The VLDB Journal
https://doi.org/10.1007/s00778-023-00783-3

REGULAR PAPER

Sliding window-based approximate triangle counting with bounded memory usage

Xiangyang Gou[1] · **Lei Zou**[1,2]

Received: 6 January 2022 / Revised: 30 November 2022 / Accepted: 20 January 2023

Abstract
Streaming graph analysis is gaining importance in various fields due to the natural dynamicity in many real graph applications. However, approximately counting triangles in real-world streaming graphs with duplicate edges and sliding window model remains an unsolved problem. In this paper, we propose *SWTC* algorithm to address approximate *s*liding-*w*indow *t*riangle *c*ounting problem in streaming graphs. In SWTC, we propose a fixed-length slicing strategy that addresses both sample maintaining and cardinality estimation issues with a bounded memory usage. We theoretically prove the superiority of our method in sample graph size and estimation accuracy under given memory upper bound. To further improve the performance of our algorithm, we propose two optimization techniques, vision counting to avoid computation peaks, and asynchronous grouping to stabilize the accuracy. Extensive experiments also confirm that our approach has higher accuracy compared with the baseline method under the same memory usage.

Keywords Streaming graphs · Approximate algorithms · Triangle counting

1 Introduction

Graphs are an omnipresent form representing large-scale entities and their relations in various fields, like biochemistry, social networks and knowledge graphs. Various kinds of data analysis can be implemented upon a graph. Among them triangle counting is one of the most fundamental queries. Many applications are based on triangle counting, like community detection [1], topic mining [2], spam detection [3] and so on [4–7].

In the era of big data, new challenges arise in graph analysis. Graphs not only grow in scale, but also become more dynamic. In some applications, data are organized as *streaming graphs*. A streaming graph is an unbounded sequence of items that arrive at a high speed, and each item indicates an edge between two nodes. Together these items form a large dynamic graph. The large scale and high dynamicity make it both memory and time consuming to store and analyze streaming graphs accurately. It is a natural choice to resort to efficiently compute approximations. A popular method is to conduct graph analysis tasks over a small-size sample graph. In this work, we focus on approximately counting triangles in large streaming graphs using sampling techniques.

Although several algorithms have been proposed in the literature, most of them do not support edge expiration [8–10]. In other words, after an edge is received in the streaming graph, it is permanently effective. However, in real world applications, we usually need to delete old data in the need of timeliness. Here we provide a motivation example. In social networks, user communications form a streaming graph. Spam and topics in social networks can be detected with *triangle counting* [3,11]. New spam and topics need to be detected as soon as possible, in order to control the damage brought by potential fraudsters, or catch up with hot spots of public opinion in real time. Therefore, we need to *continuously* monitor the triangle count within a recent period, such as the last 24 hours. Elder edges are considered of little value, as the topics or spam formed by them are out-of-date. These most recent edges are always changing, which are defined as a *sliding window* [12]. The sliding window model

This work was supported by NSFC under Grant 61932001 and U20A20174.

✉ Lei Zou
zoulei@pku.edu.cn

Xiangyang Gou
gxy1995@pku.edu.cn

[1] Peking University, Beijing, China

[2] Beijing Academy of Artificial Intelligence, Beijing, China

Published online: 09 March 2023

is widely used in streaming graph algorithms and systems [13–15]. Therefore, a sliding window-based continuous triangle counting algorithm is desired.

Besides, there are usually duplicate edges in the streaming graph, namely the same edge may appear multiple times. In the above example, each pair of users may communicate multiple times, and thus the raw communication logs have duplicate edges. Generally speaking, there are two different semantics dealing with duplicate edges, *binary counting* and *weighted counting* (see Definitions 4). Binary counting [10,16] only considers the existence of edges and filters out duplication, while weighted counting [16–18] takes duplicate edges into account. Our proposed method can also support duplicate edges, and apply to both weighted counting and binary counting.

We are also strict with time and memory consumption. In practice, we need to continuously monitor the triangle count and issue an alert when it reaches a certain threshold. Therefore, a low-latency continuous counting is desired. Besides, memory consumed by such monitor algorithm needs to be preserved. If the memory usage of an algorithm rises with the increasing stream throughput, it may exceed the preserved memory and introduce errors at peak time. The risk of such memory constraint violation is high in real-world streaming graphs, as the throughput at peak time may be multiple times higher than ordinary days and hard to predict. Therefore, we hope the memory usage of the algorithm can be bounded by a pre-defined constant. According to analysis in [19], a fixed-size-k sample needs $\Omega(log(n)k)$ memory, where n is the number of distinct edges in the sliding window. Such memory usage varies with the throughput of the streaming graph, and can hardly be pre-bounded. Therefore, we resort to algorithm with bounded-size sample. To the best of our knowledge, no existing work considers such problem. There are several following challenges:

1. Old edges will expire in the sliding window model, which changes both the original graph and the sample graph, and makes the sample biased.
2. An edge may appear multiple times. We need to filter out duplicate edges in binary counting.
3. When scaling up the triangle count in the bounded-size sample graph to the original graph, it is hard to estimate the number of edges in the sliding window, because we cannot notice expiration of unsampled edges.

1.1 Our solution

In order to approximately count triangles in sliding window-based streaming graphs with bounded memory, there are two major steps. First, we maintain a sample graph with bounded memory and estimate the number of edges in the sliding window continuously. Then, we count the triangles in the sample graph and scale up the count according to the estimated edge number.

Maintaining sample with bounded memory in sliding windows is challenging. Sampling techniques used in prior triangle counting algorithms fail to meet the demand, even if some of them are proposed for fully dynamic streaming graphs (details will be discussed in Sect. 3). The theoretical bound [19] about the space complexity rules out the chance to maintain a *fixed*-size sample in the sliding windows with bounded memory. We have to compromise to a *bounded*-size sample and struggle to maximize the expected sample size.

In order to address this issue, we begin with a baseline that combines the structure of PartitionCT [10] with BPS algorithm [19]. However, the expected sample graph size in this baseline is rather small compared to its memory usage. Therefore, we further propose an optimized sampling technique, *fixed-length slicing strategy*. It splits a streaming graph into multiple fixed-length slices, and performs priority sampling based on these slices. A carefully designed sampling algorithm produces a larger sample graph under the same memory usage compared with the baseline, which is theoretically proven in Sect. 4.2. Also, mathematical analysis in Sect. 4.4 and extensive experiments in Sect. 7 confirm that our larger sample graphs can decrease the mean absolute percentage error (MAPE) of triangle count estimation by 38% (Fig. 10a) and max error by 37% (Fig. 10c) compared with the baseline.

We also need to continuously monitor the number of edges in the sliding window.[1] It is a necessary parameter when scaling up the triangle count in the sample to get an approximation in the sliding window. Although there are classical streaming data cardinality estimation algorithms like [20,21], they cannot support edge expiration in sliding windows. Fortunately, the *fixed-length slicing strategy* proposed in Sect. 4.1 can address both sample maintaining and cardinality estimation together. Based on it, we propose a continuous cardinality estimation algorithm in Sect. 4.3.

Although we address both sample maintaining and cardinality estimation using a uniform strategy—*the fixed-length slicing*, there are still two problems: periodic computation peak and accuracy trough. In order to solve these problems, we propose two techniques: vision counting (Sect. 5.1) which smooths the computation cost and asynchronous grouping (Sect. 5.2) which stabilizes the valid sample size and accuracy.

Table 1 positions our method with regard to state-of-the-art approximate triangle counting work over dynamic graphs, and more discussions are given in Sect. 8. Generally, our method is the only work that addresses both edge duplication

[1] Depending on the semantics of binary counting or weighted counting, we need to either count distinct number of edges, or include the duplicate edges in counting.

Table 1 Comparison with Existing Work in Approximate Triangle Counting over Streaming/Dynamic Graphs

Algorithm	Dynamic graph model	Allowing Edge Duplication	Binary or Weighted
A.Pavan *et.al.* [8]	Insertion only	✗	✗
WRS [22]	Fully dynamic	✓	Weighted
PartitionCT [10]	Insertion only	✓	Binary
SWTC	Sliding window	✓	Both

and expiration. More importantly, our method can support both *binary counting* and *weighted counting* semantics. In summary, we made the following contributions.

1. In order to solve the problem of approximately counting triangles in streaming graphs with sliding windows, we propose a fixed-length slicing strategy that addresses both sample maintaining and cardinality estimation. It applies to both binary counting and weighted counting when dealing with edge duplication. We theoretically prove the superiority of our method in the sample graph size and estimation accuracy under given memory upper bound.
2. To further improve the performance of our algorithm, we propose two optimization techniques, vision counting to avoid computation peaks, and asynchronous grouping to stabilize the valid sample size.
3. Extensive experiments confirm that our method outperforms the baseline solution in terms of sample size and estimation accuracy. We released all codes at Github [23].

2 Problem definition

Definition 1 Streaming Graph: A streaming graph is an unbounded time evolving sequence of items $\mathcal{S} = \{e_1, e_2, e_3 \ldots\ldots e_n\}$, where each item $e_i = (\langle u, v\rangle, t(e_i))$ indicates an edge between nodes u and v arriving at time $t(e_i)$. This sequence continuously arrives from data sources like routers or monitors with high speed. An edge $\langle u, v\rangle$ may appear multiple times with different timestamps. These multiple occurences are called duplicate edges.

A streaming graph can be either directed or undirected. In the problem of triangle counting, as most prior work defines triangles without considering edge directions, we also ignore edge directions. Our algorithm also applies to directed graphs, and we will discuss it in Sect. 6. Note that in the streaming graph model, due to the high speed and large volume of the stream, we assume that it is not physically stored and has to be processed in one-scan manner in real time. In other words, each edge in the stream can only be processed once upon its arrival. Besides, it should be noted that the throughput of the streaming graph keeps varying. There may be multiple (or no) edges arriving at each time point.

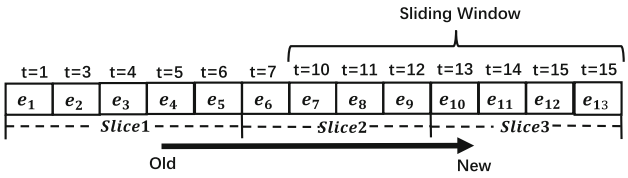


Fig. 1 Streaming Graph and Sliding Window at $T = 15$

In real world applications, we are only interested in the most recent edges, which are modeled as the *sliding window*. There are two kinds of sliding windows: *count-based sliding windows* (also called sequence-based sliding windows) and *time-based sliding windows*. In this paper, we focus on time-based sliding windows. Count-based one can be seen as a simplified time-based sliding window where there is exactly one edge coming at each time point. Most previous algorithms and applications also use time-based sliding windows [13–15]. For simplicity, we use "sliding window" to denote the time-based sliding window in the following sections.

Definition 2 Sliding window: A sliding window with window length N in a streaming graph $\mathcal{S}$ is a set of edges e_i with timestamps within $(T - N, T]$, where T is the current time, namely clock time of the system. We denote this window with W.

The window length N depends on applications, and the number of edges in the sliding window varies with the throughput of the stream. More generally, we use $W_{t_1}^{t_2}$ to represent a set of edges with timestamps between t_1 and t_2. Based on the definition of the sliding window, we introduce the snapshot graph.

Definition 3 Snapshot graph: A *snapshot* graph at time T, denoted as $\mathcal{G}$, is a graph induced by all the edges within the sliding window W.

Example 1 A streaming graph $\mathcal{S}$ with the sliding window model is given in Fig. 1. The window length is $N = 6$ and current time is $T = 15$. The timestamp of each edge is shown above it. Current sliding window is W_9^{15}, and there are 7 edges in it. The separation of slices is used in SWTC algorithm, and will be explained in Sect. 4.

In this paper, we focus on *continuous triangle counting* in the sliding window model, which monitors triangle count in

the current snapshot graph. There are two semantics for triangle counting in a graph, global counting, namely counting all triangles in the graph, and local counting, which means counting triangles including a specific node u. Our method applies to both semantics. We focus on global counting in the majority of the paper, and discuss local counting in Sect. 6. For simplicity of presentation, we use "triangle counting" to denote global counting unless specified.

When there are duplicate edges in the sliding window, there are two kinds of semantics to deal with these duplicate edges [16], i.e., *binary counting* and *weighted counting*:

Definition 4 Binary & Weighted counting: A triangle in a graph G is defined as a tuple of three edges $(\langle u, v\rangle, \langle u, w\rangle, \langle v, w\rangle)$ (u, v, w are distinct nodes), where any two edges share one common node. In binary counting, we return the total number of *distinct* triangles in graph G. In weighted counting, the *weight* of triangle $(\langle u, v\rangle, \langle u, w\rangle, \langle v, w\rangle)$ is $f(\langle u, v\rangle) \times f(\langle u, w\rangle) \times f(\langle v, w\rangle)$, where $f(\cdot)$ denotes number of occurrences of an edge, namely the frequency. Weighted counting returns the *sum* of all triangle weight.

In binary counting, we need to filter out duplication and only concentrate on distinct edges. On the other hand, in weighted counting, as a weighted triangle can be seen as multiple triangles induced by duplicate edges, duplicate edges also contribute to the triangle count. We can include them in sampling and edge count estimation. For simplicity of presentation, we focus on binary counting in the majority of our paper, and discuss how to extend our algorithm to weighted counting in Sect. 6. The denotations are presented in Table 2.

Table 2 Notation Table

Notation	Meaning
$\mathcal{S}$	Streaming graph
$t(e)$	Timestamp of an edge e
W	Sliding window
N	Length of the sliding window
$\mathcal{G}$	Snapshot graph
$\mathcal{G}_s$	Sample graph
$W_{t_1}^{t_2}$	Set of edges e where $t_1 < t(e) \leq t_2$
$\lvert W_{t_1}^{t_2}\rvert$	Number of distinct edges in slice $W_{t_1}^{t_2}$
ϵ	sample edge in BPS algorithm
ϵ_{test}	Test edge in BPS algorithm
$H(\cdot)$	Function that maps edges to substreams
$G(\cdot)$	Function that produces edge priorities
ℓ	The latest landmark before current time T
J	The second latest landmark before current time T
$\beta[i]$	Edge with the largest priority in the i_{th} substream in W_J^ℓ
$\epsilon[i]$	Edge with the largest priority in the i_{th} substream in W_ℓ^T

3 Baseline

As mentioned above, in order to estimate the number of triangles, the first challenge is to retain a *uniform* (i.e., unbiased) sample in the sliding window.

A naive solution is to use fixed probability sampling. We randomly generate a priority in range $0 \sim 1$ for each edge in the streaming graph. If the probability is smaller than a preset threshold p, we sample the edge. p is called sampling probability. Whenever an edge in the sample set expires, we discard it. Although this sampling strategy is simple and uniform, it has the drawback of unbounded memory usage. The size of the sample set is $p|W|$, and thus the memory usage is $O(|W|)$. Though the length of the sliding window N is fixed, the throughput of the streaming graph varies with time. The edge number $|W|$ is unpredictable. At peak time of throughput, the memory usage may exceed the preserved memory and result into system errors. Even if we can foresee the peak throughput and preserve enough memory, these memory is not fully used in ordinary time, resulting into low accuracy compared to the preserved memory size. Our experimental results in Sect. 7.9 confirm this.

