Sliding Window-based Approximate Triangle Counting over Streaming Graphs with Duplicate Edges

Xiangyang Gou*
gxy1995@pku.edu.cn
Peking University

Lei Zou†
zoulei@pku.edu.cn
Peking University

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 edge duplication and expiration 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 with edge duplication. In SWTC, we propose a fixed-length slicing strategy that addresses both unbiased sampling and cardinality estimation issues with a bounded memory usage. We theoretically prove the superiority of our method in the sample graph size and estimation accuracy under given memory upper bound. Extensive experiments over large real streaming graphs confirm that our approach can obtain larger sample graphs and more accurate estimation value on counting triangle numbers compared with the baseline method.

ACM Reference Format:
Xiangyang Gou and Lei Zou. 2021. Sliding Window-based Approximate Triangle Counting over Streaming Graphs with Duplicate Edges. In *SIGMOD '21: Proceeding of the 2021 ACM SIGMOD International Conference on Management of Data June 20–25, 2021, Xi'an, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/xxxxxxx.xxxxxxx

1 INTRODUCTION

Graphs are an omnipresent form representing large-scale entities and their relations in various fields, like biochemistry, social networks, knowledge graphs and so on. Various kinds of data analysis can be implemented upon a graph, and 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 over large streaming graphs using sampling techniques.

Although several algorithms have been proposed in the literature, most of them consider the problem in a very ideal situation. Existing work often assumes that there are no duplicate edges [8, 9], or no edges will expire. Recent work like PartitionCT [10] considers duplicate edges, but still fails to support edge expiration. In real-world applications, the situation is more complex. Streaming graphs usually have duplicate edges, and edge expiration is included due to the need of timeliness. Here we give a motivation example. In social networks, user communications form a streaming graph. Raw communication logs have *duplicate edges*, as each pair of users may communicate multiple times. Spam and topics in social networks can be detected with *triangle counting* [3, 11]. In order to detect new topics or spam in real time, 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 is widely used in streaming graph algorithms and systems [13–15]. Therefore, a sliding window-based continuous triangle counting algorithm with edge duplication is desired. That is the focus of our work.

Generally speaking, there are two different semantics dealing with duplicate edges, *binary counting* and *weighted counting* (see Definitions 2.4). 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 applies to both weighted counting and binary counting. For the simplicity of the presentation, we only study *binary counting* when presenting our algorithm and discuss how to extend our method to support *weighted counting* with minor extensions in Section 4.6.

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 times. The risk of such memory constraint violation is high in real-world

*Xiangyang Gou is with Peking University and National Engineering Laboratory for Big Data Analysis Technology and Application (PKU), China

†Corresponding author Lei Zou is with Peking University and National Engineering Laboratory for Big Data Analysis Technology and Application (PKU), China and Beijing Institute of Big Data Research, China

SIGMOD '21, June 20–25, 2021, Xi'an China

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/10.1145/xxxxxxx.xxxxxxx

streaming graphs, as the throughput at peak times may be multiple times higher than ordinary days and hard to predict. Therefore, we need an algorithm with bounded memory usage in applications. As it has been proven impossible to maintain a fixed-size sample in sliding windows with bounded memory usage [19], 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 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 can not notice expiration of unsampled edges.

1.1 Our Solution

In order to approximately count triangles in sliding window-based streaming graphs with edge duplication, 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 a challenging task. 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 [1]. 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. When duplication is included, problem becomes more complicated, as we have to filter out the duplicated edges under binary counting semantics.

In order to address this issue, we begin with a baseline that combines the structure of PartitionCT [10] with BPS algorithm [19], and uses hash-based sample to deal with duplication. However, expected sample graph size in this baseline is rather small compared to its memory usage. Therefore, we further propose an optimized uniform 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 Section 4.2. Also, mathematical analysis in Section 4.5 and extensive experiments in Section 5 confirm that our larger sample graphs can decrease the mean absolute percentage error (MAPE) of triangle count estimation by 62% (Figure 6(d)) compared with the baseline.

Besides maintaining the unbiased bounded-size sample, we need to continuously monitor the number of edges in the sliding window [2]. 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 Section 4.1 can address both unbiased sampling and cardinality estimation together. Based on it, we propose a continuous cardinality estimation algorithm in Section 4.4.

Although we address both unbiased sampling and cardinality estimation using a uniform strategy—*the fixed-length slicing*, when the sliding window meets each "splitting point" (called *landmark*) of the slices, there is a dramatic size increment in the sample graph, resulting in a computation peak time. To solve this problem, we further propose an optimized technique named "vision counting", which spreads the computation cost at these peaks to the entire procedure of the window sliding, and evades the congestion.

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 Section 6. Generally, our method is the only work that addresses both edge duplication and expiration. More importantly, our method can support both *binary counting* and *weighted counting* semantics. In summary, we made the following contributions.

(1) We propose the problem of approximately counting triangles in streaming graphs with sliding windows and duplicate edges.
(2) In order to solve the above problem, we propose a fixed-length slicing strategy that addresses both unbiased sampling and cardinality estimation. It can be applied to both binary counting and weighted counting. We theoretically prove the superiority of our method in the sample graph size and estimation accuracy under given memory upper bound.
(3) To avoid high latency at computation peak times, we propose a technique named *vision counting* to spread the heavy computation workloads at peak times to the entire procedure of the window sliding.
(4) Extensive experiments over large streaming graphs confirm that our method outperforms the baseline solution in terms of sample size and estimation accuracy. We released all codes at Github [23].

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-Our Method	Sliding window	✓	Both

2 PROBLEM DEFINITION

In this section, we first formally define our problem.

Definition 2.1. ***Streaming Graph**: A streaming graph is an unbounded time evolving sequence of items $S = \{e_1, e_2, e_3.....e_n\}$, where each item $e_i = (\langle v_{i_1}, v_{i_2}\rangle, t(e_i))$ indicates an edge between nodes v_{i_1} and v_{i_2} arriving at time $t(e_i)$. This sequence continuously arrives from data sources like routers or monitors with high speed. An*

[1] Details will be discussed in Section 3
[2] 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

edge $\langle v_{i_1}, v_{i_2}\rangle$ 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 Section 4.6. Note that in the streaming graph model, due to the high speed and large volume of the stream, we assume 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 none) edges arriving at each time point.

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 time-based sliding window in the follows.

