u, v), r);
    GRS(S, (u, v), r, t);
end while
return τ̂ and {τ̂_u};
```

to Sections 3.3 and 3.4, GREAT^I is faster while GREAT^{II} is more accurate.

P-I: $p_{uv}^{(s)} = (1-\alpha)^{r_{uv}}$. This edge sampling probability has been used in butterfly counting estimation [30]. We are the first to adopt it for triangle counting estimation. Lemma 3.2 guarantees that if adopting P-I, all the arrived edges stay in the reservoir with equal probability. Thus, algorithm GREAT^I generates a uniformly random sample of edge stream.

Lemma 3.2. *If applying $p_{uv}^{(s)} = (1-\alpha)^{r_{uv}}$, in any computational round r, all the arrived edges stay in the reservoir with probability $(1-\alpha)^r$.*

The proof is available in the supplementary material [47].

P-II: $p_{uv}^{(s)} = k/t_{uv}$. Existing triangle counting estimation algorithms [23, 39, 50] adopt this edge sampling probability. Lemma 3.3 shows that if adopting P-II, the probability of each arrived edge staying in the reservoir is approximate to k/t. Thus, algorithm GREAT^{II} generates an approximate uniformly random sample of edge stream.

Lemma 3.3. *Let $p_{uv}^{(s)} = k/t_{uv}$, in any computational round r, t be the number of arrived edges, and $p_{uv}^{(r)}$ be the probability that any arrived edge (u,v) staying in the reservoir. If $\alpha \le 0.7$,*

$$\left| p_{uv}^{(r)} - p^* \right| / p^* \le 1 - \exp(-2\alpha), \quad p^* = k/t. \tag{5}$$

The proof is available in the supplementary material [47].

3.3 Amortized Time Complexity

This section analyzes the amortized numbers of computational operations of algorithms GREAT^I and GREAT^{II}. In both algorithms, discovering triangles, i.e., finding the common neighbors of vertices u and v for each edge (u,v) (line 1 in Algorithm 2), dominates the computational cost. Thus, we proceed to derive the amortized time complexities of these dominant parts in both algorithms.

Lemmas 3.4 and 3.6 show the expected numbers of edges arrived in any computational round r in algorithms GREAT^I and GREAT^{II}, respectively. Lemmas 3.5 and 3.7 show the expected numbers of edges arrived to put one sampled edge in the reservoir in algorithms GREAT^I and GREAT^{II}, respectively.

Lemma 3.4. *In algorithm GREAT^I, the expected number of edges arrived in computational round r is $x_r^I = \alpha \cdot k/(1-\alpha)^r$.*

Lemma 3.5. *In algorithm GREAT^I, at the beginning of computational round r, suppose that there are Q free slots in the reservoir where each slot can store one edge. The expected number of edges arrived to put one sampled edge in the i^{th} slot is $y_{r,i}^I = 1/(1-\alpha)^r$.*

Lemma 3.6. *In algorithm GREAT^{II}, the expected number of edges arrived in computational round r is $x_r^{II} = (\exp(\alpha)-1)\cdot\exp((r-1)\cdot\alpha)\cdot k$.*

Lemma 3.7. *In algorithm GREAT^{II}, at the beginning of computational round r, suppose that there are Q free slots in the reservoir where each slot can store one edge. The expected number of edges arrived to put one sampled edge in the i^{th} slot is*

$$y_{r,i}^{II} = k\cdot\exp(\alpha\cdot(r-1))\cdot\left(\exp\left(\frac{1}{k}\right)-1\right)\cdot\exp\left(\frac{1}{k}\cdot(i-1)\right).$$

The proofs of Lemmas 3.4–3.7 are available in the supplementary material [47].

In algorithm GREAT^I, at the beginning of computational round r, the expected number of empty slots in the reservoir is $\alpha\cdot k$. Then, in this computational round, we expect that x_r^I (Lemma 3.4) edges arrive and $\alpha\cdot k$ edges are sampled and stored in the reservoir. According to Lemma 3.5, when finding the common neighbors of the vertices for $y_{r,i}^I$ edges, $(1-\alpha)\cdot k+i-1$ computational operations are needed. Thus, for the arrived edges in computational round r, the number of computational operations needed in algorithm GREAT^I is

$$OP_r^I = \sum_{i=1}^{\alpha\cdot k} y_{r,i}^I\cdot((1-\alpha)\cdot k+i-1) = \frac{2\alpha k^2-\alpha^2k^2-\alpha k}{2\cdot(1-\alpha)^r}.$$

According to Lemma 3.4, algorithm GREAT^I processes x_r^I edges in computational round r. Thus, the amortized number of computational operations of GREAT^I is

$$\frac{OP_r^I}{x_r^I} = \left(1-\frac{\alpha}{2}\right)k-\frac{1}{2}. \tag{6}$$

Similarly, for the arrived edges in computational round r, the number of computational operations needed in algorithm GREAT^{II} is

$$\begin{aligned} OP_r^{II} &= \sum_{i=1}^{\alpha\cdot k} y_{r,i}^{II}\cdot((1-\alpha)\cdot k+i-1) \\ &= \exp(\alpha(r-1))\left[(\alpha+\exp(\alpha)-1)+\frac{\exp\left(\frac{1}{k}\right)(1-\exp(\alpha))}{\exp\left(\frac{1}{k}\right)-1}\right]k^2. \end{aligned}$$

According to Lemma 3.6, algorithm GREAT^{II} processes x_r^{II} edges in computational round r. Thus, the amortized number of computational operations of GREAT^{II} is

$$\frac{OP_r^{II}}{x_r^{II}} = \left(1+\frac{\alpha}{\exp(\alpha)-1}\right)k-\frac{\exp\left(\frac{1}{k}\right)-\frac{1}{k}}{\exp\left(\frac{1}{k}\right)-1} \le k-\mathbb{C}, \tag{7}$$

where $\mathbb{C} = \frac{\exp(\frac{1}{k})-\frac{1}{k}}{\exp(\frac{1}{k})-1} > 1$.

Summary. According to Equations 6 and 7, the amortized numbers of computational operations of our algorithms are lower than that of the competitor algorithms in Table 1. As α increases, the amortized

numbers of computational operations of our algorithms decrease. Comparing Equations 6 and 7, GREAT^{I} is faster than GREAT^{II} since $\frac{OP_r^{II}}{x_r^{II}} - \frac{OP_r^{I}}{x_r^{I}} > 0$.

3.4 Accuracy Analysis

In this section, we analyze the accuracy of our algorithms and compare our algorithms with two best known algorithms TRIÈST-I and WRS.

First, we prove that our algorithm provide unbiased estimations for the global and the local triangle counts, shown in Lemma 3.8.

Lemma 3.8. *In algorithm* GREAT*, Equations 3 and 4 return unbiased estimations for the global and the local triangle counts, i.e.*

$$\mathbb{E}(\widehat{\tau}) = \tau \text{ and } \mathbb{E}(\widehat{\tau_u}) = \tau_u, \forall u \in V.$$

Proof. Given edge stream Σ, at any timestamp t, the edges arrive no later than t are called seen edges and the edges arrive after t are called unseen edges. Given any timestamp t, let Δ be the ground truth set of global triangles formed by seen edges and $\tau = |\Delta|$ is the ground truth global triangle count. Next, we prove that $\mathbb{E}(\widehat{\tau}) = \tau$. Let I_{uvw} be an indicator variable, such that

$$I_{uvw} = \begin{cases} 1 & \text{if triangle } (u,v,w) \text{ is discovered,} \\ 0 & \text{otherwise.} \end{cases}$$

Then, we have that $\mathbb{E}(I_{uvw}) = P_{uvw} \times 1 + (1 - P_{uvw}) \times 0 = P_{uvw}$. The expectation of the estimated global triangle count is

$$\mathbb{E}(\widehat{\tau}) = \mathbb{E}\left(\sum_{(u,v,w)\in\widehat{\Delta}} \frac{1}{P_{uvw}}\right) = \mathbb{E}\left(\sum_{(u,v,w)\in\Delta} \frac{I_{uvw}}{P_{uvw}}\right)$$
$$= \sum_{(u,v,w)\in\Delta} \frac{\mathbb{E}(I_{uvw})}{P_{uvw}} = \sum_{(u,v,w)\in\Delta} \frac{P_{uvw}}{P_{uvw}} = \sum_{(u,v,w)\in\Delta} 1 = \tau.$$

Hence, $\widehat{\tau}$ computes unbiased estimation for the global triangle count.