It should be noted that algorithms for fully dynamic models with bounded memory usage such as [17,18] cannot be used in sliding windows. They need to know whenever an edge is deleted, no matter the deleted edge is sampled or not. In the sliding window model, edges expire automatically as the window slides. Unless we store all edges together with their timestamps, we cannot know when unsampled edges expire. Therefore, algorithms like [17,18] can only work by storing the entire sliding window, which consumes a large amount of memory. Therefore, we need to design a new sampling scheme to maintain uniform sample in the sliding window model.

Before presenting our method, we first introduce some background knowledge about the priority sampling [24] and BPS (bounded priority sampling) algorithm [19] (in Sect. 3.1), which benefits the understanding our baseline. Since BPS does not consider duplication, we will discuss how to revise it to deal with duplication and combine it with the structure of PartitionCT [10] to improve time and memory efficiency in our baseline solution (in Sect. 3.2).

3.1 Background

BPS algorithm [19] is designed for sampling in sliding windows without duplication. Theoretically, authors in [19] prove that it is impossible to maintain a *fixed-size* uniform sample in sliding windows with memory bounded by constant[2]. As a compromise, BPS maintains a bounded-size sample, which lays the foundation of our solution. For the

[2] In page 3, Sect. 3.1 of [19].

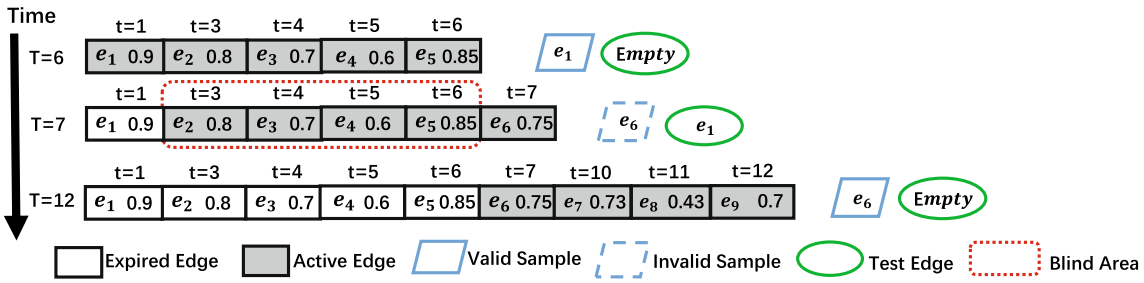


Fig. 2 Example of BPS Sampling

simplicity of the presentation, we first consider how to maintain a sample set with bounded size 1.

Generally speaking, BPS algorithm is based on priority sampling [24]. Whenever a new edge e comes in the stream, BPS generates a random priority $G(e)$. BPS algorithm selects the edge with the largest priority as the sample. Because the priority is randomly generated, each edge has an equal probability to get the largest priority. Thus the sampling is uniform.

If there are only insertions in the stream (without edge expiration in the sliding window), we can maintain the sample with the largest priority by comparing the sampled one, denoted as ϵ, with the new coming edge. When the new edge has a larger priority, we replace ϵ with it. For example, in Fig. 2, assume that the window length is 6. The sample edge from time 1 to 6 is edge e_1 that arrives at $t = 1$. Note that all unsampled edges are not stored.

However, when a sample edge expires, it is more complicated to select the successor sample. Some edges after the sample edge ϵ may be *shaded* by ϵ, since they have smaller priorities. They are discarded after compared with ϵ and we cannot retrieve them. These discarded edges are called *blind area*. After the expiration of ϵ, the edges in the blind area are still alive but we do not store them. In this case, we cannot determine the edge with the largest priority, because we do not know the priorities of edges in the blind area. For example, in Fig. 2, e_1 expires at time $t = 7$ and the four edges arriving from time 3 to 6 form a blind area. When a new edge e_6 comes at time $t = 7$, we cannot select e_6 as the sample edge, as we are not sure about the priorities of edges in the blind area. Otherwise, setting e_6 as the sample will violate the principle of priority sampling and introduce bias.

To address the above problem, BPS algorithm proposes the following solution. When a sample edge expires, we store it using another variable, a test edge ϵ_{test}, which serves as an upper bound of edge priorities in the blind area. On the other hand, ϵ is set to the next coming edge, and then maintained by keeping comparing new edges with it. ϵ is a *valid* sample only when $G(\epsilon) \geq G(\epsilon_{test})$. In this case, since priorities of edges in the blind area are smaller than $G(\epsilon_{test})$, they are also smaller than $G(\epsilon)$. Otherwise ϵ is invalid. For example, at time 7 in Fig. 2, we set $\epsilon = e_6$, but it is invalid.

ϵ_{test} will *double expire* when its timestamp is smaller than $T - 2N$, where T is the current time and N is the window length. The length of the blind area following ϵ_{test} is at most N. It means all edges in the blind area must expire when ϵ_{test} double expires. By then we can set the current sample edge as a *valid* one, since all edges in the sliding window have participated in the competition with the current sample edge. The winner has the largest priority. Thus it is a valid sample. In the example in Fig. 2, e_1 double expires at $t = 12$. At this time, edges arriving from time 2 to 6 all expire. The current sample is e_6. Since other edges in the sliding window have all been compared with it, we can set e_6 as a valid sample.

According to the above discussion, with bounded sample size 1, BPS algorithm cannot get a valid sample in some periods. We call such period a *vacuum* period. In Fig. 2, the vacuum period is from $t = 7$ to $t = 12$.

When extended to bounded sample size k ($k \geq 1$), the original BPS algorithm maintains two set S_0 and S_1, both with size k. Each new edge is added to S_0 if the size of S_0 is smaller than k, or if the new edge has larger priority than the smallest priority in S_0. Once an edge in S_0 expires, it is added to S_1. And if an edge in S_1 double expires, it is deleted. The valid sample set is computed as $top\text{-}k(S_0 \cup S_1) \cap S_0$ ($top\text{-}k(\cdot)$ means edges with top-k largest priority in the set). In order to continuously monitor the valid sample set, we need to use a data structure called Treap [25] to maintain these sets, so that we can search for edges according to IDs and sort them in priority at the same time.

3.2 The baseline method

The original BPS algorithm uses a random function $G(\cdot)$ to generate priority. When there is edge duplication, we use a *hash* function to define the priority $G(\cdot)$ instead of a random one. With a random function, duplicate edges will get multiple priorities and a higher sampling probability, which leads to bias. The hash function generates the same priority for a duplicate edge, no matter how many times the edge arrives. Therefore, it can derive a uniform sample.

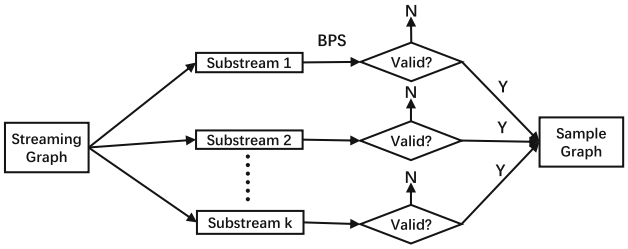


Fig. 3 Framework of the Baseline Method

The original Treap-based BPS needs $O(log(k))$ time cost to maintain the Treaps upon each edge arrival. And the tree-based structure of Treap is also memory consuming. In our baseline method, we use the technique in PartitionCT [10] to improve the method. Assume that the sample size is upper bounded by k, where k is an user-specified parameter. We use a hash function $H(\cdot)$ to split a streaming graph into k substreams, like PartitionCT. In each substream, we use BPS algorithm to obtain at most one valid sample edge. In this case, we only need $O(1)$ time to maintain the sample set upon each edge arrival. Besides, we only need a table of k bins to maintain the sample set, each bin containing a sample edge and a test edge, corresponding to a substream. Such table-based structure is much more space efficient than tree-based structure. The framework of the baseline method is shown in Fig. 3. Notice that only valid sample edges are included in the sample graph $\mathcal{G}_s$ and contribute to the triangle counter.

Let k be a user-specified parameter in the baseline method. In the best case, each substream has a valid sample edge and the number of edges in the sample graph $\mathcal{G}_s$ is k. Therefore, the memory upper bound in the baseline is able to hold a k-edges sample graph $\mathcal{G}_s$. However, as each substream has an independent probability to be in the vacuum period of BPS sampling (i.e., cannot provide a valid sample), the sample graph is smaller than k. We theoretically prove that the expected sample size of the baseline method is in range $\left[k \times \frac{|W|}{|W^T_{T-2N}|}, k \times \left(1 - \frac{|W^{T-N}_{T-2N}|}{|W^T_{T-3N}|}\right)\right]$ (see Theorem 2 in Sect. 4.2), where $|\cdot|$ denotes the number of distinct edges in the window. Assume that the streaming graph's throughput is steady, the expected sample size is $[0.5k, 0.66k]$.

We can improve the sampling strategy to get a larger sample graph. In next section, we will propose a new sampling strategy to get a larger sample graph with the same memory upper bound, and obtain a higher accuracy in triangle count estimation.

4 Our method

In this section, we propose our algorithm (called *SWTC*) to address approximate *s*liding-*w*indow *t*riangle *c*ounting problem in streaming graphs with edge duplication. First, we propose a fixed-length slicing based sampling strategy in Sect. 4.1. In Sect. 4.2, we theoretically prove that SWTC gets a larger-size sample graph than the baseline method under the same memory consumption. In Sect. 4.3, we discuss how to continuously monitor $|W|$, namely the number of distinct edges in the sliding window, and estimate the triangle count in the sliding window. In Sect. 4.4, we theoretically analyze the accuracy of SWTC. In Sect. 4.5, we further analyze the time cost of SWTC and compare it with the baseline method.

4.1 SWTC sampling strategy

Sampling Strategy: We propose a fixed-length slicing method in SWTC. Specifically, we split the timeline of the streaming graph into multiple slices with fixed-length of N time units, and each splitting point is called a *"landmark"*. It is easy to know that, the current sliding window W overlaps with at most two slices, which are denoted as $W^{\ell}_{\jmath}$ and W^{T}_{ℓ}, where ℓ and $\jmath$ are the latest two landmarks (splitting point). An example is shown in Fig. 1. The sliding window and each slice all have the same length of 6 time units. Current time is 15 and current sliding window is W^{15}_{9}, overlapping with slice-2 W^{12}_{6} and slice-3 W^{18}_{12}. Slice-3 is an ongoing slice and only has the length of 3 time units at current time 15 (though there are four edges in it as two edges arrive at the same time at $t = 15$). The sliding window may also overlap with only a single slice. For example, at time 12 it only overlaps with slice-2. Similar to the baseline method, we use a hash function $G(\cdot)$ to generate priority for each edge, and use another hash function $H(\cdot)$ to split the streaming graph into k substreams. Then in each substream, we can easily retain the edge with the largest priority in each slice, as the splitting points are fixed. We use ϵ to represent the edge with the largest priority in a slice from t_1 to t_2 in the a substream. Then we just need to set ϵ empty at time t_1, and replace it with an incoming edge e if $G(e) \geq G(\epsilon)$ or ϵ is empty until t_2.

Because the sliding window overlaps with at most two slices, we record two edges in the i_{th} substream ($0 \leq i \leq k-1$). They are the edge with the largest priority in current slice W^{T}_{ℓ} and the edge with the largest priority in the last slice $W^{\ell}_{\jmath}$, respectively, denoted as $\epsilon[i]$ and $\beta[i]$. Only one of them may participate in the sample graph. There are 3 cases, as shown in Fig. 4.

Case 1: In Case 1, both $\beta[i]$ and $\epsilon[i]$ are in the sliding window. The larger one of them is the edge with the largest priority in the sliding window in this substream, and it is the valid sample edge in this substream.

Case 2: With time passing by, Case 1 transfers to Case 2. In this case, $\beta[i]$ has already expired, but the sliding window still overlaps with $W^{\ell}_{\jmath}$. If $G(\beta[i]) > G(\epsilon[i])$, we cannot select a valid sample edge from this substream, because unexpired

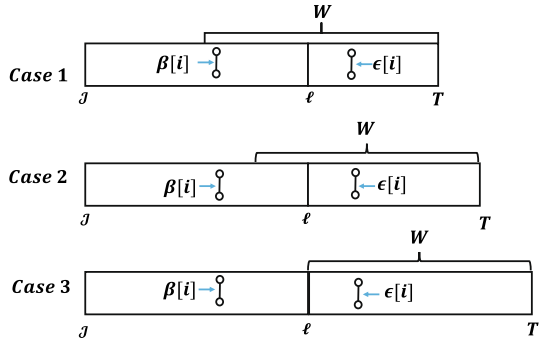


Fig. 4 Different Cases in SWTC

edges in W_J^ℓ are unknown to us. There may exist edges e' in W_J^ℓ, where $G(\beta[i]) > G(e') > G(\epsilon[i])$ and $t(e') > T - N$. Therefore we cannot determine if $\epsilon[i]$ has the largest priority in the sliding window. There is no sample edge in this substream in this case. On the other hand, if $G(\epsilon[i]) \geq G(\beta[i])$, $\epsilon[i]$ is a valid sample. It is the edge with the largest priority in W_ℓ^T, and also has larger priority than all edges in W_J^ℓ, since it has priority no less than $\beta[i]$. We can determine that it has the largest priority in the sliding window.

Case 3: After Case 2, the sliding window further slides and arrives at a new landmark. The sliding window no longer overlaps with W_J^ℓ. In other words, $W = W_\ell^T$. In this case, $\epsilon[i]$ is the valid sample edge in the sliding window.

After Case 3, a new slice emerges, and the situation returns to Case 1. The above three cases repeats recursively.

Algorithm 1 Processing new edge in the SWTC

Require: edge e = $\langle s, d \rangle$
Ensure: updated sample
1: $p \leftarrow H(e)$
2: **if** $\epsilon[p] = e$ **then**
3: Update the timestamp of $\epsilon[p]$
4: **else**
5: **if** $\epsilon[p] = null$ *or* $G(\epsilon[p]) \leq G(e)$ **then**
6: $\mathcal{G}_s.remove(\epsilon[p])$
7: $\epsilon[p] \leftarrow e$
8: **if** $\beta[p]$=*null or* $G(\beta[p]) \leq G(e)$ **then**
9: $\mathcal{G}_s.add(e)$
10: $\mathcal{G}_s.remove(\beta[p])$
11: **end if**
12: **end if**
13: **end if**

Algorithm 2 $\mathcal{G}_s.add(\cdot)$

Require: edge e
Ensure: updated sample
1: $\mathcal{G}_s.InsertEdge(e)$
2: $\mathcal{G}_s.IncreaseTriangle(e)$

Algorithm 3 $\mathcal{G}_s.remove(\cdot)$

Require: edge e
Ensure: updated sample
1: **if** $e! = NULL$ && e *is sampled* **then**
2: $\mathcal{G}_s.DecreaseTriangle(e)$
3: $\mathcal{G}_s.DeleteEdge(e)$
4: **end if**

Edge processing: Algorithm 1 shows how to process a new edge e. Firstly, edge e is hashed to the $H(e)$-th substream, where $H(\cdot)$ is a hash function. Let $p = H(e)$ (Line 1 in Algorithm 1). If edge e is same with the recorded edge $\epsilon[p]$, we just update $\epsilon[p]$'s timestamp to be the current time point T (Lines 2-3). Otherwise we compare it with $\epsilon[p]$. There are 2 cases:

1. If $\epsilon[p]$ is empty or $G(\epsilon[p]) \leq G(e)$, we update edge $\epsilon[p]$ to be e (Lines 5-7). In this case, if the old $\epsilon[p]$ is a sample edge in $\mathcal{G}_s$, we need to remove ϵ from $\mathcal{G}_s$ and decreases the number of triangles containing $\epsilon[p]$ from the triangle counter (Lines 6).
2. If $G(\epsilon[p]) > G(e)$, we do nothing.