Definition 2.2. ***Sliding window**: A sliding window with window length N in a streaming graph 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^T_{T-N}.*

The window size N depends on applications, and the number of edges in the sliding window varies with the throughput of the stream. More generally, we use $W^{t_2}_{t_1}$ 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 2.3. ***Snapshot graph**: A* snapshot *graph at time T, denoted as G^T, is a graph induced by all the edges within the sliding window W^T_{T-N}.*

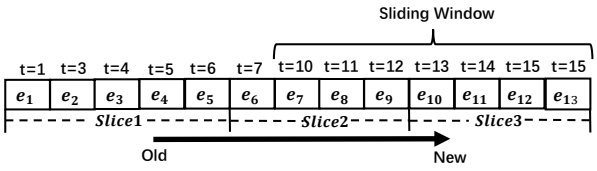


Figure 1: Streaming Graph and Sliding Window at $T = 15$

Example 1. *A streaming graph S with the sliding window is given in Figure 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^{15}_9, and there are 7 edges in it. The separation of slices is used in SWTC algorithm, and will be explained in Section 4.*

In this paper, we focus on *continuous triangle counting* in the sliding window model, which maintains the number of triangles in the current snapshot graph. There are duplicate edges in the snapshot graph, as an edge may have multiple copies with different timestamps. There are two kinds of semantics to deal with these duplicate edges [16], i.e., *binary counting* and *weighted counting*:

Table 2: Notation Table

Notation	Meaning
$S = \{e_1, e_2, ...e_n\}$	Streaming graph S
$t(e)$	Timestamp of an edge e
$W^{t_2}_{t_1}$	Set of edges e where $t_1 < t(e) \leqslant t_2$
$\lvert W^{t_2}_{t_1}\rvert$	Number of distinct edges in $W^{t_2}_{t_1}$
N	Length of the sliding window
W^T_{T-N}	Sliding window at time T with size N
G^T	Snapshot graph in the sliding window
G_s	Sample graph generated from G^T
ϵ	Sampled 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_{new}	The latest landmark before current time T
l_{old}	The second latest landmark before current time T
$\epsilon^{l_{new}}_{l_{old}}[i]$	The edge with the largest priority in the i_{th} substream in $W^{l_{new}}_{l_{old}}$
$\epsilon^{T}_{l_{new}}[i]$	The edge with the largest priority in the i_{th} substream in $W^{T}_{l_{new}}$

Definition 2.4. ***Binary** & **Weighted counting**: A triangle in a graph G is defined as a tuple of three edges $(\langle u,v\rangle, \langle u,w\rangle, \langle v,w\rangle)$, 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. The weighted counting returns the* sum *of all triangle weights.*

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 during sampling and estimating edge counts. The denotations used in this paper is presented in Table 2.

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 with bounded memory. It should be noted that algorithms for fully dynamic models such as [17, 18] cannot be used in sliding windows, since 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 in a sliding window, 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 Section 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 Section 3.2).

3.1 Background: Priority Sampling and BPS

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 bounded memory.[3] As a compromise, BPS maintains a bounded-size sample, which lays the foundation of our solution. For the simplicity of the presentation, we only introduce 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 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 Figure 2, assume that the window length is 6. The sampled 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 sampled 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 Figure 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 keep comparing new edges with it. ϵ is a *valid* sample only when $G(\epsilon) \geqslant G(\epsilon_{test})$. 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 Figure 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 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 sampled 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 Figure 2, e_1 double expires at $t = 12$. At this time, edges arriving from time 2 to 6 all expires. 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, BPS algorithm cannot get a valid sample in some periods. We call such period a *vacuum* period. In Figure 2, the vacuum period is from $t = 7$ to $t = 12$.

3.2 Structure of the Baseline Method

The original BPS algorithm uses a random function $G(\cdot)$ to generate priority. Considering two different semantics in streaming graphs with duplication, we have different function settings for $G(\cdot)$. In *binary counting*, we use a *hash* function to define the priority $G(\cdot)$ instead of random one. Because with random function, duplicated edges will get multiple priorities and a higher sampling probability, which leads to bias in binary counting. The hash function generates the same priority for a duplicated edge, no matter how many times the edge arrives. Therefore, it can derive a uniform sample in binary counting. In *weighted counting*, duplicate copies of an edge are seen as independent, and each copy get a chance to be sampled. In this case, we still use random function to define the priority $G(\cdot)$.

When extended to multiple samples, the original BPS algorithm uses a complicated data structure named treap to maintain the sampled edges and test edges, which is both memory and time consuming. We use the technique in PartitionCT to simplify the data structure. Assume that the sample size is upper bounded by k, where k is an user-specified parameter. We use a function $H(\cdot)$ to split a streaming graph into k substreams, like PartitionCT. This function is a hash function in binary counting, and a random function in weighted counting. In each substream, we use BPS algorithm to obtain at most one valid sampled edge. The framework of the baseline method is shown in Figure 3. Notice that only valid sampled edges are included in sample graph 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 sampled edge and the number of edges in sample graph G_s is k. Therefore, the memory upper bound in the baseline is able to hold a k-edges sample graph G_s. However, as each substream has 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 probability that the baseline approach can get a valid sampled edge in one substream is in range $[\frac{|W^T_{T-N}|}{|W^T_{T-2N}|}, 1 - \frac{|W^{T-N}_{T-2N}|}{|W^T_{T-3N}|}]$ (see Theorem 4.2 in Section 4.2)[4], where $|\cdot|$ denotes the number of edges in the window [5]. Assume that the streaming graph's throughput is steady, the valid sample probability is $[0.5, 0.66]$. It means that the sample graph size is between $0.5k$ and $0.66k$ edges. In expectation, only half of the memory will be efficiently used in the baseline. We can improve the sample 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.

[3] In page 3, Section 3.1 of [19].

[4] [19] only gives the lower bound, we further analyze the upper bound

[5] distinct count for binary counting, and count with duplication for weighted counting, see section 4.2 and section 4.6

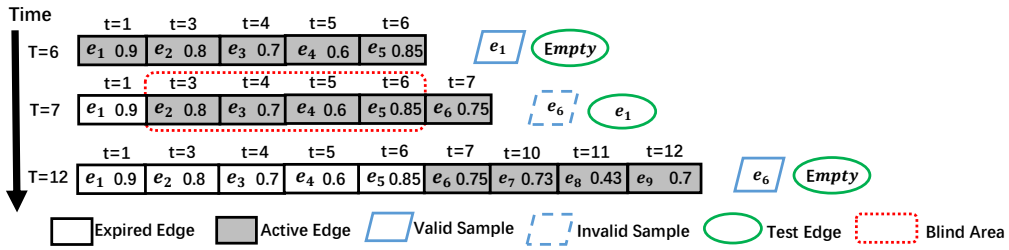


Figure 2: Example of BPS Sampling

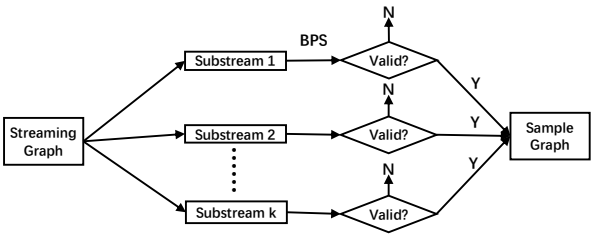


Figure 3: Framework of the Baseline Method

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 together with its optimization version in Section 4.1 and 4.3, respectively. In Section 4.2, we theoretically prove that SWTC gets a larger-size sample graph than the baseline method under the same memory consumption. In Section 4.4, we discuss how to continuously monitor $|W^T_{T-N}|$, namely the number of distinct edges in the sliding window, and estimate the binary triangle count in the sliding window. In Section 4.5, we theoretically analyze the accuracy of SWTC. For the simplicity of presentation, we only focus on binary counting until Section 4.6, in which we extend SWTC to weighted counting and directed graphs.