Similarly, we prove that $\widehat{\tau_u}, \forall u \in V$ computes unbiased estimation for the local triangle count. Given any timestamp t, let Δ_u be the ground truth set of local triangles of vertex u formed by seen edges and $\tau_u = |\Delta_u|$ is the ground truth local triangle count of vertex u.

$$\mathbb{E}(\widehat{\tau_u}) = \mathbb{E}\left(\sum_{(u,v,w)\in\Delta_u} \frac{I_{uvw}}{P_{uvw}}\right) = \sum_{(u,v,w)\in\Delta_u} \frac{\mathbb{E}(I_{uvw})}{P_{uvw}} = \tau_u.$$

□

Given that our algorithm and most of existing reservoir sampling based triangle estimation algorithms are unbiased [18, 31, 39], we proceed to compare the variances of the estimated global and the estimated local triangle counts in different algorithms.

Following previous work [23], we utilize the simplified variance $\tilde{Var}(\widehat{\tau})$ to evaluate the accuracy of different algorithms because it is has been shown that traditional variance and the simplified variance are strongly correlated ($R^2 > 0.99$) in real-world graphs.

$$\tilde{Var}(\widehat{\tau}) = \sum_{X\in\Delta} Var\left(\frac{I_X}{P_X}\right) = \sum_{X\in\Delta}\left(\frac{1}{P_X} - 1\right), \tag{8}$$

where $\widehat{\tau}$ is an estimated triangle count, X is any triangle in the ground truth triangle set Δ, I_X is a variable that indicates whether triangle X is discovered or not, and P_X is the probability of discovering triangle X.

Given a triangle $X = (x_1, x_2, x_3) \in \Delta$, where x_i is the i^{th} arrival edge of triangle X, the simplified variances of X of algorithms WRS, TRIÈST-I, and our algorithms GREAT^{I} and GREAT^{II} are shown below (according to Equation 8).

$$\begin{aligned}
\text{GREAT}^{I} &: Var^{I}\left(\tfrac{I_X}{P_X}\right) = (1-\alpha)^{-2r_{x_3}} - 1; \\
\text{GREAT}^{II} &: Var^{II}\left(\tfrac{I_X}{P_X}\right) = \tfrac{t_{x_1} t_{x_2}}{k^2} \cdot (1-\alpha)^{-(2r_{x_3} - r_{x_1} - r_{x_2})} - 1; \\
\text{TRIÈST-I} &: Var^{TRI}\left(\tfrac{I_X}{P_X}\right) = \tfrac{t_{x_3}(t_{x_3}-1)}{k(k-1)} - 1 \text{ [23, 39].} \\
\text{WRS} &: Var^{WRS}\left(\tfrac{I_X}{P_X}\right) = 0 \text{ (Case 1)}, \quad \tfrac{\tilde{t}_{x_3}}{(1-\gamma)k} - 1 \text{ (Case 2)}, \\
&\quad \tfrac{\tilde{t}_{x_3}(\tilde{t}_{x_3}-1)}{(1-\gamma)k((1-\gamma)k-1)} - 1 \text{ (Case 3) [23],}
\end{aligned}$$

where r_{x_i} is the computational round where edge x_i arrives in the corresponding algorithm, t_{x_i} is the timestamp of edge x_i, $\tilde{t}_{x_3} = t_{x_3} - k \cdot \gamma$, and γ specifies the waiting room size in WRS.

Lemma 3.9. *Let* $\tilde{Var}^{TRI}(\widehat{\tau})$, $\tilde{Var}^{I}(\widehat{\tau})$, *and* $\tilde{Var}^{II}(\widehat{\tau})$ *be the simplified variances of algorithms TRIÈST-I,* GREAT^{I}*, and* GREAT^{II}*, respectively. We have*

$$\tilde{Var}^{TRI}(\widehat{\tau}) > \tilde{Var}^{I}(\widehat{\tau}) > \tilde{Var}^{II}(\widehat{\tau}). \tag{9}$$

The proof is available in the supplementary material [47].

The simplified variance of WRS [23] consists of three cases. Case 1 and Case 2 cover the triangles with short timestamp intervals. Case 3 includes the triangles with long timestamp intervals. It is been shown that the simplified variance of WRS in Case 3 is the same as that of TRIÈST-I. Thus, our algorithms are more accurate than WRS in real-world streaming graphs where a considerable number of triangles have long timestamp intervals.

Sensitivity of α. When using a large α, more edges are removed from the reservoir in each computational round, reducing the number of edges processed for triangle counting and thereby saving computational operations. However, a large α increases the probability of an edge being discarded, making it less likely to detect triangles with long time intervals compared to those with short time intervals. In contrast, when using a small α, fewer edges are removed from the reservoir per computational round, resulting in more edges being processed for triangle counting and increasing computational costs. However, a small α decreases the probability of an edge being discarded, improving the likelihood of detecting triangles with long time intervals compared to those with short time intervals. Therefore, to balance efficiency and accuracy, very small values of α should be avoided. Instead, α can be tuned based on the characteristics of the dataset to achieve good estimations.

4 ALGORITHM GREAT WITH ADAPTIVE STRATEGY

Recall from Figure 1 that existing algorithms ignore the dynamic timestamp intervals of the triangles in real-world streaming graphs, resulting in inaccurate estimations. To address this issue, we propose algorithm GREAT^{+} that is an extension of algorithm GREAT. It has an adaptive strategy that automatically adjusts α to adapt to

the changes of the timestamp intervals of the triangles in the edge stream, so that the estimation accuracy is improved.

We first introduce the parameter z in algorithm GREAT$^+$. Let $z = (1-\alpha)^y$, where $y = 2r_{x_3} - r_{x_1} - r_{x_2}$. We call variable y as the computational round interval of a discovered triangle. The computational round interval is similar to the timestamp interval, which indicates how long it takes to discover a triangle. Based on the observation in Figure 1, the distribution of the computational round interval y is non-uniform and the distribution of y changes over time. In algorithm GREAT, since parameter α is fixed and the value of z changes with y, the variance of the estimation is not stable. In algorithm GREAT$^+$, we can set z to a proper value to obtain the desired variance. In this way, as y changes over time, α can be adjusted accordingly.

Given z, we proceed to present the proposed adaptive strategy that automatically adjusts α per computational round. We convert $z = (1-\alpha)^y$ into the following form:

$$\alpha^{(r)} = 1 - \sqrt[y^{(r)}]{z},\ y \geq 0, \tag{10}$$

where $\alpha^{(r)}$ denotes the value of α in computational round r and $y^{(r)}$ is the average value of the computational round intervals of the discovered triangles in the computational round $r-1$, i.e.,

$$y^{(r)} = \frac{\sum_{X=(x_1,x_2,x_3)\in\Delta^{(r-1)}} 2r_{x_3} - r_{x_1} - r_{x_2}}{|\Delta^{(r-1)}|}, \tag{11}$$

where $\Delta^{(r-1)}$ is the set of discovered triangles in computational round $r-1$. The adaptive strategy uses Equation 10 to compute the value of α used in computational round r. The idea of Equation 11 is that the computational round intervals of triangles may be similar in two consecutive computational rounds.