Furthermore, in the first case, we need to further check $\beta[p]$ after replacing $\epsilon[p]$ with e. If $\beta[p] = null$ or $G(\beta[p]) \leq G(e)$, we can conclude that edge e should be selected as a sample edge and inserted into sample graph $\mathcal{G}_s$. We need to add the number of triangles containing e (Lines 8-9). If $\beta[p]$ is a sample edge in $\mathcal{G}_s$, we need to delete it and reduce the number of triangles containing $\beta[p]$ from the counter (Lines 10).

Edge Expiration: In order to delete the expired sample edges, we need to continuously monitor the oldest edge in the sample graph $\mathcal{G}_s$. This can be easily achieved with a linked list maintaining the time sequence of the sample edges. Once the oldest edge expires, which means its timestamp is smaller than $T - N$, we delete it from $\mathcal{G}_s$ and decrease the triangle counter.

Slice Switching: At a landmark, namely Case 3 in Fig. 4, we scan the k substreams. In each substream, we set $\beta[i] = \epsilon[i]$ and $\epsilon[i] = null$, as a new slice is about to emerge. Furthermore, if $G(\epsilon[i]) < G(\beta[i])$ in the i_{th} substream before the scanning, edge $\epsilon[i]$ becomes a sample edge now. We insert it into $\mathcal{G}_s$ and increase the triangle counter.

4.2 Valid sample size analysis

The accuracy of the sampling-based triangle count estimation depends on the sample graph size $|\mathcal{G}_s|$. Larger $|\mathcal{G}_s|$ leads to more accurate estimation result, which will be analyzed in Sect. 4.4. In this subsection, we mathematically analyze $|\mathcal{G}_s|$ in our method SWTC and compare it with the baseline approach (proposed in Sect. 3). We first give a brief analy-

sis about the space consumption of SWTC and the baseline method. Then we analyze their valid sample size under the same memory usage.

Space Analysis: SWTC and the baseline method both consume $O(k)$ memory, and their memory consumption is the same given the same substream number k. For both SWTC and the baseline, we need a table of k bins to maintain the sample set, with each bin corresponding to a substream. In each bin, both methods need to store 2 edges. In the baseline method, we need to store the test edge and the sample edge. In SWTC, we store the edge with the largest priority in each slice and two slices are maintained. Besides, the maximum size of the sample graph is k edges for both algorithm. The same memory is needed to be preserved for the sampled graph in both algorithms. As k decides the amount of memory these algorithms consume, it should be set according to the available memory in applications.

Valid Sample Size Analysis: Based on the analysis above, we compare the valid sample size of SWTC and the baseline method given the same substream number k. We have the following results about the expectation of the valid sample size (**expected sample size** for short) in our approach SWTC and the baseline method in Theorems 1 and 2, respectively.

Theorem 1 *In SWTC, the expected sample size is* $k \times \frac{|W|}{|W_J^T|}$. *We use* J *to denote the second largest landmark which satisfies* $J \leq T$, *and* k *is the number of substreams.*

Proof Fig. 4 shows that we get a valid sample in a substream if and only if the edge with the largest priority in W_J^ℓ and W_ℓ^T lies in the sliding window W. We use ρ to denote the probability of this event at time T, namely the probability that a substream gets a valid sample. Suppose there are η distinct edges in this substream in period W, and η' distinct edges in this substream in period $W_J^T = W_J^\ell \cup W_\ell^T$. Because each edge gets a random priority, the probability that the edge with the largest priority in W_J lies in W is $\frac{\eta}{\eta'}$. Moreover, because edges are mapped to different substreams randomly, $\frac{\eta}{\eta'}$ is equal to $\frac{|W|}{|W_J|}$. In conclusion, the probability that a substream gets a valid sample edge is $\rho = \frac{|W|}{|W_J^T|}$, and the expected sample size in k substreams is $k \times \rho = k \times \frac{|W|}{|W_J^T|}$ □

Theorem 2 *For the baseline method, the expected sample size is in range* $\left[k \times \frac{|W|}{|W_{T-2N}^T|}, k \times (1 - \frac{|W_{T-2N}^{T-N}|}{|W_{T-3N}^T|})\right]$.

Proof We use ρ to denote the probability that BPS algorithm in a substream gets a valid sample. The expected sample size of the baseline method is $k \times \rho$. However, it is difficult to give an exact expression of ρ, as it is cumulatively affected by all the edges arriving before W. The original paper of BPS algorithm [19] only gives a lower bound of ρ. And we further give an upper bound here.

Lower bound: When we use BPS algorithm in a substream, we will get a valid sample if the edge with the largest priority in W has a larger priority than the test edge ϵ_{test} which arrives before $T - N$. In the worst case, ϵ_{test} is the edge with the largest priority in W_{T-2N}^{T-N}. Therefore the edge with the largest priority in W_{T-2N}^T needs to be in W. According to the proof of Theorem 1, we can see that this probability is $\frac{|W|}{|W_{T-2N}^T|}$. Therefore, $\rho \geq \frac{|W|}{|W_{T-2N}^T|}$.

Upper bound: From the former proof, we know that in a substream, if the edge with the largest priority in W_{T-3N}^{T-N} lies in W_{T-2N}^{T-N}, this edge, which we represent with e', will definitely become a valid sample edge until it expires. By the time of T, it becomes a test edge. And if it also has larger priority than the edges in W, it prevents edges in the sliding window W from becoming valid sample, and there will be no valid sample edge in this substream. In other words, e' is the edge with the largest priority in W_{T-3N}^T, and it lies in W_{T-2N}^{T-N}. In this case there will definitely be no valid sample edge in this substream. According to the former proof, this probability is $\frac{|W_{T-2N}^{T-N}|}{|W_{T-3N}^T|}$. Therefore, $\rho \leq 1 - \frac{|W_{T-2N}^{T-N}|}{|W_{T-3N}^T|}$.

Combing the lower bound and the upper bound together, we know that $\frac{|W|}{|W_{T-2N}|} \leq \rho \leq 1 - \frac{|W_{T-2N}^{T-N}|}{|W_{T-3N}|}$. Therefore, the expected sample size of the baseline method is $k \times \rho$ and lies in range $\left[k \times \frac{|W|}{|W_{T-2N}^T|}, k \times (1 - \frac{|W_{T-2N}^{T-N}|}{|W_{T-3N}^T|})\right]$. □

According to Theorems 1 and 2, the value of ρ depends on the cardinality in different periods, which varies according to both the length of the period and the throughput of the stream. In order to make ρ intuitive and comparable, we assume that the throughput of the streaming graph is steady. Then, we have the following result.

Theorem 3 *Assume that the throughput of streaming graph is st steady. In SWTC, the expected sample size varies in range* $[0.5k, k]$ *periodically. In the baseline method, the expected sample size is in range* $[0.5k, 0.66k]$.

Proof When the throughput of the streaming graph is steady, the cardinality in a window $W_{t_1}^{t_2}$ is linear relevant with its length $t_2 - t_1$. Based on this condition, we can compute numerical results about the expected sample size.

For SWTC, the expected sample size is $k \times \frac{|W|}{|W_J^T|} = k \times \frac{|W|}{|W_J^\ell \cup W_\ell^T|}$. The length of W_J^ℓ is always N, but the length of W_ℓ^T varies from 0 to N periodically. The minimum value and the maximum value appear before and after slice switching at a landmark, respectively. Therefore the expected sample size in $SWTC$ varies from $0.5k$ to k in a cycle. The average value is $k \times \frac{\int_{i=0}^{N} \frac{N}{i+N} di}{N} = k \times ln2 \approx 0.7k$.

In the baseline method, as it is hard to get an exact expression of the expected sample size, we cannot com-

pute its accurate value. But we can get its upper bound and lower bound as shown in Theorem 2, The lower bound is $k \times \frac{|W|}{|W^T_{T-2N}|} = k \times \frac{N}{2N} = 0.5k$, and the upper bound is $k \times \left(1 - \frac{|W^{T-N}_{T-2N}|}{|W^T_{T-3N}|}\right) = k \times \left(1 - \frac{N}{3N}\right) = 0.66k$. In Sect. 7.6, the experimental results show that the value of ρ is nearly a fixed value of 0.56. □

From the above analysis, we can see that the average sample size of SWTC is larger than the upper bound of the baseline method. Figure 7 in Sect. 7 confirms our analysis.

4.3 Estimating of triangle count

In this section, we show how to estimate triangle count in the snapshot graph $\mathcal{G}$ with the sample graph $\mathcal{G}_s$.

Suppose there are n distinct edges in $\mathcal{G}$, and m valid sample edges in $\mathcal{G}_s$. We use tc to denote the triangle count in $\mathcal{G}_s$. Because each edge in the sliding window has an equal chance to become one of the m valid sample edges, the probability that all the three edges in a triangle are selected is $\frac{m(m-1)(m-2)}{n(n-1)(n-2)}$. We can estimate the number of triangles in the sliding window as $tc \times \frac{n(n-1)(n-2)}{m(m-1)(m-2)}$. Detailed proof can be found in Sect. 4.4.

It is difficult to directly estimate n, namely the number of distinct edges in the sliding window. Existing algorithms like [20] cannot deal with edge expiration. However, we split the streaming graph into slices, and these slices can be viewed as fixed time windows with no edge expiration. Therefore, prior algorithms in cardinality estimation can be used in these slices. We can first estimate the cardinality of the slices which overlap with the sliding window, and then estimate the cardinality of the sliding window with it. More fortunately, as we have already stored the largest priority in each substream, we can easily transform these priorities into a HyperLogLog sketch [20] for cardinality estimation, and no other data structure is needed. HyperLogLog sketch is also the state-of-the-art for cardinality estimation.

For the i_{th} substream, we have stored $G(\beta[i])$ and $G(\epsilon[i])$. The larger one between them, denoted with θ, is the largest priority in this substream in W^T_J. It can be transformed to a variable $R[i] = \lceil -log(1-\theta) \rceil$ with $Geometric(1/2)$ distribution. If the substream is empty (both $\beta[i]$ and $\epsilon[i]$ in it are empty), we set $R[i] = 0$. Such variables in all the k substreams form a HyperLogLog sketch [20] that estimates the cardinality of W^T_J, namely $|W^T_J|$. $|W^T_J|$ can be computed as $\frac{\alpha_k k^2}{\sum_{i=1}^{k} 2^{-R[i]}}$. This equation is derived in HyperLogLog algorithm, and $\alpha_k = 0.7213/(1 + 1.079/k)$ for $k > 128$. The error bound is also the same as the analysis in [20].

Then we further estimate the cardinality of the sliding window W, namely n, with $|W^T_J|$. Suppose there are m valid samples, and M substreams that are not empty. According to Theorem 1, we can get a valid sample in a substream with probability $\frac{|W|}{|W^T_J|} = \frac{n}{|W^T_J|}$, which can be estimated as $\frac{m}{M}$. Therefore we have $n = |W^T_J| \times \frac{m}{M}$.

Note that $M \approx k$ in most cases. For a substream, the probability that one edge is not mapped to it is $1 - \frac{1}{k}$. And the probability that all n distinct edges in the sliding window are not mapped to it is $\left(1 - \frac{1}{k}\right)^n \approx e^{-\frac{n}{k}}$ (e is the Euler number). This is the probability that a substream is empty. When $k \leq 0.2n$, which is the common case in applications, such probability is only 0.67%. At any time point, very few substreams will be empty, and as we maintain a large number of substreams, several empty substreams will not influence the total sample size, neither the accuracy.

4.4 Accuracy analysis

4.4.1 Discussion about hash function independence

Because hash functions are computed according to edge IDs, hash-based sampling is a pseudo random and deterministic procedure. We cannot theoretically guarantee strict independence of the sampling. However, in most cases, the correlation between sampling and topology properties of edges is very weak. As a result, estimation with such sampling provides similar accuracy as a strictly independence sample. Such statistical property is due to 2 facts:

First, the hash functions nowadays are usually composed of complex bit operations and have strong mixing ability. Small changes of input can cause large changes in the output. The distribution of hash results usually adheres strongly to a uniform distribution. Though the IDs of edges may show correlation with their topology properties, such correlation will be dramatically weakened after hash computation. The hash value distribution of edge sets with different topology properties has little difference, which means low correlation level and approximate statistical independence [26] (as shown in the chi-square independence check in Sect. 7.4). For example, edges with large number of neighbors may have certain ID patterns, like gathering in region $1000 \sim 2000$ or being times of 8, but after hash computation, the hash value of these edges will be spread all over the hash value range.

Second, the applications of our sampling method are not antagonistic. Natural correlation between edge IDs and topology properties is usually simple and can be eliminated by hash functions. But if the data is deliberately manipulated, bias may still happen. Fortunately, there are no malicious adversaries in applications of our algorithm. Even in spam detection, the spammer does not tend to filter the addresses to send spam messages. And we can also keep the parameters (like seeds) of hash functions secret as a defence.

Therefore, though we use hash functions, the correlation between hash values and topology properties is weak. In our

paper, we approximately consider the hash functions we use as completely random and independent, and they generate a uniform output. In Sect. 7.4, we also carry out chi-squared test to check the statistical independence between the hash values and the number of involved triangles of each edge.

Such hash based sampling has been widely used in network traffic measurement [27,28], graph sampling [9,29] and systems like Hive [30]. Even with simple hash functions like modular functions [27], the samples they produce show good independence and uniformity. [31] also carries out an experimental study of different hash functions in sampling.

However, we need to emphasize again that sampling with hash functions is pseudo random and not theoretically independent. We can neither give a theoretical guarantee of the difference between our sampling and a strictly independent sample. When dealing with duplication with hash functions, our algorithm is heuristic, and the following mathematical analysis is only an approximation. This is a problem faced by most hash-based algorithms, including sketches [20,32] and sampling methods [27,28]. How to deal with duplication in streaming sampling with a strict theoretical guarantee is still a challenging problem. On the other hand, when there is not edge duplication, we can use random functions to generate priority and divide substreams, and all the following analysis holds strictly.

4.4.2 Uniformity of sampling

In this section, we prove that the sample we generate is a uniform sample. For each distinct edge e in the sliding window, the substream it is mapped into has probability ρ to produce a valid sample (ρ is analyzed in Sect. 4.2). If this substream has a valid sample, the sample is the edge with the largest priority in this substream in the sliding window. As the hash function $H(\cdot)$ is a uniform function, each substream has $\frac{|W|}{k}$ distinct edges in expectation. As the priority of each edge is randomly generated, each distinct edge has the same probability to get the largest priority. Thus the probability that e gets the largest priority in its substream and gets sampled is $\rho \times \frac{k}{|W|}$. This probability applies to all distinct edges in the sliding window. Therefore, the sample we generate is a uniform sample.