4.1 SWTC Sample Strategy

Sample 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^T_{T-N} overlaps with at most two slices, which are denoted as $W^{l_{new}}_{l_{old}}$ and $W^T_{l_{new}}$, where l_{new} and l_{old} are two landmarks (splitting point) and T is the current time point. An example is shown in Figure 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 $\epsilon^{t_2}_{t_1}[i]$ to represent the edge with the largest priority in a slice from t_1 to t_2 in the i_{th} substream. Then we just need to set $\epsilon^{t_2}_{t_1}[i]$ empty at time t_1, and replace it with an incoming edge e if $G(e) \geqslant G(\epsilon^{t_2}_{t_1}[i])$ or $\epsilon^{t_2}_{t_1}[i]$ is empty until t_2.

Because the sliding window overlaps with at most two slices, we need to record two edges $\epsilon^{l_{new}}_{l_{old}}[i]$ and $\epsilon^T_{l_{new}}[i]$ in the i_{th} substream ($1 \leqslant i \leqslant k$). Only one of them may participate in the sample graph. There are 3 cases, as shown in Figure 4.

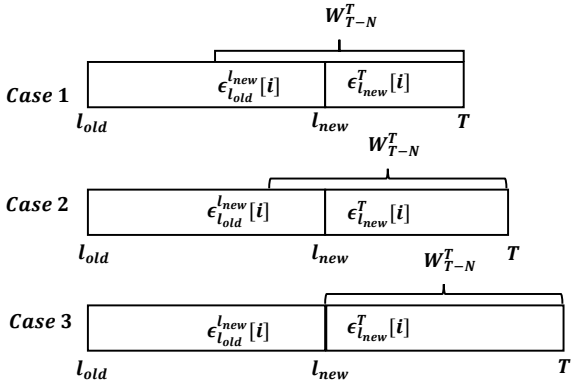


Figure 4: Different Cases in SWTC

Case 1: In case 1, both $\epsilon^{l_{new}}_{l_{old}}[i]$ and $\epsilon^T_{l_{new}}[i]$ 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 sampled edge in this substream.

Case 2: With time passing by, case 1 transfers to case 2. In this case, $\epsilon^{l_{new}}_{l_{old}}[i]$ has already expired, but the sliding window still overlaps with $W^{l_{new}}_{l_{old}}$. If $G(\epsilon^{l_{new}}_{l_{old}}[i]) > G(\epsilon^T_{l_{new}}[i])$, we cannot select a valid sampled edge from this substream. Because unexpired edges in $W^{l_{new}}_{l_{old}}$ are unknown to us. There may exist edges e' in $W^{l_{new}}_{l_{old}}$, where $G(\epsilon^{l_{new}}_{l_{old}}[i]) > G(e') > G(\epsilon^T_{l_{new}}[i])$ and $t(e') > T - N$. Therefore we cannot determine if $\epsilon^T_{l_{new}}[i]$ has the largest priority in the sliding window. There is no sampled edge in this substream in this case. On the other hand, if $G(\epsilon^T_{l_{new}}[i]) \geqslant G(\epsilon^{l_{new}}_{l_{old}}[i])$, $\epsilon^T_{l_{new}}[i]$ is a valid sample. Because it is the edge with the largest priority in $W^T_{l_{new}}$, and also has larger priority than all edges in $W^{l_{new}}_{l_{old}}$, since it has priority no less than $\epsilon^{l_{new}}_{l_{old}}[i]$. 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 $W^{l_{new}}_{l_{old}}$. In other words, $W^T_{T-N} = W^T_{l_{new}}$. In this case, $\epsilon^T_{l_{new}}[i]$ is the valid sampled edge in the sliding window.

The above three cases are repeated recursively.

Algorithm 1: Processing new edge in the SWTC

```
Input: edge e = (s, d)
Output: updated sample
1  p ← H(e)
2  if ε^T_{l_new}[p] = e then
3      Update the timestamp of ε^T_{l_new}[p]
4  else
5      if ε^T_{l_new}[p] = NULL or G(ε^T_{l_new}[p]) ⩽ G(e) then
6          G_s.remove(ε^T_{l_new}[p]) /*call Algorithm 3*/
7          ε^T_{l_new}[p] ← e
8          if ε^{l_new}_{l_old}[p] = NULL or G(ε^{l_new}_{l_old}[p]) ⩽ G(e) then
9              G_s.add(e) /*call Algorithm 2*/
10             G_s.remove(ε^{l_new}_{l_old}[p]) /*call Algorithm 3*/
```

Algorithm 2: $G_s.add(\cdot)$

```
Input: edge e
Output: updated sample
1  G_s.InsertEdge(e)
2  G_s.IncreaseTriangle(e)
```

Algorithm 3: $G_s.remove(\cdot)$

```
Input: edge e
Output: updated sample
1  if e! = NULL && e is sampled then
2      G_s.DecreaseTriangle(e)
3      G_s.DeleteEdge(e)
```

Edge processing algorithm: 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 $\epsilon^T_{l_{new}}[p]$, we just update $\epsilon^T_{l_{new}}[p]$'s timestamp to be the current time point T (Lines 2-3). Otherwise we compare it with $\epsilon^T_{l_{new}}[p]$. There are 2 cases:

(1) If $\epsilon^T_{l_{new}}[p]$ is empty or $G(\epsilon^T_{l_{new}}[p]) \leqslant G(e)$, we update edge $\epsilon^T_{l_{new}}[p]$ to be e (Lines 5-7). In this case, if the old $\epsilon^T_{l_{new}}[p]$ is a sampled edge in G_s, we need to remove $\epsilon^T_{l_{new}}$ from G_s and decreases the number of triangles containing $\epsilon^T_{l_{new}}[p]$ from the triangle counter (Lines 6).

(2) If $G(\epsilon^T_{l_{new}}[p]) > G(e)$, we do nothing.

Furthermore, in the first case, we need to further check $\epsilon^{new}_{l_{old}}[p]$ after replacing $\epsilon^T_{l_{new}}[p]$ with e. If $\epsilon^{new}_{l_{old}}[p] = NULL$ or $G(\epsilon^{new}_{l_{old}}[p]) \leqslant G(e)$, we can conclude that edge e should be selected as a sampled edge and inserted into sample graph G_s. We need to add the number of triangles containing e (Lines 8-9). Besides, if $\epsilon^{new}_{l_{old}}[p]$ is a sampled edge in G_s, we need to delete it and reduce the number of triangles containing $\epsilon^{new}_{l_{old}}[p]$ from the counter (Lines 10).