Next, we discuss how to determine the value of z. Figure 3 plots Equation 10 using different values of z. Observe that when z is large, e.g., 0.7 and 0.9, α decreases rapidly as y increases and falls within a range of very small values for the majority of the values of y. We should avoid very small α since the computational cost cannot be reduced according to Equations 6 and 7. Therefore, we suggest setting $z \leq 0.5$ and the lower bound of α to 0.1.

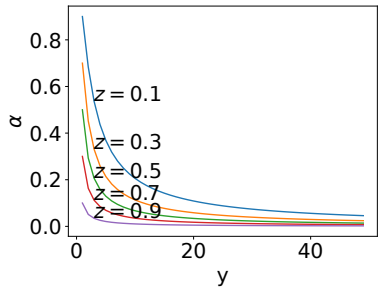


Figure 3: Equation 10 using different values of z.

In the adaptive strategy, we should avoid obtaining large α. The reason is as follows. In the first a few computational rounds, $y^{(r)}$ tends to be small, so that $\alpha^{(r)}$ might be set to large values using Equation 10. In this case, a large portion of edges will be removed when the reservoir is full. Then, the value of $y^{(r)}$ will remain consistently small in the computational rounds followed. As a result, the triangles with long computational round intervals will be missed. To avoid this situation, we propose a heuristic setting that we set $\alpha^{(r)} = 0.1$ in the first λ computational rounds, $0 \leq r \leq \lambda$. In our experiments, we use $\lambda = 5$.

Accuracy analysis. It is straightforward to derive that algorithm GREAT$^+$ provides unbiased estimations following Lemma 3.8. Because GREAT^{II} has a smaller simplified variance than GREAT^I, we derive the simplified variance of GREAT$^+$ based on GREAT^{II}. The simplified variance of GREAT$^+$ of a triangle $X = (x_1, x_2, x_3) \in \Delta$ is

$$Var^+\left(\frac{I_X}{P_X}\right) = \frac{1}{P_X} - 1$$

$$= \left[\frac{t_{x_1}}{k} \cdot \prod_{r^{II}_{x_1} \leq r \leq r^{II}_{x_3}} \left(1-\alpha^{(r)}\right)^{-1} \cdot \frac{t_{x_2}}{k} \prod_{r^{II}_{x_2} \leq r \leq r^{II}_{x_3}} \left(1-\alpha^{(r)}\right)^{-1}\right] - 1,$$

According to the setting of parameter z in GREAT$^+$, we have

$$\min_{X\in\Delta}\left[\prod_{r^{II}_{x_1} \leq r \leq r^{II}_{x_3}} \left(1-\alpha^{(r)}\right) \cdot \prod_{r^{II}_{x_2} \leq r \leq r^{II}_{x_3}} \left(1-\alpha^{(r)}\right)\right] \geq z.$$

Therefore, the simplified variance of GREAT$^+$ can be bounded:

$$Var^+\left(\frac{I_X}{P_X}\right) \leq \frac{t_{x_1}t_{x_2}}{k^2} \cdot \frac{1}{z} - 1.$$

Then, we have

$$Var^+\left(\frac{I_X}{P_X}\right) < Var^{II}\left(\frac{I_X}{P_X}\right),\ 2r^{II}_{x_3} - r^{II}_{x_2} - r^{II}_{x_1} > \frac{\ln(z)}{\ln(1-\alpha)}. \tag{12}$$

Recall that α is the parameter of GREAT^{II} and z is the parameter of GREAT$^+$. Given α and z, according to Equation 12, if a triangle (x_1, x_2, x_3) satisfies $2r^{II}_{x_3} - r^{II}_{x_2} - r^{II}_{x_1} > \frac{\ln(z)}{\ln(1-\alpha)}$ in GREAT^{II}, the simplified variance of this triangle in GREAT$^+$ is reduced. For example, consider the parameter settings used in the experiment: $z = 0.25$ and $\alpha = 0.1$. In this case, triangles that satisfy $2r^{II}_{x_3} - r^{II}_{x_2} - r^{II}_{x_1} > 13$ in GREAT^{II} have smaller simplified variances in GREAT$^+$.

Sensitivity of z. Compared to GREAT^{II} with a given α, using a large z in GREAT$^+$ results in a smaller value of $\frac{\ln(z)}{\ln(1-\alpha)}$. Consequently, more triangles satisfy Equation 12 in GREAT^{II}, leading to a reduction in the simplified variances of a greater number of triangles in GREAT$^+$. However, as shown in Figure 3 and discussed earlier, to effectively handle the dynamic timestamp intervals of triangles represented by y, it is important to avoid using a large z. This allows α to be adjusted within a reasonable range, striking a balance between efficiency and accuracy.

Remarks. We discuss the behavior of GREAT$^+$ when timestamp intervals in the stream change, and explain why GREAT$^+$ can achieve higher accuracy. As shown in Figure 1, various types of triangles exist in the stream, including those with short, medium, and long timestamp intervals. The variable y in Equation 11 identifies the type of triangle that is most prevalent in the stream at any given moment. Due to the dynamic nature of timestamp intervals, the value of y changes over time. According to Equation 10, with z held constant, α is a function of y. For example, when triangles with long timestamp intervals are most frequent, y becomes large and α is assigned a small value. As a result, fewer edges in the reservoir are removed, allowing more old edges to remain. This increases the likelihood of discovering triangles with long timestamp intervals.

Conversely, if triangles with shorter timestamp intervals become more frequent, y becomes small and α is assigned a larger value, which increases the likelihood of discovering triangles with short timestamp intervals. According to Equation 8, as the discovery probabilities of more triangles increase, the simplified variance decreases.

5 EXPERIMENT

5.1 Setup

Datasets. We evaluate the performance of our algorithms and competitors on four real-world streaming graphs. Dataset Review [15] is a network of the user reviews of Amazon products from 1997 to 2018. Dataset Yahoo [4] is a network of ratings for songs on Yahoo. Dataset StackOverflow [21] is an interaction network of questions and answers from Stack Overflow website. Dataset Wikipedia [21] is a network that represents user-page modification interactions on English Wikipedia website. We remove duplicated edges from all datasets. Table 2 shows the statistics of the four datasets.

Table 2: Statistics of Datasets.

Dataset	#Vertices	#Edges	#Triangles	Flow rate (second/edge)
Review [15]	359,501	4,873,540	333,661	0.01
Yahoo [4]	1,000,990	256,804,235	7,163,094,656	0.75
StackOverflow [21]	2,601,977	63,497,050	114,571,929	0.12
Wikipedia [21]	42,541,517	572,591,272	881,439,081	0.49

Competitors. We compare the proposed algorithms to four competitors: one state-of-the-art fixed probability based algorithm MASCOT [26] and three representative fixed-sized memory based algorithms TRIÈST-I [39], FURL-0B [18], and WRS [23].

The reasons why excluding recent competitors are as follows. RFES-I [50] did not return results within 10 hours on dataset Wikipedia. The performance of TRIÈST-B [39] is worse than that of TRIÈST-I. PartitionCT [45] is an extension of TRIÈST-B for dealing with duplicated edges, DVHT-b [51] and DEHT-b [49] are distributed versions of TRIÈST-B. The Hyperedge-based sampling [54] extends TRIÈST-B to hypergraph. TS-triangle [53] extends TRIÈST-I to estimate both triangle counts and vertex degrees. Tri-Fly [32], CoCos [34], DEHT-i [51], DVHT-i [49], ThinkD$_{ACC}$ [33] and TbEC [48] are all distributed versions of TRIÈST-I.