4.4.3 Error analysis

Based on above analysis, in this section we first prove that our estimation of the triangle count is unbiased, then we give some mathematical analysis about the variance of the triangle estimation.

Theorem 4 *Suppose at time T, SWTC gets m valid sample edges. There are n distinct edges in the snapshot graph (namely $|W| = n$), and the number of triangles induced by these sample edges is tc. We use Δ to present the set of triangles in the snapshot graph, and its number is τ. We introduce variable $\hat{\tau} = \frac{tc}{\gamma_3}$ where γ_3 is defined as*

$$\gamma_j = \frac{m(m-1)...(m-j+1)}{n(n-1)...(n-j+1)} \tag{1}$$

Then we have:

$$E(\hat{\tau}) = \tau \tag{2}$$

$$Var(\hat{\tau}) = \tau\theta_3 + 2\zeta\theta_5 + 2\eta\theta_6 \tag{3}$$

where ζ is the number of unordered pair of distinct triangles in Δ which share one edge, and $\eta = \frac{1}{2}\tau(\tau-1) - \zeta$ is the number of unordered pairs of distinct triangles in Δ which share no edge. And we define $\theta_3 = \frac{1}{\gamma_3} - 1$, $\theta_5 = \frac{\gamma_5}{(\gamma_3)^2} - 1$, $\theta_6 = \frac{\gamma_6}{(\gamma_3)^2} - 1$

Proof First we prove the correctness of the expectation. We propose the following lemma: □

Lemma 1 *At time T, the probability of SWTC sampling edge $e_1, e_2....e_j$ given m is*

$$P(e_1, e_2.....e_j \in \mathcal{G}_s) = \gamma_j \tag{4}$$

where γ_j is defined as Eq. 1.

Given j different edges $e_1, e_2....e_j$ and a set of different substreams $\{\mathcal{S}_{c_1}, \mathcal{S}_{c_2}...\mathcal{S}_{c_j}\}$, where all these substreams have valid sample edges. We can compute the probability that edge e_i is sampled in substream $\mathcal{S}_{c_i}$ $(1 \le i \le j)$. As analyzed in Sect. 4.4.2, the sample set we generate is a uniform sample. Thus we can find that any j different edges has equal probability to be sampled in these substreams. There are totally $n(n-1)....(n-j+1)$ different ways of selecting j different edges and putting them into these substreams. Therefore the probability that a particular combination is selected is $\frac{1}{n(n-1)....(n-j+1)}$.

Because there are totally m substreams with valid samples, there are $m(m-1)....(m-j+1)$ different ways to select indexes $\{c_1, c_2...c_j\}$. Therefore, the overall probability that j edges $e_1, e_2....e_j$ are sampled as valid sample edges are:

$$P(e_1, e_2.....e_j \in \mathcal{G}_s) = \frac{m(m-1)....(m-j+1)}{n(n-1)....(n-j+1)}$$

Recall that $m = \rho k$. The probability that an edge is sampled is $\frac{m}{n} = \rho \times \frac{k}{n}$ where $n = |W|$. This is also in accord with the analysis in Sect. 4.4.2. According to this lemma, we find that any triangle with three edges e_1, e_2, e_3 in the snapshot graph $\mathcal{G}$ has probability $\gamma_3 = \frac{m(m-1)(m-2)}{n(n-1)(n-2)}$ to be included in

the sample. Therefore, given the number of triangles in the sample, tc, we have:

$$E(\hat{\tau}) = E\left(\frac{tc}{\gamma_3}\right) = \tau$$

Next we compute the variance of $\hat{\tau}$. For a triangle σ in the snapshot graph $\mathcal{G}$, we set a variable ξ_σ to be 1 if all the 3 edges of σ are valid sample edges at time T and 0 otherwise. We can compute the variance of $\hat{\tau}$ given the number of valid sample edges m as:

$$\begin{aligned}
&Var\left(\hat{\tau}\right) \\
&= Var(\frac{\sum_{\sigma\in\Delta}\xi_\sigma}{\gamma_3}) \\
&= \frac{\sum_{\sigma,\sigma*\in\Delta} Cov(\xi_\sigma, \xi_{\sigma*})}{(\gamma_3)^2} \\
&= \frac{\sum_{\sigma\in\Delta} Var(\xi_\sigma)}{(\gamma_3)^2} \\
&+ \frac{\sum_{\sigma,\sigma*\in\Delta,\sigma\neq\sigma*} E(\xi_\sigma\xi_{\sigma*}) - E(\xi_\sigma)E(\xi_{\sigma*})}{(\gamma_3)^2}
\end{aligned}$$

According to lemma 1, we have

$$Var(\xi_\sigma) = \gamma_3 - (\gamma_3)^2 \tag{5}$$

$$E(\xi_\sigma)E(\xi_{\sigma*}) = (\gamma_3)^2 \tag{6}$$

$$E(\xi_\sigma\xi_{\sigma*}) = \begin{cases} \gamma_5 & \sigma \text{ and } \sigma* \text{ share one edge.} \\ \gamma_6 & \sigma \text{ and } \sigma* \text{ share no edge.} \end{cases} \tag{7}$$

Given the definition of ζ, η, θ_3, θ_5 and θ_6, we can get Eq. 2 in Theorem 4 with the former equations. □

The expectation and variance of the baseline method can be computed similarly. Because the number of valid sample edges in the baseline method is smaller than SWTC, it has a larger variance compared to SWTC.

4.5 Time cost analysis

We have analyzed the space cost of SWTC and the baseline method in Sect. 4.2. In this Section, we further analyze their time cost.

4.5.1 Time cost of SWTC

For SWTC, the time cost can be divided into 2 parts. The first part is the cost of maintaining the sample edge set. The second part is the cost of counting triangles induced by the sample edges. We will discuss these 2 parts separately.

The first cost can be further divided into 3 components: the cost of processing new edges, the cost of deleting expired samples, and the cost of slice switching in each substream at landmarks. The cost of processing a new edge includes computation of hash functions $H(\cdot)$ and $G(\cdot)$, and the cost of compare the new edge with the two stored edges in the mapped substream, which are all $O(1)$. Therefore, the cost of processing each new edge is $O(1)$. The cost of deleting one expired sample is also $O(1)$. As there are at most k times of sample expiration in N time units, and k is much smaller than the number of edge arrivals in such a slice, the amortized cost of sample expiration upon each edge arrival is beneath $O(1)$. At each landmark, SWTC needs to set $\beta[i] = \epsilon[i]$ and $\epsilon[i] = null$ in all the k substreams. Besides, in some substreams with no valid samples, we need to add $\epsilon[i]$ into the sample set. These operations introduce a cost of $O(k)$. This cost is amortized by all the edge arrivals in the N time units between two landmarks. As discussed above, k is much smaller than the number of edge arrivals in such a slice. Thus the amortized cost is beneath $O(1)$. Overall, for the first part, the amortized cost upon each edge arrival is $O(1)$

Then we consider the cost of the second part. When a sample edge is inserted into or deleted from the sample graph, we need to update the triangle counter tc with triangle counting functions $IncreaseTriangle(\cdot)$ or $DecreaseTriangle(\cdot)$. These 2 functions compute the size of common neighbors of the edge's endpoints, and the cost is $O(\rho k)$. Recall that ρ is the probability that a substream produces a valid sample, and thus ρk is the number of edges in the sample graph. As ρ varies with time, the cost of such functions also varies with time. For simplicity of analysis, we use an approximate cost of $O(k)$ for these functions. Then the amortized cost of the second part is $O(\#TriangleCounting \times k)$, where $\#TriangleCounting$ is the amortized number of triangle counting function calls upon each edge arrival.

Next we analyze $\#TriangleCounting$. In the following analysis, we use ρ^t to represent the value of ρ at time t. Whenever an edge becomes a valid sample, it induces 2 calls of triangle counting functions, one upon its insertion, and another upon its deletion (no matter it is replaced or expires). As analyzed in Sect. 4.4, the sample we produce is a uniform sample. Therefore, if an edge arrives at time t and has no duplicate copy in the sliding window yet, it has probability $\frac{\rho^t k}{|W|}$ to become a new valid sample upon its arrival, inducing $\frac{2\rho^t k}{|W|}$ function calls. Besides, at each landmark L, there are $(1-\rho^L)k$ substreams which produce new samples (see the slice switching part of Sect. 4.1), inducing $2(1-\rho^L)k$ function calls. The amortized number of function calls upon each edge arrival is

$$\begin{aligned}
&\#TriangleCounting \\
&= \frac{\sum_{e\in\mathcal{S}} \frac{\rho^{t(e)}k}{|W^{t(e)}_{t(e)-N}|} + \sum_{L\in\mathcal{S}}(1-\rho^L)k}{||\mathcal{S}||} \times 2
\end{aligned} \tag{8}$$

$||\mathcal{S}||$ is the number of edge arrivals in the streaming graph. e denotes edges where there are no e' in the last N time

units before $t(e)$, and e' is a duplicate copy of e. L denotes landmarks in the streaming graph.

In order to make this cost intuitive, we consider a simple case where there are no duplicate edges in the streaming graph, and there is exactly one edge arrival in each time unit. In this case $|W^{t(e)}_{t(e)-N}|$ is always N. ρ^L is always 0.5 as discussed in Theorem 3, and the number of landmarks is $\frac{||\mathcal{S}||}{N}$. Thus the formula above can be transformed into

$$\#TriangleCounting = \left(\frac{\sum_{e\in\mathcal{S}}\rho^{t(e)}}{||\mathcal{S}||}\times\frac{k}{N}+\frac{0.5k}{N}\right)\times 2 \tag{9}$$

According to the analysis in Theorem 3, $\frac{\sum_{e\in\mathcal{S}}\rho^{t(e)}}{||\mathcal{S}||}\approx ln2$. Thus the number of function calls is $\frac{(2ln2+1)k}{N}$.

In summary, the amortized cost upon each edge arrival is the sum of $O(1)$ cost of the first part and $O(\#Triangle\ Counting\times k)$ cost of the second part, where $\#Triangle\ Counting$ is $\frac{(2ln2+1)k}{N}$ for a streaming graph with exactly one distinct edge in each time unit.

4.5.2 Time cost of the baseline method

The time cost of the baseline method can be analyzed similarly: The cost of maintaining the sample set including the cost of processing new edges, deleting expired samples and erasing double expired test edges. The cost of the first operation is $O(1)$, and the amortized cost of the other 2 operations is beneath $O(1)$. Overall, the amortized cost of maintaining the sample set upon each edge arrival is $O(1)$. The cost of maintaining triangle count is determined by the number of triangle counting function calls. Similar to SWTC, for each edge which arrives at time t and has no duplicate copy in the sliding window yet, it has probability $\frac{\rho^t k}{|W|}$ to become a new valid sample upon its arrival, inducing a cost of $\frac{2\rho^t k}{|W|}$ function calls. Upon the double expiration of each test edge, there will also be a new valid sample produced, inducing 2 function calls. Therefore, the number of triangle counting function calls upon each edge arrival is

$$\begin{aligned}&\#TriangleCounting\\&=\frac{\sum_{e\in\mathcal{S}}\frac{\rho^{t(e)}k}{|W^{t(e)}_{t(e)-N}|}+\#DoubleExpiration}{||\mathcal{S}||}\times 2\end{aligned} \tag{10}$$

When there is exactly one edge arrival in each time unit and no edge duplication, ρ is nearly a fixed value in the baseline method (as will be experimentally proved in Sect. 7.6), and $\frac{\sum_{e\in\mathcal{S}}\rho^{t(e)}}{||\mathcal{S}||}=\rho$. Besides, because there are $(1-\rho)k$ substreams with no valid sample in a sliding window, there are $(1-\rho)k$ times of test edge double expiration in N time units. Thus $\frac{\#DoubleExpiration}{||\mathcal{S}||}=\frac{(1-\rho)k}{N}$. We can transform the former formula to

$$\begin{aligned}&\#TriangleCounting\\&=\left(\rho\times\frac{k}{N}+\frac{(1-\rho)k}{N}\right)\times 2\quad=\frac{2k}{N}\end{aligned} \tag{11}$$

In summary, the amortized cost is the sum of $O(1)$ cost of the first part and $O(\#TriangleCounting\times k)$ cost of the second part, where $\#TriangleCounting$ is $\frac{2k}{N}$ for a streaming graph with exactly one distinct edge in each time unit.

If we use Treap-based BPS to maintain the sample set, the cost of counting triangle is the same as the above analysis, but the cost of maintaining sample set becomes $O(log(k))$ rather than $O(1)$. Therefore, Treap-based method is slower than the baseline method, which will also be confirmed in the experiments of Sect. 7.5.

Comparing the time cost of the baseline method with SWTC, we can find that the number of triangle counting function calls is larger in SWTC, as it produces more valid samples. Besides, the sample graph is larger in SWTC, and thus each function call is also slightly slower. Overall, the speed of SWTC is slower than the baseline method. We believe that this is a necessary cost for getting larger valid sample size and higher accuracy. Besides, The cost of maintaining the sample set is the same in both methods, and the amortized number of triangle counting function calls upon each edge arrival is small. Therefore, the difference in speed is not large.

5 Optimization techniques

There are still two problems in SWTC: computation peak at landmarks, and periodic trough of sample size. In this section, We propose two optimization techniques: vision counting (Sect. 5.1) and asynchronous grouping (Sect. 5.2) to solve these problems.

5.1 Vision counting

Although SWTC can generate a larger sample graph and produce a more accurate triangle count estimation, there is a performance problem when the sliding window reaches landmarks, i.e., Case 3 in Fig. 4. Assume that $G(\beta[i])>G(\epsilon[i])$ in Case 2, there is no sample edge in the i_{th} substream. But, when the sliding window reaches a landmark (Case 2 is transferred into Case 3), a new sample edge will be generated. This case may happen in multiple substreams simultaneously, and it will lead to the emerging of large quantities of new samples at the same time. Adding these edges into $\mathcal{G}_s$ and counting the number of increased triangles will bring peak of computation cost, and may sharply increase the latency of processing new

edges. To address this issue, we propose a new technique named *vision counting*. This technique spreads the computation overhead of Case 3 over the entire sliding window period, so that we can avoid the burst of computation cost.

In the *vision counting* technique, we maintain 2 counters in $\mathcal{G}_s$. One is the effective triangle counter tc, and the other vc maintains number of triangles composed of $\epsilon[i]$ in all substreams. When $G(\epsilon[i]) < G(\beta[i])$ in Case 2 of Fig. 4, no sample edge is selected in this substream. However, we can forecast that at the next landmark, $\epsilon[i]$ will become a new sample edge. We insert it into $\mathcal{G}_s$, but tag it as an *invalid sample*. vc maintains the number of triangles composed of all $\epsilon[i]$, whether they are valid samples or not. We continuously maintain the set of sample edges and invalid sample edges, as well as counter tc and vc. When a landmark comes, because all $\epsilon[i]$ will become valid samples, the value of vc is just the triangle count in the snapshot graph. We simply set $tc = vc$. Besides, we need to set $\beta[i] = \epsilon[i]$ and $\epsilon[i] = null$ in each substream and set $vc = 0$, in order to prepare for a new slice. Compared to the basic version, massive triangle counting at landmarks is avoided. The computation cost at peak time is spread over the entire window sliding, in the form of maintaining vc.

