Approximately Counting Triangles in Large Graph Streams Including Edge Duplicates with a Fixed Memory Usage

Pinghui Wang[1,4], Yiyan Qi[1]*, Yu Sun[1], Xiangliang Zhang[2], Jing Tao[1], Xiaohong Guan[1,3]
[1]NSKEYLAB, Xi'an Jiaotong University, China
[2]King Abdullah University of Science and Technology, Thuwal, SA
[3]Department of Automation and NLIST Lab, Tsinghua University, Beijing, China
[4]Shenzhen Research Institute of Xi'an Jiaotong University, Shenzhen, China
Email: {phwang, jtao, xhguan}@mail.xjtu.edu.cn, {qiyiyan, sunyuxajd2013}@stu.xjtu.edu.cn, xiangliang.zhang@kaust.edu.sa

ABSTRACT

Counting triangles in a large graph is important for detecting network anomalies such as spam web pages and suspicious accounts (e.g., fraudsters and advertisers) on online social networks. However, it is challenging to compute the number of triangles in a large graph represented as a stream of edges with a low computational cost when given a limited memory. Recently, several effective sampling-based approximation methods have been developed to solve this problem. However, they assume the graph stream of interest contains no duplicate edges, which does not hold in many real-world graph streams (e.g., phone calling networks). In this paper, we observe that these methods exhibit a large estimation error or computational cost even when modified to deal with duplicate edges using deduplication techniques such as Bloom filter and hash-based sampling. To solve this challenge, we design a one-pass streaming algorithm for uniformly sampling distinct edges at a high speed. Compared to state-of-the-art algorithms, our algorithm reduces the sampling cost per edge from $O(\log k)$ (k is the maximum number of sampled edges determined by the available memory space) to $O(1)$ without using any additional memory space. Based on sampled edges, we develop a simple yet accurate method to infer the number of triangles in the original graph stream. We conduct extensive experiments on a variety of real-world large graphs, and the results demonstrate that our method is several times more accurate and faster than state-of-the-art methods with the same memory usage.

PVLDB Reference Format:
Pinghui Wang, Yiyan Qi, Yu Sun, Xiangliang Zhang, Jing Tao, and Xiaohong Guan. Approximately Counting Triangles in Large Graph Streams Including Edge Duplicates with a Fixed Memory Usage. *PVLDB*, 11(2): 162 - 175, 2017.
DOI: https://doi.org/10.14778/3149193.3149197

*Pinghui Wang and Yiyan Qi contributed equally to this work. Yiyan Qi is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 11, No. 2

DOI: https://doi.org/10.14778/3149193.3149197

1. INTRODUCTION

Recently, a considerable attention has been paid to designing one-pass streaming algorithms for mining large graph streams without storing the entire graph, because many real-world networks are given in a stream fashion, whose size (i.e., the number of edges at the end of the stream) is unknown, or even infinite. For example, computer network traffic can be modeled as a graph stream where a node represents a computer and an edge in the graph stream represents a packet transmitting from one computer to another. Similarly, a stream of phone calling records can also be modeled as a graph stream where a node represents a phone number and an edge in the graph stream represents a call from one phone number to another. Many other examples can be found in other areas such as online social networks, instant messaging networks, email networks, and financial networks. Due to the large-size and high-speed nature of these graph streams, it is prohibitive to collect the entire graph for many graph mining applications such as anomaly detection. Therefore, it is critical to develop one-pass streaming algorithms with features: 1) **computationally efficient**, e.g., quickly approximate metrics of interest; 2) **allow users to easily set algorithm parameters in appropriate values to maximize the performance of algorithms**, e.g., allow users to easily specify a limit on memory usage. Many streaming algorithms do not meet this requirement. Take random edge sampling as an example, which greatly reduces the computational cost and memory usage by sampling each edge in the graph stream of interest with a fixed probability p. However, it is problematic to use a fixed p when the graph stream's size is not known. That is, the algorithm may run out of memory when users specify a large p, because the number of sampled edges grows with the size of the graph stream. In contrary, it provides suboptimal estimations when using a smaller p because the available memory is not fully used.

In this paper, we study how to compute the number of triangles in a large graph stream, which is one of the most widely studied graph mining problems. It has been successfully used to detect network anomalies such as spam pages [6] and suspicious accounts (e.g., fraudsters and advertisers) on online social networks [23,52]. Other applications include social role identification [50], community detection [7], topic mining [14], and motif detection [7,31]. To handle large-scale graph streams, most previous works are sampling-based algorithms (e.g., edge sampling, wedge sampling, and triple sampling) for estimating the number of triangles [1, 5, 9, 20, 21, 26, 30, 38, 48]. However, these algorithms usually require users to specify in advance a fixed edge sampling probability p. As mentioned, it is difficult to set the optimal value of p when the graph's size is un-

known. Even when it is possible to specify a fixed p to fully use the available memory space at the end of the graph stream, it is still not memory-efficient for estimating the triangle count *over time*, because the allocated memory is not fully used at any time before the end of the graph stream. To address the above issues, De Stefani et al. [44] present several triangle count approximation algorithms using the reservoir sampling technique [49] to sample edges with a fixed amount of memory.