Metric. We evaluate the accuracy of estimated global triangle counts of our algorithms and the competitors in terms of relative error $RE = |\widehat{\tau} - \tau| / \tau$. The accuracy of the estimated local triangle count of each vertex is evaluated in terms of the Local Average Percentage Error (LAPE) [51], i.e., $LAPE = \frac{1}{|V|} \sum_{v \in V} \frac{|\tau_v - \widehat{\tau}_v|}{\tau_v + 1}$. The efficiency of our algorithms and the competitors is evaluated in terms of the elapsed time that is the duration required to process the entire dataset. The reported results are the average values of 10 trials on each dataset. In addition, the average edge flow rates of the four datasets are shown in Table 2. The average estimation time in our algorithms is less than 10^{-5} second/edge, shown in Figure 4. Thus, our algorithms are fast enough to make estimations before the next edge arrives in the stream.

Parameters. The common parameter of all the algorithms is reservoir size (a.k.a. memory budget size) k. By default, we set $k = 10^6$ for datasets Yahoo, StackOverflow and Wikipedia and set $k = 10^5$ for dataset Review. We also evaluate the performance of all the algorithms when varying k. Algorithm MASCOT has unlimited memory budget. To have a fair comparison, we set its edge sampling probability to k/m, where m is the total number of edges in the dataset. In this way, the expected memory size of MASCOT is the same as all the other algorithms. For algorithm WRS, the ratio of the waiting room to the reservoir is set to $\gamma = 0.1$ that is suggested in their work. In algorithms GREAT^{I} and GREAT^{II}, we set $\alpha = 0.1$. In algorithm GREAT^{+}, we set $z = 0.25$ for all datasets. We also evaluate the performance of our algorithms when varying α and z.

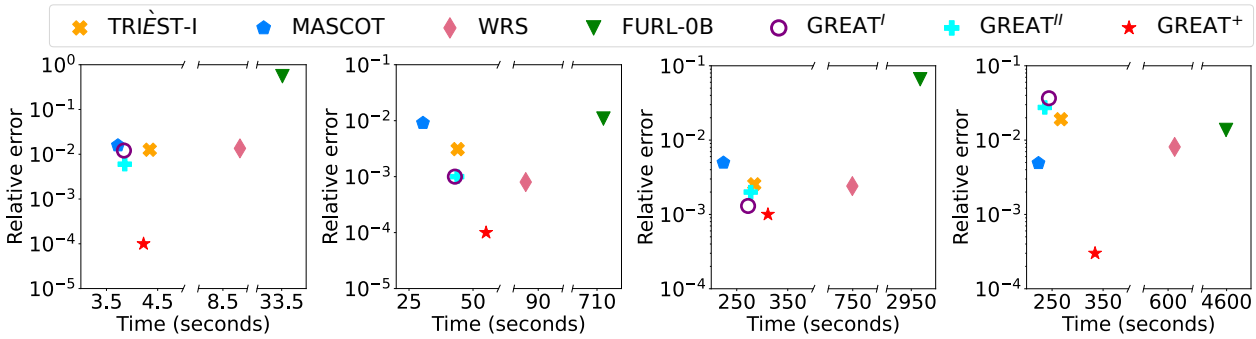
Setting. We implement our algorithms, TRIÈST-I, and MASCOT using Java. The source code of WRS and FURL-0B is provided by their authors. All the algorithms are running on a machine with 4 Intel Xeon E7-4830 CPUs (56 cores, 2.0 GHz) and 2 TB memory and with Ubuntu 16.04.7 LTS. Note that the memory size needed by our algorithms is determined by the budget size k. For example, the largest dataset Wikipedia costs 35740 MB memory during our algorithms, which is far less than the machine memory.

5.2 Results

5.2.1 Accuracy vs. Efficiency. Figure 4 compares our algorithms with the competitors from two perspectives where the x-axis is the elapsed time and the y-axis is the relative error. The same, Figure 5 compares our algorithms with the competitors according to the LAPE and the elapsed time.

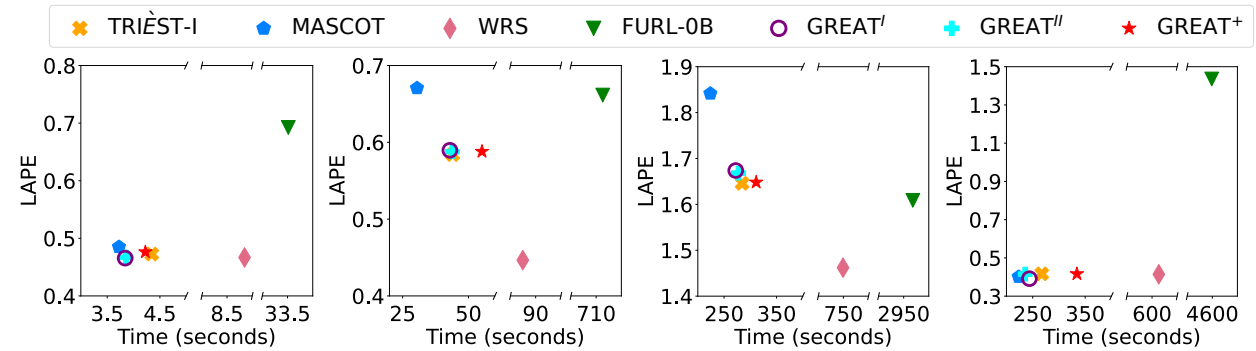
Accuracy. With respect to the relative error of the global triangle count, GREAT^{+} beats all the competitors across all datasets as Figure 4 shows. It is because GREAT^{+} adaptively generates samples based on the dynamic timestamp interval distribution in the stream. WRS assumes the temporal locality in the stream and tends to discover triangles with short timestamp intervals. TRIÈST-I generates uniform edge samples that is inconsistent with the dynamic timestamp interval distribution. Using the same amount of memory as the other algorithms, MASCOT samples much less edges, leading to inaccurate estimations. FURL-0B uses Method II in the framework (Section 2.2), i.e., performing estimations after edge sampling. Therefore, it discovers considerable less triangles than the other algorithms, resulting in inaccurate estimations. The relative errors of GREAT^{I} and GREAT^{II} are similar on all datasets. GREAT^{I} exhibits the second smallest relative error on the Yahoo dataset and GREAT^{II} achieves the second smallest relative error on datasets Review. Although GREAT^{I} and GREAT^{II} are comparable to WRS in some cases and TRIÈST-I in terms of relative error, these two methods are faster.

In terms of the accuracy of the estimated local triangle count as Figure 5 shows, MASCOT and FURL-0B exhibit poor LAPE performance, while WRS yields the best results. The algorithms GREAT^{I}, GREAT^{II}, GREAT^{+}, and TRIÈST-I have similar LAPE values, but GREAT^{I} and GREAT^{II} are slightly faster than TRIÈST-I. Our proposed algorithms GREAT^{I}, GREAT^{II}, and GREAT^{+} are slightly inferior to WRS in terms of accuracy, but offer significantly better efficiency. The LAPE is generally correlated with the number of discovered vertices; that is, as the number of discovered vertices increases, the LAPE tends to decrease. This is because a higher number of discovered vertices allows for more local triangles to be



(a) Review. (b) StackOverflow. (c) Yahoo. (d) Wikipedia.

Figure 4: Relative error vs. computational time.



(a) Review. (b) StackOverflow. (c) Yahoo. (d) Wikipedia.

Figure 5: LAPE vs. computational time.