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

At a landmark, namely case 3 in Figure 4, we scan the k substreams. In each substream, we set $\epsilon^{l_{new}}_{l_{old}}[i] = \epsilon^T_{l_{new}}[i]$ and $\epsilon^T_{l_{new}}[i] = NULL$, as a new slice is about to emerge. Furthermore, If $G(\epsilon^T_{l_{new}}[i]) < G(\epsilon^{l_{new}}_{l_{old}}[i])$ in the i_{th} substream before the scanning, edge $\epsilon^T_{l_{new}}[i]$ becomes a sampled edge now. We insert it into 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 $|G_s|$. Larger $|G_s|$ leads to more accurate estimation result, which will be analyzed in Section 4.5. In this subsection, we mathematically analyze $|G_s|$ in our method SWTC and compare it with the baseline approach (proposed in Section 3). We first give a brief analysis 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 to maintain two edges in each substream. In the baseline method, we need to store the test edge and the sampled edge. In SWTC, we store the edge with the largest priority in each slice and two slices are maintained. Besides, the maximum size of 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. For each substream, we use ρ to represent the probability of selecting a valid sampled edge. Obviously, the expected sample graph size is $\rho \times k$. We have the following results about the probability of ρ in both our approach SWTC and the baseline method in Theorems 4.1 and 4.2, respectively.

Theorem 4.1. *In SWTC,* $\rho = \frac{|W^T_{T-N}|}{|W^T_{l_{old}}|}$, *where* l_{old} *is the second largest landmark which satisfies* $l_{old} \leqslant T$.

Proof. This theorem is intuitive. From Figure 4 we can see that we get a valid sample in the substream if and only if the edge with the largest priority in $W^{l_{new}}_{l_{old}}$ and $W^T_{l_{new}}$ lies in the sliding window W^T_{T-N}. Suppose there are α distinct edges in this substream in period W^T_{T-N}, and α' distinct edges in this substream in period $W^T_{l_{old}} = W^{l_{new}}_{l_{old}} + W^T_{l_{new}}$. Because each edge gets a random priority, the probability that the edge with the largest priority in $W^T_{l_{old}}$ lies in

W^T_{T-N} is equal to the ratio $\frac{\alpha}{\alpha'}$. Moreover, because edges are mapped to different substreams randomly, $\frac{\alpha}{\alpha'}$ is equal to $\frac{|W^T_{T-N}|}{|W^T_{l_{old}}|}$. □

For BPS sampling in the baseline approach (see Section 3), it is difficult to give an exact expression of the probability ρ, as it is cumulatively affected by all the edges arriving before W^T_{T-N}. The original paper [19] only gives a lower bound of ρ. And we further give an upper bound in Theorem 4.2.

Theorem 4.2. *For the baseline method,* $\frac{|W^T_{T-N}|}{|W^T_{T-2N}|} \leq \rho \leq 1 - \frac{|W^{T-N}_{T-2N}|}{|W^T_{T-3N}|}$.

Proof. **Lower bound:** If we use BPS algorithm in a substream, we will get a valid sample if the edge with the largest priority in W^T_{T-N} 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-N}_{T-2N}. Therefore the edge with the largest priority in period W^T_{T-2N} needs to be in W^T_{T-N}. According to the proof of theorem 4.1, we can see that this probability is $\frac{|W^T_{T-N}|}{|W^T_{T-2N}|}$. Therefore, a substream has a valid sampled edge is no less than $\frac{|W^T_{T-N}|}{|W^T_{T-2N}|}$.
Upper bound: From the former proof, we know that in a substream, if the edge with the largest priority in W^{T-N}_{T-3N} lies in W^{T-N}_{T-2N}, this edge, which we represent with e', will definitely become a valid sampled 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^T_{T-N}, it prevents edges in the sliding window W^T_{T-N} from becoming valid sample, and there will be no valid sampled edge in this substream. In other words, e' is the edge with the largest priority in W^T_{T-3N}, and it lies in W^{T-N}_{T-2N}. In this case there will definitely be no valid sampled edge in this substream. According to the former proof, this probability is $\frac{|W^{T-N}_{T-2N}|}{|W^T_{T-3N}|}$. Therefore, ρ is no larger than $1 - \frac{|W^{T-N}_{T-2N}|}{|W^T_{T-3N}|}$. □

According to Theorems 4.1 and 4.2, 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 4.3. *Assume that the throughput of streaming graph is st steady, in BPS* $0.5 \leqslant \rho \leqslant 0.66$*, in SWTC* $\rho = 0.75$.

Proof. When the throughput of the streaming graph is steady, the cardinality in a window $W^{t_2}_{t_1}$ is relevant with its length $t_2 - t_1$. For SWTC, $\rho = \frac{|W^T_{T-N}|}{|W^T_{l_{old}}|} = \frac{|W^T_{T-N}|}{|W^{l_{new}}_{l_{old}} + W^T_{l_{new}}|}$. The length of $W^{l_{new}}_{l_{old}}$ is always N, but the length of $W^T_{l_{new}}$ varies from 0 to N with time. Therefore ρ in *SWTC* varies from 0.5 to 1 with a steady speed, and the average value, namely the expectation, is 0.75. In BPS, ρ is a constant value. It is hard to compute the exact value, but we can get its upper bound and lower bound as shown in Theorem 4.2, The lower bound is $\frac{|W^T_{T-N}|}{|W^T_{T-2N}|} = \frac{N}{2N} = 0.5$, and the upper bound is $1 - \frac{|W^{T-N}_{T-2N}|}{|W^T_{T-3N}|} = 1 - \frac{N}{3N} = 0.66$. □

We also experimentally evaluate $|G_s|$ in both SWTC and the baseline method in Section 5, both in steady streaming graphs (Figure 5) and real-world streaming graphs (Figure 6(a) 6(b) and 7(a)). It confirms that the sample graph size in SWTC is larger than the baseline by 30%, resulting in more accurate triangle count estimation.

4.3 Optimization-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 Figure 4. Assume that $G(\epsilon^{l_{new}}_{l_{old}}[i]) > G(\epsilon^T_{l_{new}}[i])$ in case 2, there is no sampled edge in the i_{th} substream. But, when the sliding window reaches a landmark (case 2 is transferred into case 3), a new sampled 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 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 G_s. One is the effective triangle counter tc, and the other is a *vision* vc which predicates the triangle counter at the next landmark. When $G(\epsilon^T_{l_{new}}[i]) < G(\epsilon^{l_{new}}_{l_{old}}[i])$ in case 2 of Figure 4, no sampled edge is selected in this substream. However, we can forecast that at the next landmark, $\epsilon^T_{l_{new}}[i]$ will become a new sampled edge. We insert it into G_s, but tag it as an *invalid sample*. The triangles including invalid sampled edges are counted in vc, but not in tc.