However, all the above works assume there exist no edge duplicates in the graph stream of interest, which does not hold in real-world graph streams such as sequences of packets in computer networks, calling records in phone networks, transactions in financial networks, and emails in email networks. In addition, many of these real networks are directed graphs, and counting directed triangles in figure 1 has been used to characterize evolution patterns in online social networks [10, 41], evaluate the transitivity of directed graphs [42], and detect web spams [6]. Directed graph streams can be modeled as a stream of edges $e^{(1)}, e^{(2)}, \cdots$, where $e^{(t)} = (u^{(t)}, v^{(t)}, l^{(t)})$, $t \geq 1$, is the t^{th} edge, $l^{(t)} \in \{\rightarrow, \leftarrow\}$ is the direction label of edge $e^{(t)}$ between nodes $u^{(t)}$ and $v^{(t)}$. Let $G^{(t)}$ be the directed graph built on edges $e^{(1)}, \cdots, e^{(t)}$. Two nodes u and v may have more than one edge between them occurred in the edge stream, and the direction label of edge (u, v) in $G^{(t)}$ may change over time. For example, there may exist two edges $e^{(l)} = (u, v, \rightarrow)$ and $e^{(t)} = (u, v, \leftarrow)$ in the graph stream, where $1 \leq l < t$, and the direction label of edge (u, v) in $G^{(t)}$ changes from $\rightarrow$ to $\leftrightarrow$ at time t. To some extent, a directed graph stream can be viewed as an undirected edge stream with undirected edge duplicates and updates of edge direction labels. We observe that it is not trivial to deal with edge duplicates when estimating the number of triangles in a large graph stream. More specifically, we summarize the shortcomings of existing methods as:

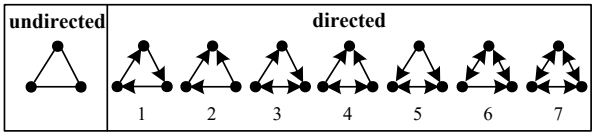


Figure 1: Undirected/Directed triangles. The numbers are type IDs of different directed triangles.

• ***Existing triangle count estimation algorithms produce inaccurate results when applied to graph streams including edge duplicates.*** When we directly apply existing sampling algorithms such as the state-of-the-art method Trièst [44] to a large graph stream including edge duplicates, their original probabilistic models fail to model sampled edges due to the existence of duplicate edges, so estimations given by the probabilistic models are not able to converge to the correct value. For example, when applying the reservoir sampling to sample edges in the graph stream in a direct manner, at any time, (at most) k edges are randomly sampled from all the edges presented so far, but they are not necessarily distinct due to the existence of edge duplicates, where k is the user-specified reservoir size, i.e., the maximum number of sampled edges. Unlike sampling each edge uniformly in [44], the reservoir sampling tends to sample edges with a large number of duplicates and it leads to the estimation method in [44] being inaccurate for graph streams including edge duplicates. We notice that De Stefani et al. [45] (the extended version of [44]) extend their method Trièst to approximate the number of triangles in a large *multigraph* stream. Compared to simple graphs having no more than one edge between two nodes, there may exist multiple edges between two nodes in multigraphs. The set of triangles in multigraphs defined in [44] is allowed to contain multiple triangles with the same set of nodes. Take a multigraph with edges $e_1 = (u, w), e_2 = (u, w), e_3 = (u, v), e_4 = (w, v)$ as an example. The multigraph has two triangles between nodes u, v, and w consisting of edges: 1) e_1, e_3, e_4; and 2) e_2, e_3, e_4. Therefore, Trièst essentially treats all edges in a stream graph as distinct edges no matter whether they have the same two endpoints or not. Clearly, it is different from the problem studied in this paper.

• ***Hash-based sampling can help on dealing with edge duplicates but causes serious issues such as high computational cost and large memory usage.*** To handle edge duplicates, one can use a uniform random hash function r to hash each edge e into the range $(0, 1)$, and e is sampled when $r(e) \leq p$. This hash-based sampling is essentially equivalent to the method of sampling edges with a fixed probability p, so it has serious issues (e.g., the algorithm may run out of memory) for analyzing large graph streams discussed previously. Moreover, to avoid storing all duplicates of a sampled edge e with $r(e) \leq p$ in memory, hash-based sampling requires an extra computational cost to check whether a coming edge e has been already sampled when $r(e) \leq p$. Using hash-based sampling, Jha et al. [19] recently present the first triangle count approximation algorithm for large graph streams including edge duplicates. Besides no guarantee on the amount of memory usage, it also has a high computational cost because it enumerates each sampled wedge to check whether a coming edge is the closing edge or one edge of the wedges. Jung et al. [22] develop an algorithm FURL that immediately extends Trièst [44] by combining reservoir and hash-based sampling techniques. It keeps track of k distinct edges with the smallest hash values at each time. However, the disadvantage of FURL is its computational complexity $O(k)$ in the sampling process of each edge. Fast-searching and fast-sorting algorithms can be used to reduce this complexity to $O(\log k)$, but require a large amount of memory for storing auxiliary information (e.g., child pointers in a search tree) besides sampled edges.