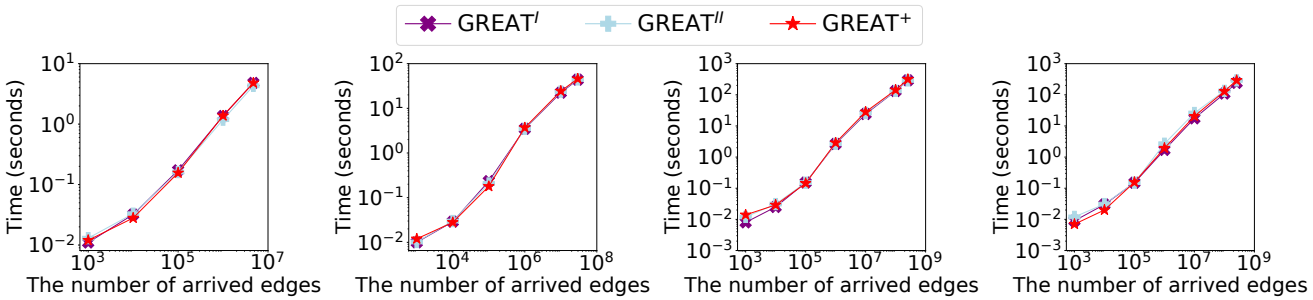
computed, thereby reducing the LAPE value. Except for FURL-0B, whose performance is primarily influenced by the default hash function, MASCOT samples fewer vertices across all datasets, resulting in the highest (worst) LAPE. On the other hand, WRS has the largest number of discovered triangle vertices across all datasets, leading to the lowest (best) LAPE. The algorithms GREAT^{I}, GREAT^{II}, GREAT^{+}, and TRIÈST-I exhibit similar LAPEs because they discover a similar number of triangle vertices. However, as the number of discovered vertices increases, the cost of maintaining the count of local triangles also increases, which naturally leads to a decrease in efficiency.

Efficiency. According to Figure 4, MASCOT is the fastest because it is a fixed-probability algorithm which saves the computational cost for removing edges. However, it has no bound on memory size, which is not applicable for streaming graphs. As mentioned earlier, when using the same amount of memory as the other algorithms, MASCOT produces inaccurate estimations. The efficiencies of algorithms GREAT^{I} and GREAT^{II} are nearly identical, both tied for the second place. The reason is provided in Section 3.3 that their reservoirs are often not full (storing fewer than k edges) most of the time, which reduces the computational cost of triangle counting. TRIÈST-I is slower than GREAT^{I} and GREAT^{II} because its reservoir is full all the time. GREAT^{+} is slightly slower than GREAT^{I}, GREAT^{II} and TRIÈST-I due to the cost of the adaptive strategy. WRS is the second slowest algorithm because it samples each arrived edge with probability 1 and updates the sample graph stored in the waiting room. In contrast, the other algorithms only update the sample graph in the reservoir when an edge is sampled successfully. FURL-0B is the slowest due to the HashHeap based reservoir, which takes $O(k \log k)$ maintenance cost for each sampled edge.

Summary. MASCOT is a fixed-probability algorithm which has no bound on memory size, so that it is not suitable for the streaming setting. Based on the evaluation results above, we rank all the other algorithms by their accuracy and efficiency in most cases, as shown in Table 3. The relative error of GREAT^{+} is significantly better (an order of magnitude less) than that of the competitors. GREAT^{II} and GREAT^{I} secure second and third place in terms of relative error, respectively. GREAT^{+}, GREAT^{I}, and GREAT^{II} sample a similar number of vertices, resulting in comparable LAPEs, which rank second best among all competitors. Algorithms GREAT^{I} and GREAT^{II} are the fastest across all datasets among FM-based algorithms. GREAT^{+} is slightly slower than GREAT^{I} and GREAT^{II}. When $\alpha = 0.1$, the performance of GREAT^{I} and GREAT^{II} are close in Figures 4 and 5. We will demonstrate in Figure 7 that GREAT^{I} is faster, while GREAT^{II} is more accurate across different values of α.

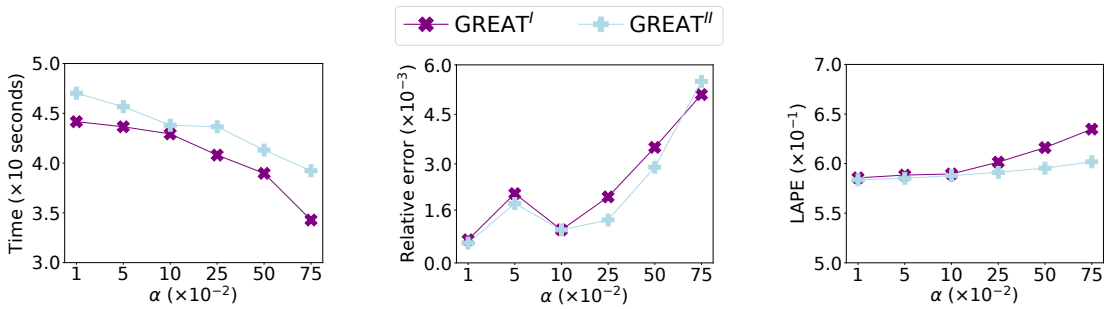
5.2.2 Effect of parameters. This section reports the performance of different algorithms under different parameter settings.

Scalability. Figure 6 shows the elapsed time of our algorithms when varying the number of arrived edges. Across all datasets, the elapsed time of our algorithms scales linearly with the number of arrived edges. The time spent for processing each edge is approximately 10^{-5} seconds, regardless of the total number of edges processed so far.



(a) Review. (b) StackOverflow. (c) Yahoo. (d) Wikipedia.

Figure 6: The elapsed time of our algorithms when varying the number of arrived edges.



(a) Time. (b) Relative error. (c) LAPE.

Figure 7: Varying α of GREAT^I and GREAT^{II} on StackOverflow.

Effect of α. Figure 7 shows the performance of algorithms GREAT^I and GREAT^{II} when varying α on dataset StackOverflow. The elapsed time decreases as α increases, which is consistent with the amortized time complexity analysis (Equations 6 and 7). Both the relative error and the LAPE become worse as α increases. This is because when α is large, the reservoir stores fewer edges so that fewer triangles are discovered, which is consistent with the analysis of sensitivity of α in Section 3.4.

We observe that GREAT^I is slightly more efficient but is slightly less accurate than GREAT^{II}, which has been proved in Sections 3.3 and 3.4.

Effect of z. The performance of GREAT$^+$ with different values of z on StackOverflow is presented in Figure 8. The larger the value of z, the smaller the value of $\alpha^{(r)}$, leading to more accurate but inefficient results. The main reason is that more edges are stored in the reservoir. We observe that as z increases, the elapsed time increases, and the relative error and the LAPE decrease, which is consistent with the analysis of sensitivity of z in Section 4. When using $z = 0.25$ with the StackOverflow dataset, GREAT$^+$ strikes a balance between accuracy and efficiency. When we choose a larger z, even though the LAPE becomes better slightly, the relative error cannot be improved and the elapsed time increases a lot. Therefore, we recommend avoiding large values of z.

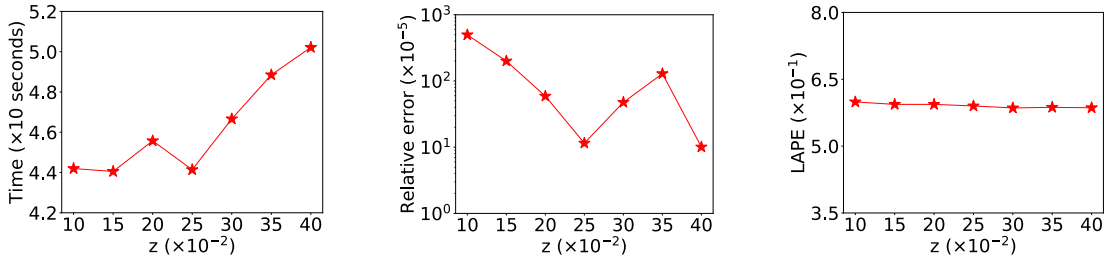
Effect of memory budget size. Figure 9 shows the performance of our algorithms and the competitors on dataset StackOverflow when varying the memory budget size. The elapsed time of all the algorithms increase as the memory budget size increases because more edges are maintained in the reservoir. As the memory budget size increases, we observe that (i) the relative error of MASCOT and GREAT^I fluctuates and all other algorithms have a decreasing trend, (ii) except FURL-0B, the LAPEs of all the other algorithms become better. In general, as the memory budget size increases, the accuracy of most algorithms is improved. This is because the larger the memory budget size is, more edges are stored in the reservoir and more triangles can be discovered. Note that GREAT^I, GREAT^{II}, and GREAT$^+$ have the smallest relative errors when using small memory budge size.