5.2 Asynchronous grouping

5.2.1 Implementation

In SWTC, the expectation of valid sample size is $k \times \frac{|W|}{|W_j^T|}$ (Theorem 1), where k is the number of substreams in SWTC. It varies as the window slides. When the throughput of the stream is steady, it varies from $0.5k$ to k in a cycle (Theorem 3). The instability of valid sample size in SWTC is a huge drawback. In some cases the valid sample size may reach the minimum value, and a small valid sample size brings a low accuracy.

In this section, we propose a technique, Asynchronous Grouping, AG-technique for short, to solve this problem. This technique will stabilize the valid sample size. It will increase the lower bound of the expected sample size under the same memory usage, and sharply shrink the *fluctuation range*, which is defined as follows:

Definition 5 **Fluctuation range:** the gap between the maximum value and the minimum value of the expected sample size of SWTC in a streaming graph with steady throughput.

Before optimization, the fluctuation range of SWTC is $0.5k$, and the minimum sample size is $0.5k$. With AG-technique, we can shrink the fluctuation range by more than 10 times, and arise the minimum sample size to $0.67k$. With a much higher lower bound of the sample size, we can reduce the risk of getting low accuracy.

The idea of AG-technique is simple yet effective: we can divide the substreams into multiple groups. These groups use asynchronous slicing, which means they use different landmark sequences. In each group, the expected sample size varies as the window slides, but as a whole, the sample size of these groups is stable. We call the algorithm with AG-technique as SWTC-AG.

To be precise, we divide the k substreams into g groups $\{GP_i | 0 \leq i \leq g-1\}$, with $\frac{k}{g}$ substreams in each group, and group GP_i contains substreams $i\frac{k}{g}$ to $(i+1)\frac{k}{g} - 1$. In each group, there is a landmark sequence, with a gap of N time units between adjacent landmarks. In different groups, the landmark sequences are *asynchronous*. The i_{th} landmark in GP_{j+1} is larger than the i_{th} landmark in GP_j by $\frac{N}{g}$, and the first landmark of group GP_0 is 0. An example of the landmarks and slices is shown in Fig. 5. In this example the number of groups is set to $g = 3$.

Each group works as a SWTC algorithm:

Sampling: In substream j of group GP_i, we maintain 2 edges, denoted as $\epsilon[j]$ and $\beta[j]$. They are edges with the largest priority in the last 2 slices in this substream. Note that these slices are partitioned with the landmark sequence of GP_i, and different groups have different landmark sequences. We compare these 2 stored edges, and try the select the edge with the largest priority in the sliding window in this substream as a sample edge, as discussed in Fig. 4 of Sect. 4.1. The sample edges from all the g groups together form the sample graph. We continuously monitor the triangle count tc in the sample graph.

Estimating: In group GP_i, we also monitor the number of distinct edges mapped to it in the sliding window. The procedure is the same as discussed in Sect. 4.3, except that only the $\frac{k}{g}$ substreams in this group take part in the computation. We add the estimated edge number of the g groups together, and get the number of distinct edges in the sliding window. Then we can scale up the triangle count tc in the sample graph to the snapshot graph with the estimated sliding window cardinality n as $tc \times \frac{n(n-1)(n-2)}{m(m-1)(m-2)}$, where m is the number of edges in the sample graph.

Slice switching: There is another difference between SWTC-AG and SWTC. In SWTC, we encounter a landmark whenever the window slides N time units, and we update all the k substreams at the landmark. On the other hand, In SWTC-AG, we will encounter a landmark whenever the window slides $\frac{N}{g}$. This landmark belongs to a specific group GP_i, and we only update the substreams in GP_i. In the j_{th} substream which belongs to GP_i, we set $\beta[j] = \epsilon[j]$ and $\epsilon[j] = NULL$. Furthermore, if $G(\epsilon[j]) < G(\beta[j])$ before the updating, edge $\epsilon[j]$ becomes a sample edge now. We insert it into the sample graph and increase the triangle counter.

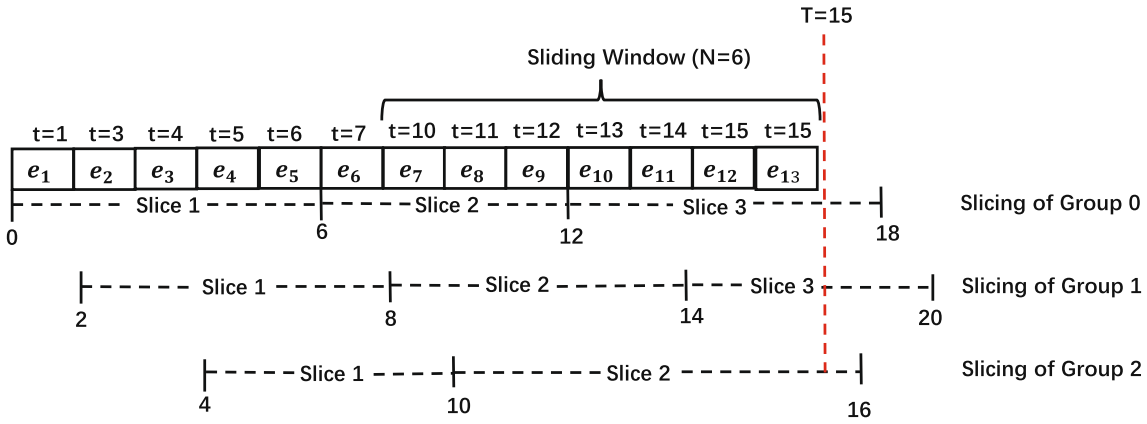


Fig. 5 Example of Slicing of SWTC-AG

The AG technique also relieves the problem of computation peak discussed in Sect. 5.1. We spread the cost of switching slices to g landmarks in each N time units. And at each landmark we only need to update $\frac{k}{g}$ substreams. Therefore, we will not suffer from heavy computation cost at one landmark like in the basic version of SWTC. Furthermore, AG technique can also cooperate with vision counting technique to further smooth the computation cost.

5.2.2 Valid sample size analysis

Next we analyze the sample size in SWTC-AG. We use ℓ_i and $_{Ji}$ to represent the latest landmark and the last landmark in group GP_i in the following analysis. We propose the following theorem:

Theorem 5 *The expected sample size of SWTC-AG is* $\frac{k}{g} \times |W| \times \left(\sum_{i=0}^{g-1} \frac{1}{|W_{Ji}^T|}\right)$

Proof According the Theorem 1, in the i_{th} group which has $\frac{k}{g}$ substreams, the expected sample size is $\frac{k}{g} \times \frac{|W|}{|W_{Ji}^T|}$ where T is current time and N is the length of the sliding window. The sample size of g groups is $\sum_{i=0}^{g-1}\left(\frac{k}{g} \times \frac{|W|}{|W_{Ji}^T|}\right) = \frac{k}{g} \times |W| \times \left(\sum_{i=0}^{g-1} \frac{1}{|W_{Ji}^T|}\right)$. □

This sample size is influenced by the throughput of the streaming graph. For the convenience of analysis, we suppose the throughput is steady, which means the number of distinct edges in a slice is linear related with its length. Then we have the following theorem:

Theorem 6 *When the throughput of the streaming graph is steady, the expected sample size of SWTC-AG varies in range* $\left(k \times \left(\sum_{i=g+1}^{2g} \frac{1}{i}\right), k \times \left(\sum_{i=g}^{2g-1} \frac{1}{i}\right)\right]$.

Proof In the i_{th} group, we denote the length of the latest slice $W_{\ell_i}^T$ as $\delta_i N$. Then the sample size is:

$$\begin{aligned} &Expected\ Sample\ Size \\ &= |W| \times \frac{k}{g} \times \left(\sum_{i=0}^{g-1} \frac{1}{|W_{Ji}^T|}\right) \\ &= \frac{k}{g} \times \left(\sum_{i=0}^{g-1} \frac{|W|}{|W_{Ji}^{\ell_i} \cup W_{\ell_i}^T|}\right) \\ &= \frac{k}{g} \times \left(\sum_{i=0}^{g-1} \frac{N}{N + \delta_i N}\right) \\ &= \frac{k}{g} \times \left(\sum_{i=0}^{g-1} \frac{1}{1 + \delta_i}\right) \end{aligned} \tag{12}$$

Note that we can transform $\frac{|W|}{|W_{Ji}^{\ell_i} \cup W_{\ell_i}^T|}$ to $\frac{N}{N+\delta_i N}$ because under the steady throughput assumption, the cardinality of a slice is linear related with its length.

At any time point which is the multiple of $\frac{N}{g}$, there is a landmark belonging to one of the g groups. Therefore, for any time T, there is always a group GP_c $(0 \le c < g)$ where $\ell_c = \left\lfloor (T/\frac{N}{g}) \right\rfloor \times \frac{N}{g}$ and $0 \le T - \ell_c < \frac{N}{g}$ (e.g. GP_1 in Fig. 5). Recall that we denote the length of $W_{\ell_c}^T$ as $\delta_c N$. Therefore $0 < \delta_c < \frac{1}{g}$. Suppose ℓ_c is the q_{th} landmark in the landmark sequence of GP_c (e.g., $q = 2$ for GP_1 in Fig. 5).

For a group GP_i $(0 \le i < c)$, its latest landmark ℓ_i is also the q_{th} landmark, and $\ell_c - \ell_i = (c - i) \times \frac{N}{g}$ (e.g., GP_0 in Fig. 5). Therefore we have

$$\begin{aligned} \delta_i &= \frac{T - \ell_i}{N} = \frac{(T - \ell_c) + (\ell_c - \ell_i)}{N} \\ &= \delta_c + \frac{c - i}{g} \end{aligned} \tag{13}$$

For group GP_i $(c < i \leq g-1)$, its q_{th} landmark is $\ell_c + (i-c) \times \frac{N}{g} > T$. This landmark has not arrived yet. Therefore, its latest landmark is the $(q-1)_{th}$ landmark, and is N time units smaller than the q_{th} landmark (e.g., GP_2 in Fig. 5). Therefor, ℓ_i is $\ell_c + (i-c) \times \frac{N}{g} - N = \ell_c - (g + c - i) \times \frac{N}{g}$. In this group, we have

$$\begin{aligned} \delta_i &= \frac{T - \ell_i}{N} = \frac{(T - \ell_c) + (\ell_c - \ell_i)}{N} \\ &= \delta_c + \frac{g + c - i}{g} \end{aligned} \tag{14}$$

According to Eq. 13 and 14, we can further change Eq. 12 to

$$\begin{aligned} &Expected\ Sample\ Size \\ &= \frac{k}{g} \times \left(\sum_{i=0}^{g-1} \frac{1}{1+\delta_i} \right) \\ &= \frac{k}{g} \times \left(\frac{1}{1+\delta_c} + \sum_{i=0}^{c-1} \frac{1}{1+\delta_i} + \sum_{i=c+1}^{g-1} \frac{1}{1+\delta_i} \right) \\ &= \frac{k}{g} \times \left(\frac{1}{1+\delta_c} + \sum_{i=0}^{c-1} \frac{1}{1+\delta_c + \frac{c-i}{g}} + \sum_{i=c+1}^{g-1} \frac{1}{1+\delta_c + \frac{g-i+c}{g}} \right) \\ &= \frac{k}{g} \times \left(\frac{1}{1+\delta_c} + \sum_{x=1}^{c} \frac{1}{1+\delta_c + \frac{x}{g}} + \sum_{y=c+1}^{g-1} \frac{1}{1+\delta_c + \frac{y}{g}} \right) \\ &= \frac{k}{g} \times \left(\sum_{x=0}^{g-1} \frac{1}{1+\delta_c + \frac{x}{g}} \right) \\ &= k \times \left(\sum_{i=g}^{2g-1} \frac{1}{i + g\delta_c} \right) \end{aligned} \tag{15}$$

The transition from the 3_{rd} equality to the 4_{th} is based on $x = c - i$ and $y = g - i + c$. And the transition from the 5_{th} quality to the 6_{th} is based on $i = x + g$.

The value of expected sample size varies with δ_c. Because $0 \leq \delta_c < \frac{1}{g}$, we have:

$$\begin{aligned} Lower\_bound &= k \times \sum_{i=g+1}^{2g} \frac{1}{i} \\ Upper\_bound &= k \times \sum_{i=g}^{2g-1} \frac{1}{i} \end{aligned} \tag{16}$$

$Lower\_bound < Expected\ sample\ size \leq Upper\_bound$. □

Theorem 7 *The fluctuation range of sample size in SWTC-AG is* $k\frac{1}{2g}$.

Proof Based on the lower bound and the upper bound computed in Theorem 6, we can get the fluctuation range:

$$\begin{aligned} &Upper\_bound - Lower\_bound \\ &= k \times \left(\sum_{i=g}^{2g-1} \frac{1}{i} - \sum_{i=g+1}^{2g} \frac{1}{i} \right) \\ &= k \times \left(\frac{1}{g} - \frac{1}{2g} \right) \\ &= k \frac{1}{2g}. \end{aligned} \tag{17}$$

When we set $g = 10$, the value range of expected sample size of SWTC-AG is $(0.67k, 0.72k]$, with a fluctuation range of $0.05k$. We can see that the lower bound of expected sample size of SWTC-AG is larger than the upper bound of the baseline method, and the fluctuation range is 10 times smaller than the basic version of SWTC.

In Sect. 7, we analyze the sample size and accuracy of SWTC-AG, basic version of SWTC and the baseline method. The result in Fig. 7 confirms our analysis about expected sample size. The experimental results (like Figs. 10c and 11c) also show that SWTC-AG obtains higher accuracy than the basic version of SWTC in most circumstances.

6 Extension to other semantics

Weighted Counting: In weighted counting, each triangle is weighted with the multiplication of the frequencies of its three edges. If we treat f occurences of an edge as f distinct edges, a triangle with weight w can also be seen of w distinct triangles induced by different edge tuples. Therefore, when applying SWTC to weighted counting, we replace the hash functions $H(\cdot)$ and $G(\cdot)$, which are responsible for mapping an edge to substreams and generating priorities, with random functions. In other words, multiple occurences of an edge may be mapped to different substreams and get different priorities. Besides, we carry out weighted counting in the sample graph $\mathcal{G}_s$ and maintain the result in tc. The other operations are the same as the binary counting. The sample size analysis in Sect. 4.2, accuracy analysis in Sect. 4.4 and the fluctuation range analysis in Sect. 5.2.2 all apply to weighted counting. The only difference is that notations like $|W_{t_1}^{t_2}|$ and τ represent edge count or triangle count with duplication in weighted counting semantics. The baseline method can also be extended to weighted counting semantics in a similar manner.