The procedure of the optimized-version of SWTC is as follows:
Edge processing algorithm: When a new edge e comes and is mapped to substream p, most operations are the same as Algorithm 1, except 2 differences. First, in functions $G_s.add(\cdot)$ and $G_s.remove(\cdot)$, we need to check whether the edge is valid. If it is, we increase (or decrease) both tc and vc. Otherwise we only modify vc. Second, when the new edge e replaces $\epsilon^T_{l_{new}}[p]$, we compare it with $\epsilon^{l_{new}}_{l_{old}}[p]$, as shown in line 8 in Algorithm 1. If $G(e) \geqslant G(\epsilon^{l_{new}}_{l_{old}}[p])$ or $\epsilon^{l_{new}}_{l_{old}}[p]$ is empty, we add e to G_s as a valid sample, and the operations are the same as line $9-10$ in Algorithm 1. If $G(e) < G(\epsilon^{l_{new}}_{l_{old}}[p])$, we further check if $\epsilon^{l_{new}}_{l_{old}}[p]$ expires. If it expires, we tag e as invalid and carry out $G_s.add(e)$ to modify vc. Otherwise we do nothing.
Expiration algorithm: When an edge e in G_s expires, we first carry out $G_s.remove(e)$ to modify the counters. We can assert that in its mapped substream p, e is stored in $\epsilon^{l_{new}}_{l_{old}}[p]$ and has larger priority than $\epsilon^T_{l_{new}}[p]$. Because e expires ($t(e) \leqslant T - N < l_{new}$) and used to be a sample in G_s (corresponding to $G(\epsilon^{l_{new}}_{l_{old}}[i]) > G(\epsilon^T_{l_{new}}[i])$ in case 1 of Figure 4) . Therefore, after the deletion we add $\epsilon^T_{l_{new}}[p]$ to G_s as an invalid sampled edge, and increase vc. We only add $\epsilon^T_{l_{new}}[p]$ to G_s after expiration of $\epsilon^{l_{new}}_{l_{old}}[p]$, so that there

is at most 1 edge (whether valid or invalid) inserted into G_s in each substream. This guarantees that G_s will not has a size larger than the upper bound k.

When a landmark comes, we scan the k substreams. In each substream, we set $\epsilon^{l_{new}}_{l_{old}}[i] = \epsilon^{T}_{l_{new}}[i]$ and $\epsilon^{T}_{l_{new}}[i] = NULL$. Besides, we tag the invalid sampled edges as valid. As for the triangle count, we simply set $vc = tc$. Compared to the basic version, massive triangle counting at landmarks is avoided.

4.4 Estimating of Triangle Count

In this section, we show how to estimate the triangle count in the snapshot graph G^T with the sample graph G_s.

Suppose there are n distinct edges in G^T, and m valid sampled edges in G_s. We use tc to denote the triangle count in G_s. Because each edge in the sliding window has an equal chance to become one of the m valid sampled 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 Section 4.5.

It is difficult to directly estimate n, namely the number of distinct edges in the sliding window. Existing algorithms like [20] can not 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 Hypperloglog sketch [20] for cardinality estimation, and no other data structure is needed. Hypperloglog sketch is also the state-of-the-art for cardinality estimation.

For the i_{th} substream, we have stored $G(\epsilon^{l_{new}}_{l_{old}}[i])$ and $G(\epsilon^{T}_{l_{new}}[i])$. The larger one between them, denoted with θ, is the largest priority in this substream in $W^T_{l_{old}}$. 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 $\epsilon^{l_{new}}_{l_{old}}[i]$ and $\epsilon^{T}_{l_{new}}[i]$ 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 $W^T_{l_{old}}$, namely $|W^T_{l_{old}}|$. $|W^T_{l_{old}}|$ 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^T_{T-N}, namely n, with $|W^T_{l_{old}}|$. Suppose there are m valid samples, and M substreams that are not empty [6]. According to theorem 4.1, we can get a valid sample in a substream with probability $\frac{|W^T_{T-N}|}{|W^T_{l_{old}}|} = \frac{n}{|W^T_{l_{old}}|}$, which can be esimated as $\frac{m}{M}$. Therefore we have $n = |W^T_{l_{old}}| \times \frac{m}{M}$.

Continusous query of cardinality: In order to keep track of the cardinality to achieve continuous triangle counting, we can continuously maintain a variable $q = \sum_{i=1}^{k} 2^{-R[i]}$, and compute the cardinality with it in queries. However, like triangle counter tc, q, m and M all go through a drastic change at each landmark. Because at a landmark, emerging of new valid samples leads to change in m, and the alternation of slices results in changes in $\epsilon^{l_{new}}_{l_{old}}[i]$ and $\epsilon^{T}_{l_{new}}[i]$, causing changes in q and M. In order to avoid dense computing at one time point, we also utilize vision counting technique upon them. The details are omitted due to space limitation.

4.5 Error 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 4.4. *Suppose at time T, SWTC gets m valid sampled edges. There are n distinct edges in the snapshot graph, and the number of triangles induced by these sampled edges is tc. We use Δ^T to present the set of triangles in the snapshot graph G^T, and its number is τ. We introduce variable $\hat{\tau} = \frac{tc}{\gamma^T_{3,m}}$ where $\gamma^T_{3,m}$ is defined as*

$$\gamma^T_{j,m} = \frac{m(m-1)...(m-j+1)}{n(n-1)...(n-j+1)} \tag{1}$$

Then we have:

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

$$Var(\hat{\tau}|m) = \tau\theta^T_{3,m} + 2\zeta^T\theta^T_{5,m} + 2\eta^T\theta^T_{6,m} \tag{3}$$

where ζ^T is the number of unordered pair of distinct triangles in Δ^T which share one edge, and $\eta^T = \frac{1}{2}\tau(\tau-1) - \zeta^T$ is the number of unordered pairs of distinct triangles in Δ^T which share no edge. And we define $\theta^T_{3,m} = \frac{1}{\gamma^T_{3,m}} - 1$, $\theta^T_{5,m} = \frac{\gamma^T_{5,m}}{(\gamma^T_{3,m})^2} - 1$, $\theta^T_{6,m} = \frac{\gamma^T_{6,m}}{(\gamma^T_{3,m})^2} - 1$

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

Lemma 4.1. *At time T, the probability of SWTC sampling edge $e_1, e_2....e_j$ given m is*

$$P(e_1, e_2.....e_j \in G_s|m) = \gamma^T_{j,m} \tag{4}$$

where $\gamma^T_{j,m}$ is defined as equation 1.

Given j different edges $e_1, e_2....e_j$ and a set of different substreams $\{S_{c_1}, S_{c_2}...S_{c_j}\}$, where all these substreams have valid sampled edges. We can compute the probability that edge e_i is sampled in substream S_{c_i} $(1 \leqslant i \leqslant j)$. Because each edge is mapped into a substream at random, and the priority is randomly generated, 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 $\frac{1}{n(n-1)....(n-j+1)}$. Suppose the set φ represent the indexes where S_i has a valid sampled edge if $i \in \varphi$. $|\varphi| = m$. There exist $m(m-1)....(m-j+1)$ different ways to select indexes $\{c_1, c_2...c_j\}$ where $c_1, c_2...c_j \in \varphi$. Therefore, the overall probability that j edges $e_1, e_2....e_j$ are sampled as valid sampled edges are:

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

According to this lemma, we find that any triangle with three edges e_1, e_2, e_3 in the snapshot graph G^T has probability $\gamma^T_{3,m} =$

[6] As $n \gg k$, the probability that a substream is empty is very small, in other words $M \approx k$