Table 3: The performance rankings of different algorithms.

Algorithm	Accuracy		Efficiency
	Relative error	LAPE	Elapsed time
GREAT$^+$	1	2	3
GREAT^I	3	2	1
GREAT^{II}	2	2	2
WRS	4	1	5
TRIÈST-I	5	2	4
FURL-0B	6	6	6
MASCOT	Unlimited memory size		

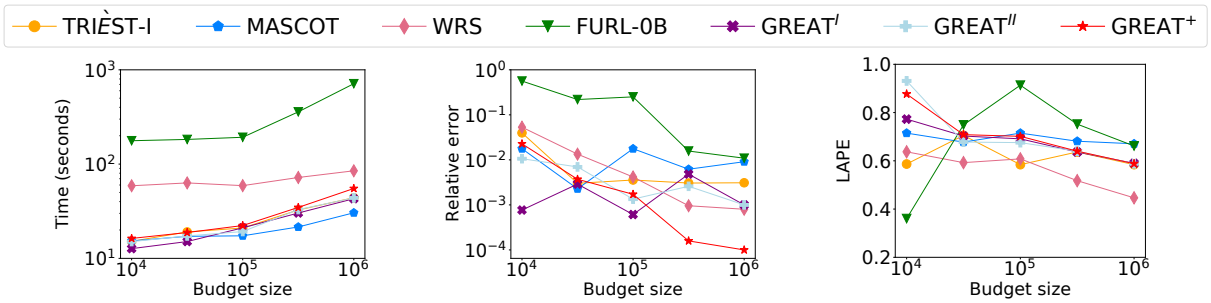
6 RELATED WORK

Most of existing algorithms for triangle counting estimation over streaming graphs are based on sampling techniques. In terms of the way of sampling the edge stream, existing triangle counting estimation algorithms can be classified into two categories [36]. One category of algorithms samples edges with fixed probability and uses unlimited memory budget size. The other category of algorithms uses a fixed-sized memory and samples edges with changing probability.



(a) Time. (b) Relative error. (c) LAPE.

Figure 8: Varying z of GREAT$^+$ on StackOverflow.



(a) Time. (b) Relative error. (c) LAPE.

Figure 9: Varying memory budget size on StackOverflow.

Fixed probability (FP) based estimation algorithms. Buriol et al. [5] sample a vertex v and an edge (a, b). If both the cross edges (a, v) and (b, v) arrive in the stream, triangle (v, a, b) is discovered. Pavan et al. [28] propose the neighborhood sampling that samples one edge together with one of its neighboring edges. If the third edge arrives in the stream, a triangle is discovered. Kavassery-Parakkat et al. [19] improve these two works using multi-sampling to achieve better theoretical bounds on the memory usage. They propose (i) the EVMS algorithm that samples multiple vertices and multiple edges and searches for corresponding cross edges in the stream and (ii) the NMS algorithm that samples multiple edges together with multiple neighboring edges and searches for the corresponding third edges in the stream. Framework gSH [1] samples an edge based on the number of previously selected edges that are adjacent to it and discovers triangles from sampled edges. MASCOT [26, 27] uses the DOULION algorithm [42] to sample each edge with a given fixed probability.

The FP based algorithms are not applicable to streaming graphs due to the following reasons. Some of the FP based algorithms require the size of the graph in order to determine the sampling probability. However, this information may be unavailable. The space complexity may be high since the memory size needed is related to the number of sampled edges.

Fixed-sized memory (FM) based estimation algorithms. This type of algorithms adopts reservoir sampling [20] where the reservoir occupies fixed-sized memory space and one random edge is removed to free space for future edges when the reservoir is full. Jha et al. [16, 17] are the pioneer to use reservoir sampling for the triangle counting estimation. These works maintain a reservoir for wedges (paths of length 2) and a reservoir for edges. The number of triangles is estimated based on the probability of the closed wedges. However, these works cannot estimate local triangle counts and do not guarantee unbiasedness. Moreover, these works suffer from high computation cost since two reservoir sampling processes are conducted. TRIÈST [38, 39] reduces computation cost by maintaining one reservoir for edges. One variant TRIÈST-B performs estimations after edge sampling and computes the probability of a discovered triangle as the probability that all three edges are sampled uniformly without replacement. The other variant TRIÈST-I performs estimations before edge sampling. Later, TS-triangle [53] estimates vertex degrees and triangle counts at the same time using the T-Sample [52] that is a dual sampling mechanism performing both uniform sampling and non-uniform sampling with a base reservoir and an incremental reservoir. The uniform sampling is used to count triangles by employing the TRIÈST-I. RFES and its variants [50] extend TRIÈST by storing the adjacent vertices of the two endpoints of each edge in the reservoir. It is able to discover the triangles when only one of the edges is sampled. However, the space complexity is high. WRS [23, 31] stores recent edges in the waiting room and keeps old edges in the reservoir. Each arrived edge is first sampled into the waiting room with probability 1. When the waiting room is full, the oldest edge in the waiting room are moved to the reservoir. FURL-0B [18] is based on TRIÈST-B that performs estimation after sampling. It uses a HashHeap to implement the reservoir. Recently, NHMS [36] extends NMS [19] by setting the memory limit. THS [36] extends TRIÈST by fixing the sampling probability. However, these two algorithms are biased.

The type of FM-based algorithms is superior to the type of FP-based algorithms in our problem setting because it requires no prior knowledge of the streaming graph and uses limited memory budget size to handle unbounded edge stream. Our algorithms

belong to the type of FM-based algorithms and outperform existing FM-based algorithms in terms of efficiency (see Tables 1 and 3) and accuracy (see Lemma 3.9 and Table 3). Specifically, existing FM-based algorithms are inaccurate because they ignore the dynamic timestamp intervals of triangles in real-world streaming graphs, resulting in sampled triangles that do not reflect the true timestamp interval distribution. Moreover, these algorithms are inefficient, as their reservoirs are always full. This leads to a computational cost for counting triangles that is proportional to the reservoir size k, which can be substantial.

Distributed triangle counting estimation. BulkUpdateAll [40] is a distributed version of the neighborhood sampling [28]. REPT [44] extends MASCOT to a distributed version where each arrived edge is broadcast to all the workers for triangle counting before sampling. Tri-Fly [32], CoCos [34], DVHT-b [51], DEHT-i [51], DEHT-b [49], DVHT-i [49], and TbEC [48] all extend TRIÈST [39] to distributed algorithms. They differ in the way of broadcasting edges to the workers. Tri-Fly [32] broadcasts each edge to all the workers. CoCos [34], DVHT-b [51], and DVHT-i [49] broadcast edges using the hash values of the vertices. DEHT-b [49] and DEHT-i [51] broadcast edges using the hash values of the edges. TbEC [48] is based on DiSLR and uses the binary trie to enable lossless compression and efficient transmission.

These distributed algorithms focus on how to assign the edges to the workers and handle communication issues. Our algorithms can be easily extended to distributed versions using these techniques. Due to the space limitations, we leave this direction as our future work.