Directed Triangle Counting: In prior works, triangle is defined without edge directions. When edge directions are considered, it is in fact a more general problem named motif counting [33,34]. There are seven kinds of triangle-shape

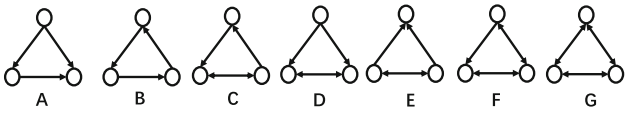


Fig. 6 Different Kinds of Directed Triangles

motifs with different direction constraints, as shown in Fig. 6. The first two kinds only include edges with one direction, and the last five kinds include bi-direction edge into consideration.[3] Our algorithm applies to all of them. The estimating process is the same as discussed above, except that we need to count the corresponding directed motif in the sample graph rather than un-directed triangles.

Local Counting: local counting means to count triangles including a specified node u. Local counting is usually used in spam detection. If one node is included into too many triangles, there is usually a spam activity or malicious attack around it [3]. Our algorithm also applies to local counting. When applying to local counting, we need to maintain one counter tc_u for each node u in the sample graph. We use tc_u to continuously monitor the count of triangles including u in the sample graph. When estimating, we scale up the local triangle count as $tc_u \times \frac{n(n-1)(n-2)}{m(m-1)(m-2)}$, where m is the number of edges in the sample graph and n is the estimated sliding window's cardinality. For nodes in the snapshot graph but not in the sample graph, the local count is estimated as 0.

7 Experimental evaluation

In this section, we experimentally evaluate our method over four real-world datasets. Details about the datasets, experiment settings and metrics are shown in Sect. 7.1, Sect. 7.2 and Sect. 7.3, respectively.

As discussed in Sect. 6, weighted counting is similar to binary counting if we view the duplicate copies of an edge as distinct edges. Therefore, we focus on binary counting from Sects. 7.4 to 7.11. In these subsections, we first test the independence of the hash functions we use with the topology of edges (7.4). Then we compare the Treap-based BPS sampling in [19] with the baseline method proposed in Sect. 3 (7.5). Recall that the baseline method is a combined method of Partition CT [10] and BPS [19], where edges are partitioned into k substreams and BPS sampling with bounded size 1 is carried out in each substream. As the result shows the baseline method is much superior, we mainly compare our SWTC with it in the following experiments. Then we evaluate the valid sample size (7.6) of SWTC, SWTC-AG and the baseline method, test their unbiasedness (7.7) and evaluate their accuracy (7.8). In Sect. 7.9, We compare SWTC with fixed probability sampling based method discussed in Sect. 3. In Sect. 7.10 and 7.11, we evaluate the performance of SWTC, SWTC-AG and the baseline method in semantics including directed graph (7.10) and local counting (7.11).

In Sect. 7.12, we evaluate the performance of our proposed method in weighted counting semantics. We also evaluate two most recent fully dynamic algorithms WRS [22] and ThinkD [18] in sliding window model by storing the entire sliding window. Notice that prior work for fully dynamic model only applies to weighted counting semantics. Therefore we only compare with them in Sect. 7.12. In Sect. 7.13, we further evaluate the processing speed of different versions of SWTC and show the effect of vision counting.

Besides, we also carry out experiments to evaluation the influence of duplication ratio, namely the percentage of duplicate edges in the sliding window. The experimental result is shown in the appendix which can be found one line [23] due to space limitation.

Experiments are implemented in a PC server with dual 18-core CPUs (Intel Xeon CPU E5-2697 v4@2.3 G HZ, 2 threads per core) and 192 G memory, running CentOS. All codes are written in C++ and compiled with GCC 4.8.5.

7.1 Datasets

Four real-world datasets are used in experiments. In order to make the window length intuitive, we divide the total time span of each dataset with the number of edges in it to get the average time span between two edge arrivals, and use this average time span as the unit of the window length. The frontier of the sliding window, T, is set to the timestamp of the last edge that the algorithm has processed. The datasets are as follows:

(1)StackOverflow:[4] This is a dataset of interactions on the stack exchange website Stack Overflow. Nodes are users and edges represent user interactions. There are 63,497,050 edges with duplication and 2,601,977 nodes.

(2)Yahoo:[5] This is a network flow dataset collected from three border routers by Yahoo. We use IP addresses as nodes and communications among them as edges. It includes 561,754,369 edges and 33,635,059 nodes.

(3)WikiTalk:[6] This is the communication network of the English Wikipedia. Nodes represent users, and an edge from user A to user B denotes that user A wrote a message on the talk page of user B at a certain timestamp. It includes 24,981,163 edges and 2,987,535 nodes.

[3] notice that a bi-direction edge means one edge marked as bi-direction rather than 2 edges with reverse directions. In the latter case the motif include 4 edges rather than 3.

[4] http://snap.stanford.edu/data/sx-stackoverflow.html

[5] https://webscope.sandbox.yahoo.com/catalog.php?datatype=g

[6] http://konect.cc/networks/wiki_talk_en/

Table 3 Average Triangle Count in Different Datasets

	StackOverflow	Yahoo	WikiTalk	Actor
N	4.5 M	35 M	3 M	4.5 M
τ	2.16 M	0.61 M	2.63 M	11.4 M

(4)Actor:[7] This is a dataset describing cooperation of actors. Nodes are actors and edges represent films in which they cooperate. There are totally 33,115,812 edges with duplication and 382,219 nodes.

The last dataset Actor does not have timestamps. Therefore we randomly generate timestamps for it. We shuffle the dataset three times and compute the average performance whenever we use it. For the StackOverflow, Yahoo and WikiTalk, as they have original timestamps, we sort the dataset by these timestamps.

As the accuracy of the algorithm is related with the number of triangles in the snapshot graph, we list the average number of triangles τ in the sliding window for each dataset in Table 3. The corresponding window length N is also listed in the table.

7.2 Experiment settings

The number of substreams, denoted with k, decides the memory used in both SWTC and the baseline method. In applications, it is set according to the available memory. But it should be noted that as shown in Sect. 4.4, too small sample size will bring a large variance. We define the ratio of k against the window length N as *sample rate*, where the window length uses average time span as unit. As will be shown in Fig. 11, we vary the sample rate to carry out experiments, and results show that we can get a promising accuracy when the sample rate is larger than 4%. Further growing sample size brings relatively slow increment on accuracy. Therefore, we suggest k to be set $4\% \sim 6\%$ of the window length, if the memory is enough. We also use this setting in our experiments, and two methods (SWTC and baseline) have the same sample rate and the same memory usage.

In each experiment, we carry out SWTC and the baseline method five times with different hash functions, and use the average performance as experimental result. The hash functions used in SWTC and the baseline method include BobHash [35], MurmurHash [36], RSHash [37], APHash [38] and so on. More hash functions can be found at [38]. The group number in SWTC-AG is set to 10.

We set a checkpoint whenever the window slides $\frac{1}{10}$ of the window length, namely when the maximum timestamp of the inserted edges increases by $\frac{1}{10}N$. We measure metrics at these checkpoints. When T is less than two times of the window length, we do not set any checkpoint, as there are not enough expired edges and both algorithms produce large but not representative sample sets. For Yahoo, in which the window length is very large and compute the accurate triangle count is too time consuming, we estimate the algorithm performance at first 50 checkpoints. For other datasets, we estimate the algorithm performance at all checkpoints.

[7] http://konect.cc/

7.3 Metrics

In the experiments we evaluate 5 metrics of the algorithms for global triangle counting: average valid sample size, percentage of valid sample, MAPE, max error and MSPE of triangle count, defined as follows:

Average Valid Sample Size: In both SWTC and the baseline method, the number of valid sample edges varies as the window slides. We measure the number of valid sample edges at each checkpoint, and compute the average value of all checkpoints to get the average valid sample size.

Percentage of Valid Sample: The ratio of the number of valid sample edges against the total number of substreams.

MAPE: At each checkpoint, we compute the accurate triangle count, denoted as τ, and the estimated triangle, $\hat{\tau}$. The Absolute Percentage Error (APE) is estimated as $|\frac{\hat{\tau}-\tau}{\tau}|$. We compute the average value of all checkpoints to get Mean Absolute Percentage Error (MAPE).

Max Error: At each checkpoint, we compute the Absolute Percentage Error (APE) of the triangle estimation. We use the maximum value of all checkpoints as max error.

MSPE: At each checkpoint, the Signed Percentage Error (SPE) is estimated as $\frac{\hat{\tau}-\tau}{\tau}$. We compute the average value of all checkpoints to get Mean Signed Percentage Error (MSPE).

For local counting, we use 2 metrics: average Pearson and average ϵ error.

Average Pearson: We organize the local counts of all nodes in the graph as a vector x, and the estimated counts are organized as a vector y with the same dimension. Pearson Correlation Coefficient is defined as $\rho = \frac{Cov(x,y)}{\sigma_x \sigma_y}$ where $Cov(x, y)$ means the covariance of vector x and y, and σ_x (or σ_y) means the standard deviation of x (or y). Pearson Correlation Coefficient is used to estimated the correlation between two vector. A high Pearson Correlation Coefficient reveals a close relation between x and y, and implies the high accuracy of the estimation result. We compute the Pearson Correlation Coefficient at each check point, and compute the average value as average Pearson.

Average ϵ error: ϵ error is defined as $\frac{1}{|V|}\sum_{u\in V}\frac{|\hat{\tau}(u)-\tau(u)|}{\tau(u)+1}$, where V is the node set in the graph. $\tau(u)$ is the local triangle count of node u, and $\hat{\tau}(u)$ is the estimated local count. We add $\tau(u)$ by 1 in case that $\tau(u) = 0$. We compute ϵ error at each

Table 4 Chi-squared Statistic of Different Hash Functions

	MurmurHash1	MurmurHash2	BobHash1	BobHash2	CRC32	RSHash	APHash
Substream Partition	78.1	80.2	90	81.9	80.4	79.8	98.9
Priority Computation	81.7	84	84.7	85	76.7	84	95.9

check point, and compute the average value as experiment result.

7.4 Independence test of hash functions

In this section, we first use chi-squared test to check the statistical independence between the hash functions and the topology of the edges. We use WikiTalk dataset and the window length is set to $3M$. In this test, we divide the edges in the graph into 10 bins according to the number of triangles they are involved in. Initially, we divide the value range of involved triangle numbers of edges into 10 equal-sized region, and each bin contains edges with involved triangle numbers in one region. Then we recursively merge the smallest bin with its neighbor and bisect the largest bin, until all bins contain more than 100 edges. We do this to avoid underemphasizing contribution of small bins to the chi-squared computation. We also divide the value range of the hash function into 10 equal-sized region. We compute the number of edges in the i_{th} bins with hash values in the j_{th} region, denote as a_{ij}. We also compute the expected size of a_{ij}, denoted as $\hat{a_{ij}}$, and $\hat{a_{ij}} = m_i \times \frac{n_j}{n}$. m_i is the number of edges in the i_{th} bin. n_j is the number of edges with hash value in the j_{th} region, and n is the total number of edges. This expected size is computed under the assumption that the hash values are independent with the involved triangle number of edges. The chi-squared statistic is computed as follows:

$$\chi = \sum_{i=0}^{9}\sum_{j=0}^{9} \frac{(a_{ij} - \hat{a_{ij}})^2}{\hat{a_{ij}}} \tag{18}$$

The hypothesis that the hash values are independent with the involved triangle numbers of edges holds if the chi-squared statistic is smaller than the critical value of a given significance level, where the significance level is usually taken as 0.05.

In the above exam, the chi-squared statistic has $(10-1) \times (10-1) = 81$ degrees of freedom, and the corresponding critical value is 103. In Table 4. We list the chi-squared statistic of all 7 hash functions we use, in both priority computation and substream partition (the substream number is set to $0.12M$). MurmurHash1 and MurmurHash2 denote MurmurHash with different seeds, similar to BobHash1 and BobHash2. From the table, we can see that all chi-squared statistic is smaller than 103, indicating independence of hash values and the involved triangle number of edges.

7.5 Comparison of the baseline method with treap-based BPS

In this section, we compare the baseline method with Treap-based BPS, in order to show why we choose the former as a competitor of SWTC. We use StackOverflow dataset, and set the window length to $4.5M$. We use the baseline method and Treap-based BPS with the same bounded size $k = 0.18M$ to continuously maintain samples. We compute the average valid sample size among all checkpoints, as well as memory usage and processing speed. The result is shown in Table 5. The unit of speed is million (edge) insertions per second (Mips). Note that we focus on the sampling procedure, and the cost of maintaining sample graphs and counting triangles are not included. From the table, we can see that both methods get similar valid sample size, but the baseline method uses less memory and has higher speed. As the baseline method is superior than treap-based BPS, we mainly compare SWTC with it in the following experiments.

7.6 Valid sample size

We conduct experiments on sample size in two ways. First, in order to confirm our mathematical analysis in Sect. 4.2, we build a dataset with steady throughput: We filter out duplicate edges in Actor and arrange the timestamps so that there are exactly one edge in each time unit. In such a dataset, the cardinality of a window is linear correlated with the window length. Note that this specialized dataset is only used in this experiment. We evaluate the percentage of valid sample of SWTC and the baseline in it. The result is shown in Fig. 7, where the window length is set to $4M$ and the x-axis denotes the total number of processed time units (i.e. average time span defined in Sect. 7.1). The sample rate is set to 4%. In Fig. 7, we can see that the baseline method always get a 56% percentage of valid sample. On the other hand, the percentage of valid sample in SWTC varies from 50% to 100% in a cycle. This conforms to our mathematical analysis in Theorem 3. Moreover, by using AG technique, the percentage of valid sample in SWTC-AG is stabilized to around 72%.

We also report the average valid sample size in Actor and Yahoo in Figs. 8a and 8b. For Actor, we fix the sample rate

Table 5 Comparison of the Baseline Method with Treap-based BPS

Method	Valid Sample Size	Memory Usage(MB)	Speed(Mips)
Baseline	108, 425	14.4	1.55
Treap-based BPS	100, 612	31	0.95

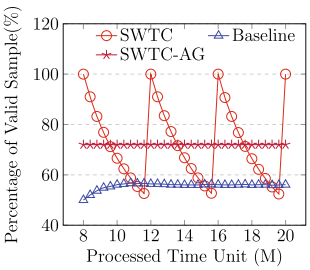


Fig. 7 Percentage of Valid Sample

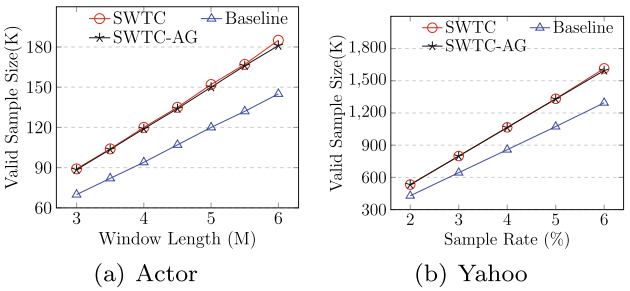


Fig. 8 Valid Sample Size of Different Algorithms

to be 4% and vary the window length. For Yahoo, we fix the window length to be $35M$ and vary the sample rate. We can see that the valid sample size rises with the increasing of the window length and the sample rate, as they both brings a larger k. SWTC always has a larger sample size than the baseline method, and SWTC-AG has the same average sample size as SWTC. The gap between SWTC and the baseline method varies since the cardinality of the sliding window varies with the throughput of the stream and the duplication ratio in the window. In average the sample size in $SWTC$ is 30% larger.