$\frac{m(m-1)(m-2)}{n(n-1)(n-2)}$ to be included in the sample. Therefore, given the number of triangles in the sample, tc, we have:

$$E(\hat{\tau}|m) = E(\frac{tc}{\gamma^T_{3,m}}) = \tau$$

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

$$\begin{aligned} Var(\hat{\tau}|m) &= Var(\frac{\sum_{\sigma\in\Delta^T}\xi^T_\sigma}{\gamma^T_{3,m}}|m) \\ &= \frac{\sum_{\sigma,\sigma*\in\Delta^T} Cov(\xi^T_\sigma, \xi^T_{\sigma*}|m)}{(\gamma^T_{3,m})^2} \\ &= \frac{\sum_{\sigma\in\Delta^T} Var(\xi^T_\sigma|m)}{(\gamma^T_{3,m})^2} + \\ &\quad \frac{\sum_{\sigma,\sigma*\in\Delta^T,\sigma\neq\sigma*} E(\xi^T_\sigma\xi^T_{\sigma*}|m) - E(\xi^T_\sigma|m)E(\xi^T_{\sigma*}|m)}{(\gamma^T_{3,m})^2} \end{aligned}$$

According to lemma 4.1, we have

$$Var(\xi^T_\sigma|m) = \gamma^T_{3,m} - (\gamma^T_{3,m})^2 \tag{5}$$

$$E(\xi^T_\sigma|m)E(\xi^T_{\sigma*}|m) = (\gamma^T_{3,m})^2 \tag{6}$$

$$E(\xi^T_\sigma\xi^T_{\sigma*}|m) = \begin{cases} \gamma^T_{5,m} & \sigma \text{ and } \sigma* \text{ share one edge.} \\ \gamma^T_{6,m} & \sigma \text{ and } \sigma* \text{ share no edge.} \end{cases} \tag{7}$$

Given the definition of ζ^T, η^T, $\theta^T_{3,m}$, $\theta^T_{5,m}$ and $\theta^T_{6,m}$, we can get equation 3 in theorem 4.4 with the former equations. □

The expectation and variance of the baseline method is similar, except that the number of valid sample edges is smaller than SWTC. Therefore SWTC has a smaller variance.

4.6 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 G_s and maintain the result in tc. The other operations are the same as the binary counting. The analysis in Section 4.2 and Section 4.5 applies to weighted counting. The only difference is that denotations like $|W^{t_2}_{t_1}|$ and τ represent edge count or triangle count with duplication in weighted counting semantics. A detailed analysis is presented in the technical report [23].

Directed Graphs: 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 [25, 26]. There are multiple kinds of triangle-shape motifs with different direction constraints, and our algorithms apply to all of them. Suppose we get a sample graph with m edges with SWTC or the baseline method, and the size of the snapshot graph is n. According to Lemma 4.1, a motif with j edges has probability $\frac{m(m-1)...(m-j+1)}{n(n-1)...(n-j+1)}$ to be included in the sample graph. Therefore, for any motif with size j, we can count it in the sample graph, and divide the count with $\frac{m(m-1)...(m-j+1)}{n(n-1)...(n-j+1)}$ to get an estimation of the motif count in the snapshot graph. We focus on triangles for simplicity in this paper.

5 EXPERIMENTAL EVALUATION

In this section, we experimentally evaluate our method over three real-world datasets and one synthetic dataset. Details about the datasets, experiment settings and metrics are shown in Section 5.1, Section 5.2 and Section 5.3, respectively. As discussed in Section 4.6, 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 in Section 5.4 and Section 5.5. In these subsections, we evaluate the valid sample size and accuracy of SWTC and compare it with the baseline method. In Section 5.6, we evaluate the performance 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 Section 5.6. In Section 5.7 and Section 5.8, we further evaluate the influence of duplication ratio and the effect of vision counting. Experiments are implemented in a PC server with dual 18-core CPUs (Intel Xeon CPU E5-2697 v4@2.3G HZ, 2 threads per core) and 192G memory, running CentOS. All codes are written in C++ and compiled with GCC 4.8.5.

5.1 Data Sets

Three real-world datasets and one synthetic dataset 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:[7] 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 network:[8] 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)Actor:[9] 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.

(4)FF: This is a synthetic dataset generated by Fire-Forest model [27]. It includes 18, 311, 282 edges and 1 million nodes. There are no duplicate edges. We generate edge frequencies for it with power-law distribution and vary the duplication ratio to carry out experiments in Section 5.7.

[7] http://snap.stanford.edu/data/sx-stackoverflow.html
[8] https://webscope.sandbox.yahoo.com/catalog.php?datatype=g
[9] http://konect.cc/

The last two datasets do not have timestamps. Therefore we randomly generate timestamps for them. We shuffle the dataset three times and compute the average performance whenever we use them. For the StackOverflow and Yahoo network, as they have original timestamps, we sort the dataset by these timestamps.

5.2 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 Section 4.5, 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 Figure 7(b), 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% ∼ 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. The hash functions used in SWTC and the baseline method are BobHash [28] and MurmurHash [29]. Any popular hash functions like RSHash [30], APHash [31] and MurmurHash [29] can be used without influencing the performance. More hash functions can be found at [31].

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, and compute the average value of all checkpoints as experiment results. When the number of inserted edges 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 Actor, StackOverflow and FF we estimate the performance of the two algorithms at 100 checkpoints (when there are less than 100 checkpoints due to the limitation of the dataset size, we estimate the performance at all available checkpoints). For Yahoo network, in which the window length is very large and compute the accurate triangle count is too time consuming, we estimate the performance at 40 checkpoints.

5.3 Metrics

In the experiments we evaluate 3 metrics of the algorithms: average valid sample size, percentage of valid sample, and MAPE of triangle count, defined as follows:

Average Valid Sample Size: In both SWTC and the baseline method, the number of valid sampled edges varies as the window slides. We measure the number of valid sampled 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 sampled 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 $\left|\frac{\hat{\tau}-\tau}{\tau}\right|$. We compute the average value of all the checkpoints to get Mean Absolute Percentage Error (MAPE).

Besides, in Section 5.7, we vary the duplication ratio the carry out experiments. The duplication ratio is defined as follows:

Duplication Ratio: $1 - \frac{number\ of\ distinct\ edges}{total\ number\ of\ edges}$.

5.4 Valid Sample Size

We conduct experiments on sample size in two ways. First, in order to confirm our mathematical analysis in Section 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 dataset, the cardinality of a window is linear correlated with the window size. 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 Figure 5, where the window length is set to 4 million and the x-axis denotes the total number of processed time units (i.e. average time span defined in Section 5.1). The sample rate is set to 4%. In Figure 5, 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 various varies from 50% to 100% in a cycle, and the average value is 75%. This conforms to our mathematical analysis in theorem 4.3. Moreover, at 25 checkpoints, SWTC gets a much larger valid sample size. At the remaining 6 checkpoints, SWTC and the baseline method obtain similar valid sample size.