Triangle counting estimation with deleted and duplicated edges. TRIÈST-FD [38] extends TRIÈST-B to deal with deleted edges using the Random Pairing [9]. PartitionCT [45] extends TRIÈST-B to handle duplicated edges. It considers the reservoir with k edges as k buckets and maps each arrived edge to different buckets using a hash function. MultiWMascot and MultiBMascot [26] extend MASCOT for handling deleted and duplicated edges in the stream. ThinkD$_{FAST}$ [33] extends MASCOT for handling deleted edges in the stream. ThinkD$_{ACC}$ [33] extends TRIÈST-I to handle deleted edges. To deal with duplicated edges, FURL-B [18] is developed based on TRIÈST-B and FURL-W [18] extends TRIÈST-I using weighted sampling. HyperSV [54] performs triangle counting on streaming hyperedge graphs based on TRIÈST-B. SWTC [13, 14] combines the Bound Priority Sampling [8] and PartitionCT to deal with the triangle estimation problem with the sliding window setting where both duplicated and deleted edges are included.

WRS [23] has been extended to handle deleted edges, while MASCOT [33] and TRIÈST-I [33] have been extended to handle both deleted and duplicated edges. Similarly, our algorithms can be easily extended to support deleted and duplicated edges. However, due to space constraints, we leave this extension for future work.

Reservoir based sampling algorithms for streaming data. The reservoir based sampling method is appropriate for streaming data because it maintains a bounded memory size, preventing memory leaks. Many researches on streaming data use traditional reservoir sampling, because it generate a uniform sample [20, 25, 43]. Besides traditional reservoir sampling that has been widely used in existing triangle counting estimation algorithms, there exist other reservoir based sampling algorithms.

Counting sampling algorithm [11] samples elements with a probability and maintains a counter C for each element in the reservoir. When the reservoir is full, the counter of each element in the reservoir is decreased by 1 with a probability. If the counter of an element equals 0, the element is removed from the reservoir. The counter in the counting sampling algorithm tracks the occurrence count of an element. Thus, the removing strategy of the counting sampling algorithm is different from our GRS. In addition, maintaining the counters increases the computational cost. Distinct sampling algorithm [12, 22] is a biased algorithm. Each arrived element is given a hash value. An element is added to the reservoir if its hash value is less than a threshold. When the reservoir is full, all the elements in the reservoir are removed with probability 0.5. The removing strategy of the distinct sampling algorithm is a special case of our GRS when the parameter α is set to 0.5. The performance of this special case is included in our experiment, shown in Figure 7. StreamSamp [7, 35] removes newly arrived sampled elements with probability 0.5 and keeps the old elements in the reservoir, so that it is unable to discover the triangles with short timestamp-intervals. The sliding window based algorithm SWTC [14] removes a batch of old elements, so that it may miss the triangles with long timestamp-intervals. Since our GRS randomly removes the edges in the reservoir with probability α, the sets of removed elements in both StreamSamp and SWTC represent one possible outcome of our GRS.

7 CONCLUSION

Triangle counting estimation is an important tool for various applications, e.g., network evolution analysis, community detection, anomaly detection, and subgraph detection. However, state-of-the-art algorithms, including FM-based algorithms and FP-based algorithms, for counting triangles over streaming graphs still suffer from substantial accuracy and efficiency issues. To overcome these two issues, we further develop the generalized reservoir sampling based triangle counting estimation (GREAT), which achieves lower amortized time complexity and smaller variance (cf. Sections 3.3 and 3.4). By exploring the property of dynamic timestamp interval in real-world streaming graphs, our advanced solution GREAT$^+$ further improves the estimation accuracy. Experiment results on four real streaming graph datasets verify that our algorithms significantly improve the accuracy of triangle counting estimations and achieve visible speedups.

In the future, we plan to extend our work for supporting streaming graphs with deleted and duplicated edges. Moreover, we will investigate how to extend our algorithms, GREAT and GREAT$^+$, on distributed systems for further improving the scalability of triangle counting estimation.

ACKNOWLEDGMENTS

This work is supported in part by the National Natural Science Foundation of China under Grants 62372308, 23IAA00610, and 62202401 and the GuangDong Basic and Applied Basic Research Foundation under Grant 2023A1515011619.

REFERENCES

[1] Nesreen K. Ahmed, Nick G. Duffield, Jennifer Neville, and Ramana Rao Kompella. 2014. Graph sample and hold: a framework for big-graph analytics. In *KDD*. 1446–1455.

[2] Nesreen K. Ahmed, Jennifer Neville, and Ramana Rao Kompella. 2013. Network Sampling: From Static to Streaming Graphs. *TKDD* 8, 2 (2013), 7:1–7:56.

[3] David A. Bader, Fuhuan Li, Anya Ganeshan, Ahmet Gündogdu, Jason Lew, Oliver Alvarado Rodriguez, and Zhihui Du. 2023. Triangle Counting Through Cover-Edges. In *HPEC*. 1–7.

[4] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *WWW*. 595–602.

[5] Luciana S. Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. 2006. Counting triangles in data streams. In *PODS*. 253–262.

[6] Don Coppersmith and Shmuel Winograd. 1987. Matrix Multiplication via Arithmetic Progressions. In *STOC*. 1–6.

[7] Baptiste Csernel, Fabrice Clerot, and Georges Hébrail. 2006. Datastream clustering over tilted windows through sampling. *Knowledge discovery from data streams* (2006), 127.

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

[9] Rainer Gemulla, Wolfgang Lehner, and Peter J. Haas. 2008. Maintaining bounded-size sample synopses of evolving datasets. *VLDB* 17, 2 (2008), 173–202.

[10] Sayan Ghosh and Mahantesh Halappanavar. 2020. TriC: Distributed-memory Triangle Counting by Exploiting the Graph Structure. In *HPEC*. 1–6.

[11] Phillip B. Gibbons and Yossi Matias. 1998. New Sampling-Based Summary Statistics for Improving Approximate Query Answers. In *SIGMOD*. 331–342.

[12] Phillip B. Gibbons and Srikanta Tirthapura. 2001. Estimating simple functions on the union of data streams. In *SPAA*. 281–291.

[13] Xiangyang Gou and Lei Zou. 2021. Sliding Window-based Approximate Triangle Counting over Streaming Graphs with Duplicate Edges. In *SIGMOD*. 645–657.

[14] Xiangyang Gou and Lei Zou. 2023. Sliding window-based approximate triangle counting with bounded memory usage. *VLDB J.* 32, 5 (2023), 1087–1110.

[15] Shenyang Huang, Farimah Poursafaei, Jacob Danovitch, Matthias Fey, Weihua Hu, Emanuele Rossi, Jure Leskovec, Michael M. Bronstein, Guillaume Rabusseau, and Reihaneh Rabbany. 2023. Temporal Graph Benchmark for Machine Learning on Temporal Graphs. In *NIPS*.

[16] Madhav Jha, C. Seshadhri, and Ali Pinar. 2013. A space efficient streaming algorithm for triangle counting using the birthday paradox. In *SIGKDD*. 589–597.

[17] Madhav Jha, C. Seshadhri, and Ali Pinar. 2015. A Space-Efficient Streaming Algorithm for Estimating Transitivity and Triangle Counts Using the Birthday Paradox. *TKDD* 9, 3 (2015), 15:1–15:21.

[18] Minsoo Jung, Yongsub Lim, Sunmin Lee, and U Kang. 2019. FURL: Fixed-memory and uncertainty reducing local triangle counting for multigraph streams. *Data Min. Knowl. Discov.* 33, 5 (2019), 1225–1253.

[19] Neeraj Kavassery-Parakkat, Kiana Mousavi Hanjani, and A. Pavan. 2018. Improved Triangle Counting in Graph Streams: Power of Multi-Sampling. In *ASONAM*. 33–40.