7.7 Unbiasedness of estimation

In this section, we experimentally check the unbiasedness of our triangle count estimation. We compute MSPE for the baseline method, SWTC and SWTC-AG. If the method is unbiased, MSPE should be near 0. We use WikiTalk to carry out experiment. The window length is set to $3M$, and we vary the sample rate. The result is shown in Fig. 9. From the figure, we can see that the MSPE of all 3 methods is in range $-2\% \sim 2\%$. As the sample rate grows, MSPE becomes closer to 0, indicating that all 3 methods are unbiased.

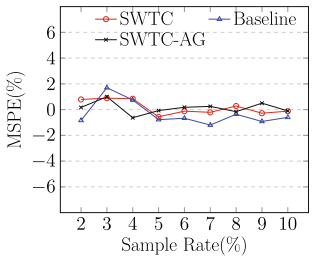


Fig. 9 MSPE in WikiTalk

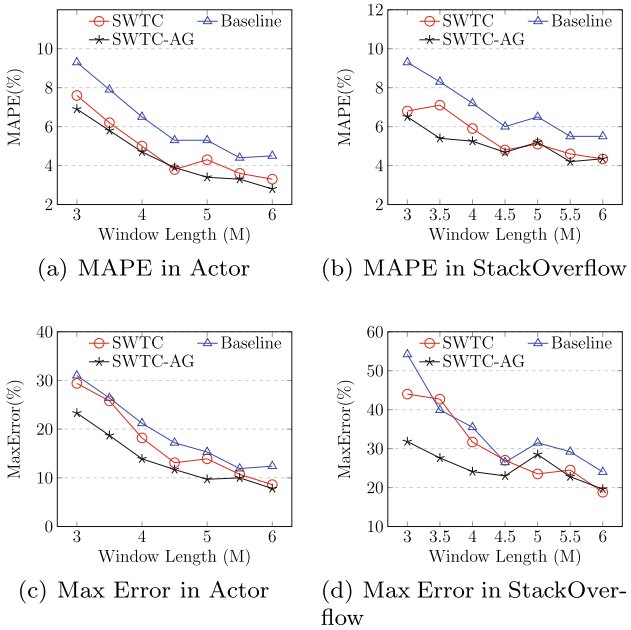


Fig. 10 Accuracy Varying with Window Length

7.8 Accuracy

Figure 10 shows the accuracy of SWTC and the baseline method varying with window length. The sample rate is set to 4%.

Figure 11 shows the accuracy of SWTC and the baseline method varying with sample rate. The window length is set to $3M$ for WikiTalk and $35M$ for Yahoo.

From the figures, we can see that SWTC has an MAPE up to 38% smaller than the baseline. And in all experiments, MAPE of SWTC is below 0.1. SWTC-AG has a lower MAPE than SWTC in most cases due to its stability. The max error of SWTC has no significant difference with the baseline method, as it produces periodic small sample graph and low accuracy. But in SWTC-AG, as the sample size is stabilized,

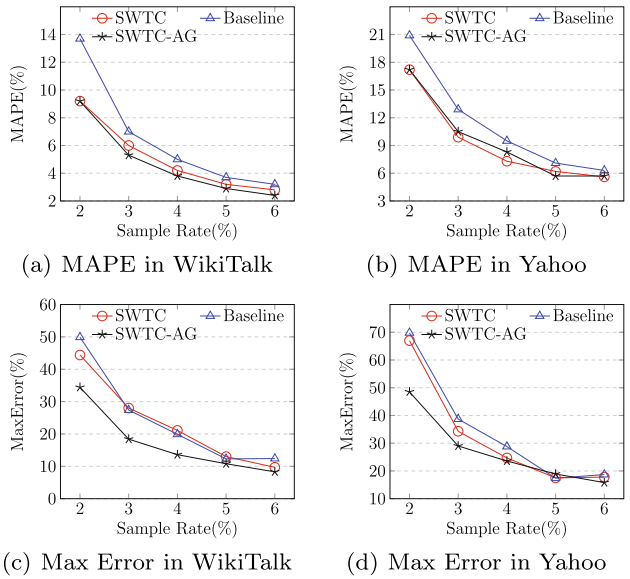


Fig. 11 Accuracy Varying with Sample Rate

the accuracy also becomes more reliable. The max error of SWTC-AG is up to 34% smaller than the baseline method.

Figure 10 also shows that as the window length grows, the estimation error of algorithms decreases, because when there are more triangles in the window, the influence of randomicity decreases and the estimated result becomes stable and accurate.

Figure 11 shows that the estimation error of algorithms decreases when the sample rate grows. This is intuitive, as a larger sample set produces a higher accuracy.

7.9 Comparison with fixed probability sampling

As discussed in Sect. 3, there is a naive sampling method, fixed probability sampling (fixed-p sampling for short), which can produce uniform sample in the sliding window but has no memory upper bound. Approximate triangle counting algorithm can also be built with fixed-p sampling. We use fixed-p sampling to maintain a sample edge set and build a sample graph with these edges. We monitor the triangle count tc in the sample graph and estimate the triangle count in the snapshot graph as $\frac{tc}{p^3}$.

In this Section. We compare SWTC with fixed-p sampling. The dataset used is WikiTalk, and the window length is set to $3M$. The experimental results, including MAPE and Max Error of triangle estimation, are shown in Table 6. We set the sample rate of SWTC and SWTC-AG to 4%. In other words both of them have $0.12M$ substreams. The memory usage of fixed-p sampling varies with the streaming graph throughput and has no upper bound. Therefore, we scan the entire dataset in advance, and find the maximum distinct edge number that will appear in the sliding window. Based on it,

Table 6 Comparison with Fixed Probability Sampling

Algorithm	MAPE (%)	Max Error (%)
SWTC	4.2	21.2
SWTC-AG	3.8	13.6
Fixed-p (same memory usage)	6.2	23.2
Fixed-p (same stored edge number)	3.7	12.5

we compute the sample probability p of fixed-p sampling, so that it has the same memory upper bound as SWTC. The experimental results of fixed-p sampling with such setting are shown in row 3 of Table 6. Experimental results show that both SWTC and SWTC-AG have higher accuracy than fixed-p sampling. The preserved memory of fixed-p sampling, which is set according to the throughput of peak time, is not fully used in ordinary times. On the other hand, SWTC and SWTC-AG can always make full use the preserved memory.

We also use another parameter setting (row 4 of Table 6). It allows fixed-p sampling to store at most $240k$ edges, i.e., storing the same number of edges as SWTC, because SWTC stores two edges in each substream. However, with such setting, the maximum sample graph size in fixed-p method is 2 times of SWTC. The memory for maintaining the sample graph, including the node table and the neighbor lists, makes the memory upper bound of fixed-p much higher than SWTC. Experimental results show that even with much higher memory preserved, the accuracy of fixed-p sampling is still close to that of SWTC-AG. Furthermore, it is difficult to predict the exact throughput in real world applications. Therefore fixed-p sampling can hardly achieve the accuracy in this experiment.

7.10 Counting directed triangles

In this section, we evaluate the performance of SWTC in directed triangle counting. We use StackOverflow to carry out experiment. There are no bi-direction edges in this dataset, but each edge is tagged with a label. There are totally 3 labels, corresponding to 3 kinds of interactions. We use edges with one label as bi-direction edges. The MAPE and MaxError of 7 types of triangles are shown in Figs. 12 and 13. We fix the window length to $4.5M$ and the sample rate to 6%. Triangles of type G are very rare, with an average number of only 164, 891, and the average numbers of the other 6 types are at least 409, 522. Therefore the error in estimation of this type is larger than others. We also show the result with sample rate 12% for type G. We can see that with a larger sample rate, the error is significantly decreased. Besides, we can also see that in all types, SWTC-AG always has the best accuracy.

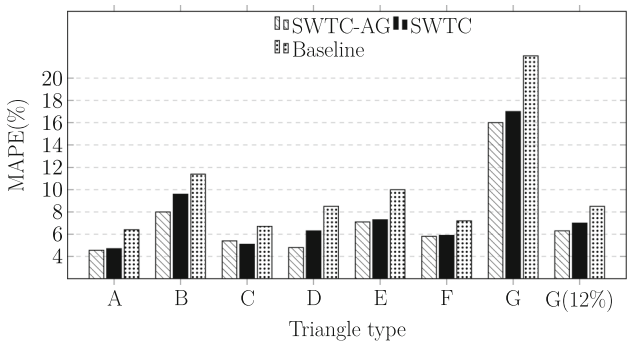


Fig. 12 MAPE in Directed Triangle Counting

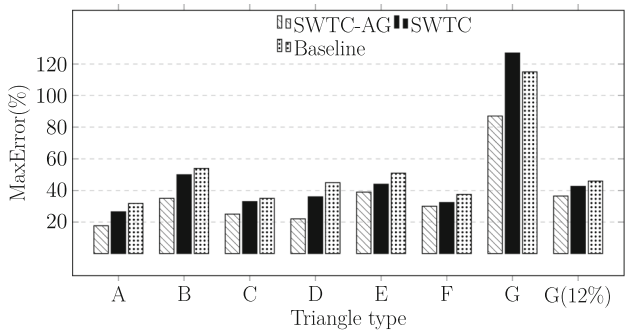
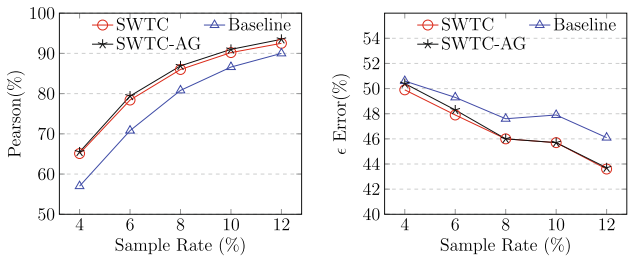


Fig. 13 Max Error in Directed Triangle Counting



(a) Average Pearson (b) Average ϵ Error

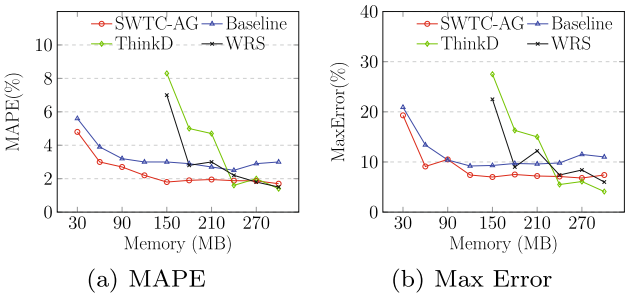
Fig. 14 Accuracy of local counting

7.11 Local counting

In this section, we evaluate the performance of SWTC in local counting. We use StackOverflow to carry out experiment. We fix the window length to $4.5M$ and change the sample rate, the result is shown in Fig. 14. From the figure, we can see that SWTC and SWTC-AG always have higher average Pearson and lower average ϵ error than the baseline method, implying higher accuracy.

7.12 Weighted counting

We carry out an experiment on weighted counting with StackOverflow dataset, with window length set to $4.5M$. Besides comparing with the baseline, we also compare our method with 2 prior algorithms in fully dynamic stream model,



(a) MAPE (b) Max Error

Fig. 15 Accuracy of Weighted Counting

WRS[22] and ThinkD [18]. In order to make the figure clear, we only show AG version of SWTC here. The MAPE and max error are shown in Fig. 15, where the x axis is the memory usage. As discussed in Sect. 3, WRS and ThinkD need to store the entire sliding window to work. We keep tracking the number of edges as the window slides, and find that the maximum number of edges in the sliding window is $5.4M$. Therefore, we reserve space for storing $5.4M$ edges for WRS and ThinkD. Each edge has two 4-byte node IDs and one 8-byte timestamp. As the edges are organized as a linked list, an additional pointer is needed by each edge. Therefore 24 bytes are needed for each edge in the sliding window. In total, WRS and ThinkD need at least $129.6MB$ memory to start to work. Therefore, in the figure we begin to present their accuracy at $150MB$. We can see that they begin to have MAPE lower than 4% only when the memory is larger than $210MB$. On the other hand, our algorithms get same accuracy with only $60MB$ memory. In other words, our algorithms achieve competitive performance with much less space. Besides, the memory used by WRS and ThinkD is unbounded in real world applications, because the number of edges in the sliding window varies with the throughput, and they need to store all the edges to work. The result in Fig. 15 also shows that in weighted counting, SWTC-AG still has a higher accuracy than the baseline.

7.13 Processing speed

In this section, we compare the speed of different versions of SWTC and the baseline method. We use Actor dataset in this experiment, and set the window length to be $4.5M$. The sample rate is set to 4%, and the triangles are count in binary counting semantics. We calculate average processing speed of the algorithms in each batch with $45K$ time units, and draw the curve of processing speed varying with total number of processed time units. The result is shown in Fig. 16. The measurement of speed is million insertions per second (Mips). The figure shows the change of speed at a landmark. We can see that at the landmark, the speed of SWTC suffers from a sharp decrease, because at the landmark, SWTC need a period to add new valid samples and count the triangles. On

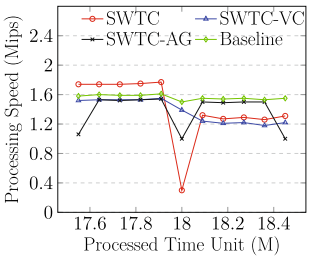


Fig. 16 Processing Speed of Different Algorithms

the other hand, in SWTC-AG, there is a small computation peak in every $0.45M$ time units, namely $\frac{1}{10}$ of the sliding window. As SWTC-AG carries out slice switch for a group of $\frac{k}{10}$ substreams at these time points. With vision counting technique (SWTC-VC), the processing speed is completely smoothed. Notice that vision counting technique can also be implemented with SWTC-AG, and the speed is the same as SWTC-VC. The processing speed of the baseline method is also shown in the figure. Because the baseline method has a relatively small valid sample size, it processing speed is slightly higher. But as shown in the experiments above, its accuracy is also poorer.

8 Related work

8.1 Prior arts in triangle counting

The problem of counting triangles in large graphs has been researched for decades. It can be divided into 2 problems, counting global triangles (triangles in the entire graph) and local triangles (triangles which include a certain node). Compared to algorithms [39–43] which exactly count the number of triangles in large graphs, approximately counting algorithms [44–47] are much faster and consume less memory. Recent work in approximation triangle counting includes the algorithm of Pavan et al. [8] which uses a neighborhood sampling to sample and count triangles, and the algorithm of Jha et al. [48] which samples wedges to estimate triangle count. Tsourakakis et at. [49] proposes to sample each edge with a fixed-probability and their algorithm can be directly used in streaming graphs. Ahmed et al. [9] presents a general edge sampling framework for graph statistics estimation including the triangle count.