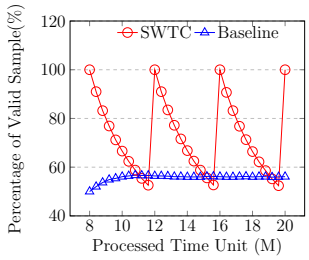


Figure 5: Percentage of Valid Sample

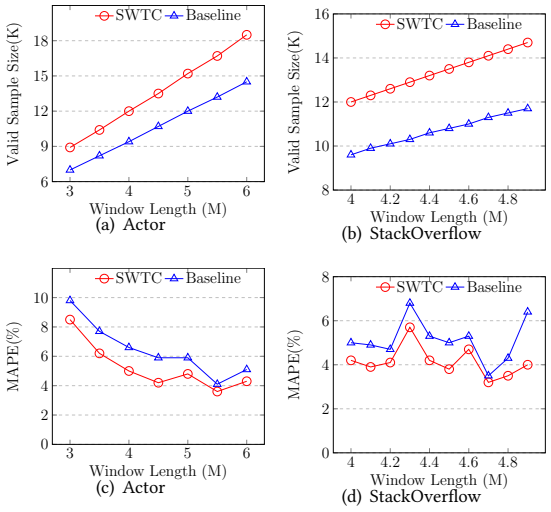
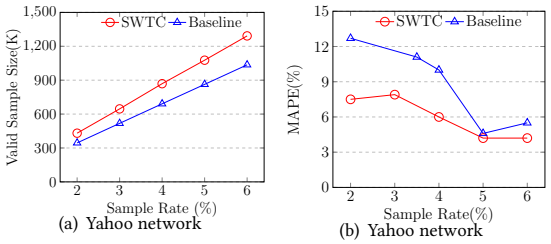


Figure 6: Performance Varying with Window Size



(a) Yahoo network (b) Yahoo network

Figure 7: Performance Varying with Sample Rate

We also report the average valid sample size in the three datasets in Figures 6(a) 6(b) and 7(a). For Actor and StackOverflow, we fix the sample rate to be 4% and vary the window length. For Yahoo network, we fix the window length to be 35 million 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 the both brings a larger k. And SWTC always has a larger sample size than the baseline method. The gap between them 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.

5.5 Accuracy

Figures 6(c), 6(d) and 7(b) show the mean absolute percentage error (MAPE) of all checkpoints in each dataset to measure the accuracy of SWTC and the baseline method. The parameter settings are the same as the valid sample size experiments. From the figures, we can see that SWTC has an MAPE up to 62% smaller than the baseline. And in all experiments, MAPE of SWTC is below 0.1. In Figure 6(c), we can see that the MAPE has a decreasing trend as the window length grows. Because when there are more triangles in the window, the influence of randomicity decreases and the estimated result becomes stable and accurate. Figure 7(b) shows that MAPE has a trend of decrement when the sample rate grows. This is intuitive, as a larger sample set produces a higher accuracy.

5.6 Experiments on Weighted Counting

We carry out an experiment on weighted counting with StackOverflow dataset, with window length set to 4.5 million. Besides comparing with the baseline, we also compare SWTC with 2 prior algorithms in fully dynamic stream model, WRS[22] and ThinkD [18]. The MAPE is shown in Figure 8, where the x axis is the memory usage. As discussed in Section 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 5.4 million. Therefore, we reserve space for storing 5.4 million edges for WRS and ThinkD. Each edge has 2 node IDs and one timestamp, each of which occupies 8 bytes. As the edges are organized as a linked list, an additional pointer is needed by each edge. Therefore 32 bytes are needed for each edge in the sliding window. In total, WRS and ThinkD need at least 172.8M memory to start to work. Therefore, in the figure we begin to present their accuracy at 180M. We can see that they begin to have MAPE lower than 4% only when the memory is larger than 240M. On the other hand, our algorithms get same accuracy with only 60M 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 Figure 8 also shows that in weighted counting, SWTC still has a higher accuracy than the baseline.

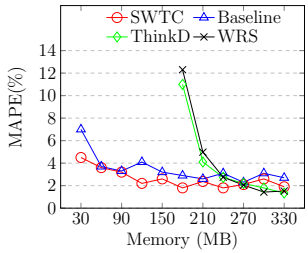


Figure 8: Accuracy of Weighted Counting

5.7 Influence of Duplication Ratio

In order to evaluate the influence of duplication ratio of the streaming graph, we use the synthetic dataset FF to carry out experiments. We generate edge frequencies with power-law distribution and vary the duplication ratio. The window length is set to be 3 million and the sample rate is set to be 4%. The memory usage and the valid sample size does not change with the duplication ratio. In binary counting semantics, MAPE decreases with the increment of duplication ratio, the result is shown in Figure 9. Because with more duplicate edges, the number of distinct edges in the sliding window decreases, and the sample size becomes relatively large. In weighted counting semantics, the accuracy does not change with the duplication ratio, because we treat duplicate edges the same as distinct edges. We omit the figure due to space limitation.

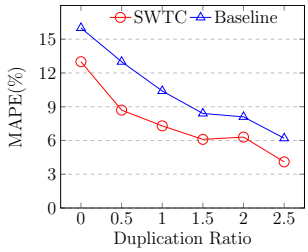


Figure 9: Accuracy Varying with Duplication Ratio

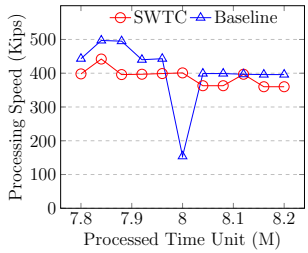


Figure 10: Processing Speed of Different Versions of SWTC

5.8 The Effect of Vision Counting

In order to illustrate the effect of vision counting, we compare the speed fluctuation of the final version of SWTC with the basic version. We call the basic version which does not use vision counting as SWTC-nv. We use FF dataset in this experiment, and set the window length to be 4 million. 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 $40K$ time units, and draw the curve of processing speed varying with total number of processed time units. The result is shown in Figure 10. The measurement of speed is kilo insertions per second (Kips). The figure shows the change of speed at a landmark. We can see that at the landmark, the speed of SWTC-nv suffers from a sharp decrease. Because at the landmark, SWTC-nv need a period as long as $0.04s$ to add new valid samples and count the triangles. The processing of edges received during this period is delayed. This delay will be worse when the sliding window is larger. On the other hand, if we use vision counting, though the speed is a little lower, but there will be no computation peak at the landmark.

6 RELATED WORK

6.1 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). In this paper we focus on counting global triangles. Compared to algorithms [32–36] which exactly count the number of triangles in large graphs, approximately counting algorithms [37–40] 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. [41] which samples wedges to estimate triangle count. Tsourakakis et at. [42] 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. TRIÈST [17] uses reservoir sampling method [43] and has a fixed sample size. It supports edge deletions in fully dynamic streaming graphs with a technique named random pairing [44]. 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 Section 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.

6.2 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, 45, 46]. 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. The extended version where multiple samples are produced can be found in the original paper [19].

6.3 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 $c_i (1 \leqslant i \leqslant m)$. 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 $\frac{1.04}{\sqrt{m}}$.

7 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 sample 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. Mathematical analysis and experiments show that it 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.

ACKNOWLEDGEMENT

This work was supported by Scientific and technological innovation 2030 - new generation of artificial intelligence major project 2020AAA0108505 and NSFC under grant 61932001, 61961130390, U20A20174. This work was also partially supported by Beijing Academy of Artificial Intelligence (BAAI) and Key Research and Development Program of Hubei Province (No. 2020BAB026). The corresponding author of this work is Lei Zou (zoulei@pku.edu.cn).