[20] Donald E Knuth. 1973. The art of computer programming, vol. 2:Seminumerical Algorithms. *Reading MA: Addison-Wisley* (1973), 144.

[21] Jérôme Kunegis. 2013. KONECT: the Koblenz network collection. In *WWW*. 1343–1350.

[22] Bibudh Lahiri and Srikanta Tirthapura. 2009. *Stream Sampling*. Springer US, 2838–2842.

[23] Dongjin Lee, Kijung Shin, and Christos Faloutsos. 2020. Temporal locality-aware sampling for accurate triangle counting in real graph streams. *VLDB J.* 29, 6 (2020), 1501–1525.

[24] Jure Leskovec and Rok Sosic. 2016. SNAP: A General-Purpose Network Analysis and Graph-Mining Library. *ACM Trans. Intell. Syst. Technol.* 8, 1 (2016), 1:1–1:20.

[25] Kim-Hung Li. 1994. Reservoir-sampling algorithms of time complexity o (n (1+ log (n/n))). *TOMS* 20, 4 (1994), 481–493.

[26] Yongsub Lim, Minsoo Jung, and U Kang. 2018. Memory-Efficient and Accurate Sampling for Counting Local Triangles in Graph Streams: From Simple to Multigraphs. *TKDD* 12, 1 (2018), 4:1–4:28.

[27] Yongsub Lim and U Kang. 2015. MASCOT: Memory-efficient and Accurate Sampling for Counting Local Triangles in Graph Streams. In *SIGKDD*. 685–694.

[28] A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun-Lung Wu. 2013. Counting and Sampling Triangles from a Graph Stream. *PVLDB* 6, 14 (2013), 1870–1881.

[29] Marco Sánchez-Aguayo, Luis Urquiza-Aguiar, and José Estrada-Jiménez. 2021. Fraud Detection Using the Fraud Triangle Theory and Data Mining Techniques: A Literature Review. *Comput.* 10, 10 (2021), 121.

[30] Seyed-Vahid Sanei-Mehri, Yu Zhang, Ahmet Erdem Sariyüce, and Srikanta Tirthapura. 2019. FLEET: Butterfly Estimation from a Bipartite Graph Stream. In *CIKM*. 1201–1210.

[31] Kijung Shin. 2017. WRS: Waiting Room Sampling for Accurate Triangle Counting in Real Graph Streams. In *ICDM*. 1087–1092.

[32] Kijung Shin, Mohammad Hammoud, Euiwoong Lee, Jinoh Oh, and Christos Faloutsos. 2018. Tri-Fly: Distributed Estimation of Global and Local Triangle Counts in Graph Streams. In *PAKDD*. 651–663.

[33] Kijung Shin, Jisu Kim, Bryan Hooi, and Christos Faloutsos. 2018. Think Before You Discard: Accurate Triangle Counting in Graph Streams with Deletions. In *PKDD*. 141–157.

[34] Kijung Shin, Euiwoong Lee, Jinoh Oh, Mohammad Hammoud, and Christos Faloutsos. 2021. CoCoS: Fast and Accurate Distributed Triangle Counting in Graph Streams. *TKDD* 15, 3 (2021), 38:1–38:30.

[35] Rayane El Sibai, Yousra Chabchoub, Jacques Demerjian, Zakia Kazi-Aoul, and Kablan Barbar. [n.d.]. Sampling algorithms in data stream environments. In *ICDEc*. 29–36.

[36] Paramvir Singh, Venkatesh Srinivasan, and Alex Thomo. 2021. Fast and Scalable Triangle Counting in Graph Streams: The Hybrid Approach. In *AINA*. 107–119.

[37] Stavros Souravlas, Angelo Sifaleras, M. Tsintogianni, and Stefanos Katsavounis. 2021. A classification of community detection methods in social networks: a survey. *Int. J. Gen. Syst.* 50, 1 (2021), 63–91.

[38] Lorenzo De Stefani, Alessandro Epasto, Matteo Riondato, and Eli Upfal. 2016. TRIÈST: Counting Local and Global Triangles in Fully-Dynamic Streams with Fixed Memory Size. In *SIGKDD*. 825–834.

[39] Lorenzo De Stefani, Alessandro Epasto, Matteo Riondato, and Eli Upfal. 2017. TRIÈST: Counting Local and Global Triangles in Fully Dynamic Streams with Fixed Memory Size. *TKDD* 11, 4 (2017), 43:1–43:50.

[40] Kanat Tangwongsan, A. Pavan, and Srikanta Tirthapura. 2013. Parallel triangle counting in massive streaming graphs. In *CIKM*. 781–786.

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

[42] Charalampos E. Tsourakakis, U Kang, Gary L. Miller, and Christos Faloutsos. 2009. DOULION: counting triangles in massive graphs with a coin. In *KDD*. 837–846.

[43] Jeffrey S Vitter. 1985. Random sampling with a reservoir. *TOMS* 11, 1 (1985), 37–57.

[44] Pinghui Wang, Peng Jia, Yiyan Qi, Yu Sun, Jing Tao, and Xiaohong Guan. 2019. REPT: A Streaming Algorithm of Approximating Global and Local Triangle Counts in Parallel. In *ICDE*. 758–769.

[45] Pinghui Wang, Yiyan Qi, Yu Sun, Xiangliang Zhang, Jing Tao, and Xiaohong Guan. 2017. Approximately Counting Triangles in Large Graph Streams Including Edge Duplicates with a Fixed Memory Usage. *PVLDB* 11, 2 (2017), 162–175.

[46] Stanley Wasserman and Katherine Faust. 1994. *Social Network Analysis: Methods and Applications*. Cambridge University Press.

[47] Siyue Wu, Dingming Wu, Sinhong Cheuk, Tsz Nam Chan, and Kezhong Lu. 2025. Supplementary material for "GREAT: Generalized Reservoir Sampling based Triangle Counting Estimation over Streaming Graphs". https://github.com/sinhong-cheuk/GREAT-Generalized-Reservoir-Sampling-based-Triangle-Counting-Estimation-over-Streaming-Graphs.

[48] Xu Yang, Chao Song, Jiqing Gu, Ke Li, and Hongwei Li. 2023. A distributed streaming framework for edge-cloud triangle counting in graph streams. *KBS* 278 (2023), 110878.

[49] Xu Yang, Chao Song, Mengdi Yu, Jiqing Gu, and Ming Liu. 2022. Distributed Triangle Approximately Counting Algorithms in Simple Graph Stream. *TKDD* 16, 4 (2022), 79:1–79:43.

[50] Changyong Yu, Huimin Liu, Fazal Wahab, Zihan Ling, Tianmei Ren, Haitao Ma, and Yuhai Zhao. 2023. Global triangle estimation based on first edge sampling in large graph streams. *TJS* 79, 13 (2023), 14079–14116.

[51] Mengdi Yu, Chao Song, Jiqing Gu, and Ming Liu. 2019. Distributed Triangle Counting Algorithms in Simple Graph Stream. In *ICPADS*. 294–301.

[52] Lingling Zhang, Hong Jiang, Fang Wang, Dan Feng, and Yanwen Xie. 2019. T-Sample: A Dual Reservoir-Based Sampling Method for Characterizing Large Graph Streams. In *ICDE*. 1674–1677.

[53] Lingling Zhang, Hong Jiang, Fang Wang, Dan Feng, and Yanwen Xie. 2020. Reservoir-based sampling over large graph streams to estimate triangle counts and node degrees. *FGCS* 108 (2020), 244–255.

[54] Lingling Zhang, Zhiwei Zhang, Guoren Wang, Ye Yuan, and Zhao Kang. 2023. Efficiently Counting Triangles for Hypergraph Streams by Reservoir-Based Sampling. *TKDE* 35, 11 (2023), 11328–11341.