The above algorithms do not consider edge duplication. TRIÉST [17] uses reservoir sampling method [50] and has a fixed sample size. It supports edge deletions in fully dynamic streaming graphs with a technique named random pairing [51]. It also supports weighted counting with edge duplication. But it can neither support binary counting, nor support sliding window model. PartitionCT [10] estimates triangle counts in streaming graphs by filtering duplicate edges and counting binary triangles. It divides the streaming graph into substreams with a hash function. In each substream, it performs a priority sampling with another hash function. PartitionCT also solves the problem of cardinality estimation with the help of prior works including [20,21]. However it cannot be directly used in sliding windows, as it does not support edge expiration. Subsequent work includes [16,18,22], but they either do not support deletion or do not support binary counting semantics. Besides, as discussed in Sect. 3, even algorithms supporting deletions in fully-dynamic model cannot support the expiration in sliding windows. To the best of our knowledge, no algorithm has addressed the problem of triangle count estimation in streaming graphs with sliding window and edge duplication using bounded-size memory.

8.2 Sampling algorithms in sliding windows

It has been proved impossible to maintain a fixed-size sample with bounded memory over a time-based sliding window [19]. Therefore most related works sample data streams in sliding windows with unbounded memory like [24,52,53]. We find them not suitable for sampling in the triangle counting problem for 2 reasons. First, unbounded memory usage makes it difficult to reserve enough memory in advance. Second, most of them need to compute sample set upon query, but we hope to achieve continuous query in triangle counting. BPS algorithm [19] suits the need of triangle counting most, because It has a strict upper bound of the memory usage and achieves continuous query. But its sample set has an uncertain size as a cost. In the baseline method, we use a simplified version of BPS, where we only need to maintain at most one sample in the sliding window. We combine such BPS algorithm with PartitionCT to achieve sampling with bounded size k.

8.3 HyperLogLog algorithm

The HyperLogLog algorithm [20] is proposed by Flajolet et al. It is a highly compact algorithm to estimate the number of distinct items (i.e., cardinality) in a set.

It uses a sketch with m counters $c_1.c_2.....c_m$ and 2 hash functions. The counters are all 0 initially. One hash function is $g(\cdot)$ which uniformly maps the input to integers in range $1 \sim m$. The other hash function is $y(\cdot)$ whose output has a $Geometric(\frac{1}{2})$ distribution. In other words, the probability that $y(e) = x$ is $\frac{1}{2^x}$ for $x = 1, 2, 3....$ When inserting an item e, it first uses $g(\cdot)$ to map it to one of the m counters $c_i (1 \leq i \leq m)$. Then it computes $y(e)$ and set $c_i = y(e)$ if $y(e) > c_i$. After inserting all the items, apparently a counter will get higher value when more distinct items are mapped to it, and duplicate items will not influence the sketch, as the same item will always get the same value in $y(\cdot)$ and $g(\cdot)$.

The cardinality is estimated as $\frac{\alpha_m m^2}{\sum_{i=1}^{m} 2^{-c_i}}$, and α_m is used to correct the bias which is $\alpha_m = 0.7213/(1 + 1.079/m)$ for $m > 128$. The error percentage is about $\frac{1.04}{\sqrt{m}}$.

9 Conclusion

Triangle counting in real-world streaming graphs with edge duplication and sliding windows has been an unsolved problem. In this paper, we propose an algorithm named $SWTC$. It uses an original sampling strategy to retain a bounded-size sample of the snapshot graph in the sliding window. With this sample, we can continuously monitor the triangle count in the sliding window with bounded memory usage. We further propose two optimization techniques to improve the performance of SWTC: vision counting and asynchronous grouping. Mathematical analysis and experiments show that SWTC generates a larger sample set and has higher accuracy than the baseline method, which is a combination of several existing algorithms, under the same memory consumption.

10 Appendix

10.1 Influence of duplication ratio

In order to evaluate the influence of duplication ratio of the streaming graph, we use a synthetic dataset FF to carry out experiments. This dataset is generated by Fire-Forest model [54]. It includes 18, 311, 282 edges and $1M$ nodes. There are originally no duplicate edges. We generate edge frequencies for it with power-law distribution and vary the duplication ratio to carry out experiments. The timestamps are randomly generated in this dataset. We formally define the duplication ratio as $\frac{total\ number\ of\ edges}{number\ of\ distinct\ edges} - 1$. The window length is set to be $3M$ and the sample rate is set to be 4%. We use binary counting semantics in this experiment. The memory usage and the valid sample size does not change with the duplication ratio. The experimental result in Fig. 17 shows that MAPE and max error decreases with the increment of duplication ratio. Because with more duplicate edges, the number of distinct edges in the sliding window decreases, and the sample size becomes relatively large.

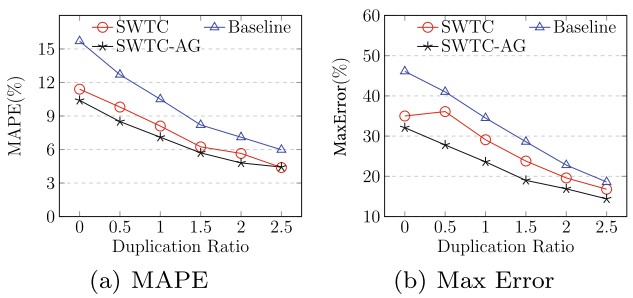


Fig. 17 Accuracy Varying with Duplication Ratio

References

1. Berry, J.W., Hendrickson, B., LaViolette, R.A., Phillips, C.A.: Tolerating the community detection resolution limit with edge weighting. Phys. Rev. E **83**(5), 056119 (2011)
2. Jean-Pierre, E., Elisha, Moses: Curvature of co-links uncovers hidden thematic layers in the world wide web. Proc. Natl. Acad. Sci. USA **99**(9), 5825–5829 (2002)
3. Becchetti, L., Boldi, Paolo, Castillo, C., Gionis, A.: Efficient algorithms for large-scale local triangle counting. ACM Trans. Know. Dis. Data (TKDD) **4**(3), 13 (2010)
4. Milo, R., Shen-Orr, Shai, Itzkovitz, S., Kashtan, N., Chklovskii, D., Alon, Uri: Network motifs: simple building blocks of complex networks. Science **298**(5594), 824–827 (2002)
5. Kang, U., Meeder, B., Papalexakis, Evangelos E., Faloutsos, C.: Heigen: Spectral analysis for billion-scale graphs. IEEE Trans. Know. Data Eng. **26**(2), 350–362 (2012)
6. Yang, Z., Wilson, C., Wang, X., Gao, T., Zhao, B.Y., Dai, Y.: Uncovering social network sybils in the wild. ACM Trans. Know. Dis. Data (TKDD) **8**(1), 1–29 (2014)
7. Li, Z., Yunting, Lu., Zhang, W.-P., Li, R.-H., Guo, J., Huang, X., Mao, Rui: Discovering hierarchical subgraphs of k-core-truss. Data Sci. Eng. **3**(2), 136–149 (2018)
8. Pavan, A., Tangwongsan, K., Tirthapura, S., Kun Lung, Wu.: Counting and sampling triangles from a graph stream. Proc. Vldb Endowment **6**(14), 1870–1881 (2013)
9. Ahmed, N. K., Duffield, N., Neville, J., & Kompella, R.: Graph sample and hold: A framework for big-graph analytics. In: Acm Sigkdd International Conference on Knowledge Discovery & Data Mining, (2014)
10. Wang, P., Qi, Y., Sun, Yu., Zhang, X., Guan, X.: Approximately counting triangles in large graph streams including edge duplicates with a fixed memory usage. Proc. Vldb Endowment **11**(2), 162–175 (2017)
11. Boykin, P.O., Roychowdhury, Vwani P.: Leveraging social networks to fight spam. Computer **38**(4), 61–68 (2005)
12. Datar, M., Gionis, A., Indyk, P., Motwani, R.: Maintaining stream statistics over sliding windows. Siam J. Comput. **31**(6), 1794–1813 (2002)
13. Li, Y., Zou, L., Özsu, M.T., Dongyan, Z.: Time constrained continuous subgraph search over streaming graphs. In 2019 IEEE 35th International Conference on Data Engineering (ICDE), pages 1082–1093. IEEE, (2019)
14. Crouch, M.S., McGregor, A., Stubbs, D.: Dynamic graphs in the sliding-window model. In European Symposium on Algorithms, pages 337–348. Springer, (2013)
15. Qiu, X., Cen, W., Qian, Z., Peng, Y., Zhang, Y., Lin, X., Zhou, J.: Real-time constrained cycle detection in large dynamic graphs. Proc. VLDB Endowment **11**(12), 1876–1888 (2018)
16. Jung, M., Lim, Y., Lee, S., Kang, U.: Furl: fixed-memory and uncertainty reducing local triangle counting for multigraph streams. Data Min. Know. Dis. **33**(5), 1225–1253 (2019)
17. De Stefani, Lorenzo, Epasto, Alessandro, Riondato, Matteo, Upfal, Eli: Triest: counting local and global triangles in fully dynamic streams with fixed memory size. ACM Trans. Know. Dis. Data (TKDD) **11**(4), 1–50 (2017)
18. Shin, Kijung, Sejoon, Oh., Kim, Jisu, Hooi, Bryan, Faloutsos, Christos: Fast, accurate and provable triangle counting in fully

dynamic graph streams. ACM Trans. Know. Dis. Data (TKDD) **14**(2), 1–39 (2020)
19. Gemulla, R., Lehner, W.: Sampling time-based sliding windows in bounded space. In: Acm Sigmod International Conference on Management of Data, (2008)
20. Flajolet, P., Fusy, É., Gandouet, O., Meunier, F.: Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In Discrete Mathematics and Theoretical Computer Science, pages 137–156. Discrete Mathematics and Theoretical Computer Science, (2007)
21. Ting, D.: Streamed approximate counting of distinct elements: Beating optimal batch methods. In Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 442–451 (2014)
22. Dongjin, L., Kijung, S., Christos, F.: Temporal locality-aware sampling for accurate triangle counting in real graph streams. The VLDB Journal, pages 1–25 (2020)
23. Source code of swtc and the baseline method. https://github.com/StreamingTriangleCounting/TriangleCounting.git
24. Brian, B., Mayur, D., Rajeev, M.: Sampling from a moving window over streaming data. In Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, pages 633–634. Society for Industrial and Applied Mathematics (2002)
25. Seidel, R., Aragon, Cecilia R.: Randomized search trees. Algorithmica **16**(4), 464–497 (1996)
26. Kac, Mark: Statistical Independence in Probability. Courier Dover Publications, Analysis and Number, New York (2018)
27. Duffield, N.G., Grossglauser, M.: Trajectory sampling for direct traffic observation. IEEE/ACM Trans. Netw. **9**(3), 280–292 (2001)
28. Duffield, Nick: Sampling for passive internet measurement: A review. Stat. Sci. **19**(3), 472–498 (2004)
29. Aggarwal, C.C., Yuchen, Z., Yu, P.S.: Outlier detection in graph streams. In 2011 IEEE 27th international conference on data engineering, pages 399–409. IEEE, (2011)
30. Ashish, T., Sen, S.J., Namit, J., Zheng, S., Prasad, C., Ning, Z., Suresh, A., Hao, L., Raghotham, M.: Hive-a petabyte scale data warehouse using hadoop. In: 2010 IEEE 26th international conference on data engineering (ICDE 2010), pages 996–1005. IEEE (2010)
31. Maurizio, M., Saverio, N., Duffield, N.G.: A comparative experimental study of hash functions applied to packet sampling. In: Proc. of International Teletraffic Congress (ITC) (2005)
32. Slota, G.M., Madduri, Kamesh: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13**(7), 422–426 (1970)
33. Slota, G.M., Kamesh, M.: Complex network analysis using parallel approximate motif counting. In 2014 IEEE 28th International Parallel and Distributed Processing Symposium, pages 405–414. IEEE, (2014)
34. Bressan, M., Chierichetti, F., Kumar, R., Leucci, S., Panconesi, A.: Motif counting beyond five nodes. ACM Trans. Know. Dis. Data (TKDD) **12**(4), 1–25 (2018)
35. Bobhash function. http://burtleburtle.net/bob/hash/doobs.html
36. Murmurhash function. Published by Austin Appleby at https://github.com/aappleby/smhasher
37. Sedgewick, R.:. Algorithms in c. Pearson Education, (2001)
38. Aphash and collection of other hash functions. http://www.partow.net/programming/hashfunctions/#RSHashFunction
39. Alon, N., Yuster, R., Zwick, U.: Finding and counting given length cycles. Algorithmica **17**(3), 209–223 (1997)
40. Shaikh, A., Maleq, K., Madhav, M.: Patric: a parallel algorithm for counting triangles in massive networks. In Acm International Conference on Information & Knowledge Management, (2013)
41. Xiaocheng, H., Yufei, T., Chung, C.W.: Massive graph triangulation. In: Acm Sigmod International Conference on Management of Data, (2013)
42. Jinha, K., Wook, S.H., Sangyeon, L., Kyungyeol, P., Yu, H.: Opt:a new framework for overlapped and parallel triangulation in large-scale graphs. (2014)
43. Ha-Myung, P., Sung-Hyon, M., Kang, U.: Pte: Enumerating trillion triangles on distributed systems. In Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 1115–1124 (2016)
44. Bar-Yossef, Z., Kumar, R., Sivakumar, D.: Reductions in streaming algorithms, with an application to counting triangles in graphs. In: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms, pages 623–632. Society for Industrial and Applied Mathematics (2002)
45. Buriol, S.L., Frahling, G., Leonardi, S., Marchetti-Spaccamela, A., Sohler, C.: Counting triangles in data streams. In Acm Sigmod-sigact-sigart Symposium on Principles of Database Systems, (2006)
46. Jowhari, H., Ghodsi, M.: New streaming algorithms for counting triangles in graphs. In International Computing and Combinatorics Conference, pages 710–716. Springer, (2005)
47. Lim, Y., Kang, U.: Mascot: Memory-efficient and accurate sampling for counting local triangles in graph streams. In: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pages 685–694. ACM (2015)
48. Jha, M., Seshadhri, C., Pinar, A.: A space efficient streaming algorithm for triangle counting using the birthday paradox. (2013)
49. Tsourakakis, C.E., Kang, U., Miller, G.L., Faloutsos, C.: Doulion: counting triangles in massive graphs with a coin. In Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, pages 837–846, (2009)
50. Vitter, Jeffrey S.: Random sampling with a reservoir. ACM Trans. Math. Soft. (TOMS) **11**(1), 37–57 (1985)
51. Gemulla, R., Lehner, Wolfgang, Haas, P.J.: Maintaining bounded-size sample synopses of evolving datasets. The VLDB J. **17**(2), 173–201 (2008)
52. Braverman, V., Ostrovsky, R., Zaniolo, C.: Optimal sampling from sliding windows. In Twenty-eighth Acm Sigmod-sigact-sigart Symposium on Principles of Database Systems, (2009)
53. Cormode, G., Muthukrishnan, S., Yi, K., Zhang, Q.: Optimal sampling from distributed streams. In Proceedings of the twenty-ninth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pages 77–86, (2010)
54. Leskovec, J., Kleinberg, J., Faloutsos, C.: Graph evolution: Densification and shrinking diameters. ACM Trans. Know. Dis. Data (TKDD), 1(1):2–es, (2007)