REFERENCES

[1] Jonathan W Berry, Hendrickson Bruce, Randall A Laviolette, and Cynthia A Phillips. Tolerating the community detection resolution limit with edge weighting. *Physical Review E Statistical Nonlinear Soft Matter Physics*, 83(5 Pt 2):056119, 2011.

[2] Eckmann Jean-Pierre and Moses Elisha. Curvature of co-links uncovers hidden thematic layers in the world wide web. *Proc Natl Acad Sci U S A*, 99(9):5825–5829, 2002.

[3] Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. Efficient algorithms for large-scale local triangle counting. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 4(3):13, 2010.

[4] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[5] U Kang, Brendan Meeder, Evangelos E Papalexakis, and Christos Faloutsos. Heigen: Spectral analysis for billion-scale graphs. *IEEE Transactions on knowledge and data engineering*, 26(2):350–362, 2012.

[6] Zhi Yang, Christo Wilson, Xiao Wang, Tingting Gao, Ben Y Zhao, and Yafei Dai. Uncovering social network sybils in the wild. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 8(1):1–29, 2014.

[7] Zhenjun Li, Yunting Lu, Wei-Peng Zhang, Rong-Hua Li, Jun Guo, Xin Huang, and Rui Mao. Discovering hierarchical subgraphs of k-core-truss. *Data Science and Engineering*, 3(2):136–149, 2018.

[8] A. Pavan, Kanat Tangwongsan, Srikanta Tirthapura, and Kun Lung Wu. Counting and sampling triangles from a graph stream. *Proceedings of the Vldb Endowment*, 6(14):1870–1881, 2013.

[9] Nesreen K. Ahmed, Nick Duffield, Jennifer Neville, and Ramana Kompella. Graph sample and hold: A framework for big-graph analytics. In *Acm Sigkdd International Conference on Knowledge Discovery Data Mining*, 2014.

[10] Pinghui Wang, Yiyan Qi, Sun Yu, Xiangliang Zhang, and Xiaohong Guan. Approximately counting triangles in large graph streams including edge duplicates with a fixed memory usage. *Proceedings of the Vldb Endowment*, 11(2):162–175, 2017.

[11] P Oscar Boykin and Vwani P Roychowdhury. Leveraging social networks to fight spam. *Computer*, 38(4):61–68, 2005.

[12] Mayur Datar, Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Maintaining stream statistics over sliding windows. *Siam Journal on Computing*, 31(6):1794–1813, 2002.

[13] Youhuan Li, Lei Zou, M Tamer Özsu, and Dongyan Zhao. Time constrained continuous subgraph search over streaming graphs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, pages 1082–1093. IEEE, 2019.

[14] Michael S Crouch, Andrew McGregor, and Daniel Stubbs. Dynamic graphs in the sliding-window model. In *European Symposium on Algorithms*, pages 337–348. Springer, 2013.

[15] Xiafei Qiu, Wubin Cen, Zhengping Qian, You Peng, Ying Zhang, Xuemin Lin, and Jingren Zhou. Real-time constrained cycle detection in large dynamic graphs. *Proceedings of the VLDB Endowment*, 11(12):1876–1888, 2018.

[16] Minsoo Jung, Yongsub Lim, Sunmin Lee, and U Kang. Furl: Fixed-memory and uncertainty reducing local triangle counting for multigraph streams. *Data Mining and Knowledge Discovery*, 33(5):1225–1253, 2019.

[17] Lorenzo De Stefani, Alessandro Epasto, Matteo Riondato, and Eli Upfal. Triest: Counting local and global triangles in fully dynamic streams with fixed memory size. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 11(4):1–50, 2017.

[18] Kijung Shin, Sejoon Oh, Jisu Kim, Bryan Hooi, and Christos Faloutsos. Fast, accurate and provable triangle counting in fully dynamic graph streams. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 14(2):1–39, 2020.

[19] Rainer Gemulla and Wolfgang Lehner. Sampling time-based sliding windows in bounded space. In *Acm Sigmod International Conference on Management of Data*, 2008.

[20] Philippe Flajolet, Éric Fusy, Olivier Gandouet, and Frédéric Meunier. 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] Daniel Ting. 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 Lee, Kijung Shin, and Christos Faloutsos. 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 Babcock, Mayur Datar, and Rajeev Motwani. 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] George M Slota and Kamesh Madduri. Complex network analysis using parallel approximate motif counting. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 405–414. IEEE, 2014.

[26] Marco Bressan, Flavio Chierichetti, Ravi Kumar, Stefano Leucci, and Alessandro Panconesi. Motif counting beyond five nodes. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 12(4):1–25, 2018.

[27] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. Graph evolution: Densification and shrinking diameters. *ACM transactions on Knowledge Discovery from Data (TKDD)*, 1(1):2–es, 2007.

[28] Bobhash function. Published by Bob Jenkins at http://burtleburtle.net/bob/hash/doobs.html.

[29] Murmurhash function. Published by Austin Appleby at https://github.com/aappleby/smhasher.

[30] Robert Sedgewick. *Algorithms in c*. Pearson Education, 2001.

[31] Aphash and collection of other hash functions. Published by Arash Partow at http://www.partow.net/programming/hashfunctions/#RSHashFunction.

[32] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

[33] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. Patric: A parallel algorithm for counting triangles in massive networks. In *Acm International Conference on Information Knowledge Management*, 2013.

[34] Xiaocheng Hu, Yufei Tao, and Chin Wan Chung. Massive graph triangulation. In *Acm Sigmod International Conference on Management of Data*, 2013.

[35] Jinha Kim, Wook Shin Han, Sangyeon Lee, Kyungyeol Park, and Hwanjo Yu. Opt:a new framework for overlapped and parallel triangulation in large-scale graphs. 2014.

[36] Ha-Myung Park, Sung-Hyon Myaeng, and U Kang. 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.

[37] Ziv Bar-Yossef, Ravi Kumar, and D Sivakumar. 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.

[38] Buriol, S Luciana, Frahling, Gereon, Leonardi, Stefano, Marchetti-Spaccamela, Alberto, Sohler, and Christian. Counting triangles in data streams. In *Acm Sigmod-sigact-sigart Symposium on Principles of Database Systems*, 2006.

[39] Hossein Jowhari and Mohammad Ghodsi. New streaming algorithms for counting triangles in graphs. In *International Computing and Combinatorics Conference*, pages 710–716. Springer, 2005.

[40] Yongsub Lim and U Kang. 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.

[41] Madhav Jha, C. Seshadhri, and Ali Pinar. A space efficient streaming algorithm for triangle counting using the birthday paradox. 2013.

[42] Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. 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.

[43] Jeffrey S Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[44] Rainer Gemulla, Wolfgang Lehner, and Peter J Haas. Maintaining bounded-size sample synopses of evolving datasets. *The VLDB Journal*, 17(2):173–201, 2008.

[45] Vladimir Braverman, Rafail Ostrovsky, and Carlo Zaniolo. Optimal sampling from sliding windows. In *Twenty-eighth Acm Sigmod-sigact-sigart Symposium on Principles of Database Systems*, 2009.

[46] Graham Cormode, Shanmugavelayutham Muthukrishnan, Ke Yi, and Qin Zhang. 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.