• ***Bloom filter [8] has the potential to handle edge duplicates but requires non-neglectable additional cost (memory usage and computing time) and introduces errors difficult to be corrected.*** Lim et al. [30] suggest dealing with edge duplicates by using a Bloom filter, which is a space-efficient probabilistic approach for testing whether an element is a member of a set. A Bloom filter consists of m bits and z independent hash functions. All bits are initialized to zero. To insert an edge into the Bloom filter, z bits are randomly selected from the m bits using the z hash functions, and then set to one. To check whether a coming edge in a graph stream is presented previously, the Bloom filter uses the z hash functions to get z bits associated with the edge. When any of these z bits is zero, the edge is definitely not presented previously and then inserted into the Bloom filter. Otherwise, the edge is discarded as a duplicate edge, because it is either presented previously or its associated z bits have been set to one by chance during the insertion of other edges, which results in a false positive. The more edges added to the Bloom filter, the more false positives caused. Let $n^{(t)}$ denote the number of distinct edges occurred up to time t. Then, the false positive probability of wrongly discarded edges equals $p^{(t)} = (1 - (1 - 1/m)^{zn^{(t)}})^z$ at time t. To achieve a small desired false positive probability $p^{(t)}$, the Bloom filter requires at least $m = -\frac{n^{(t)} \ln p^{(t)}}{(\ln 2)^2}$ bits, given that z is set to the optimal value $z = \frac{m \ln 2}{n^{(t)}}$. For example, to guarantee $p^{(t)} \leq 0.001$, the Bloom filter requires $m = 14.4n^{(t_{\max})}$ bits of memory given $z = 10.0$, where $t_{\max}$ is the discrete time of the graph stream's last edge. When introducing Bloom filter to existing triangle count ap-

proximation algorithms, non-neglectable extra memory usage and computational cost are introduced by the Bloom Filter. Moreover, the false positives of wrongly discarded edges also make the estimation errors of triangle count approximation algorithms uncontrollable and not quantifiable, because the value of false positive probability $p^{(t)}$ is unknown and varies overtime.

Our contributions. To solve the above challenges, we design a one-pass streaming algorithm PartitionCT to fast and accurately estimate the number of triangles in a large graph stream including edge duplicates. PartitionCT only requires users to specify one parameter k, i.e., the maximum number of sampled edges determined by the available memory space. To deal with edge duplicates, it randomly assigns different ranks for different edges in the graph stream and hashes edges into k buckets. At any time, a bucket holds only one edge and keeps track of the edge with the smallest rank among all edges occurred so far that are hashed into the bucket. Based on edges stored in all k buckets over time, we develop a novel method to accurately estimate the number of triangles and provide theoretical proofs for the accuracy of our method. Compared to the state-of-the-art method FURL [22], PartitionCT reduces the sampling cost per edge from $O(\log k)$ to $O(1)$ with no additional memory usage. We conduct extensive experiments on a variety of real-world large graphs, and experimental results show that 1) PartitionCT is several times more accurate than FURL and other state-of-the-art methods with the same memory usage; 2) when setting the same k, PartitionCT is better than or comparable to FURL on estimation accuracy but significantly outperforms FURL in terms of running time and memory usage. To guarantee the reproducibility of the experimental results, we release the source code of our algorithm in open source[1].

The rest of this paper is organized as follows. The problem formulation is presented in Section 2. Section 3 summarizes related work. Sections 4 and 5 present our algorithms of approximately counting triangles in undirected and directed graph streams. The performance evaluation and testing results are presented in Section 6. Conclusions then follow.

2. PROBLEM FORMULATION

To formally define our problem, we first introduce some notation. Let Π denote the graph stream of interest, representing a stream of edges $e^{(1)}, e^{(2)}, \ldots, e^{(t_{\max})}$, where $t_{\max}$ is the time of the last edge of Π. For any discrete time $t > 0$, $e^{(t)} = (u^{(t)}, v^{(t)})$ represents the t^{th} edge of Π, where $u^{(t)}$ and $v^{(t)}$ are the edge's two endpoints. For ease of presentation, in this paper we model directed graphs as undirected graphs with edge direction label (in short, edge label). That is, we denote $e^{(t)} = (u^{(t)}, v^{(t)}, l^{(t)})$, where $l^{(t)} \in \{\rightarrow, \leftarrow\}$ is the label of edge $(u^{(t)}, v^{(t)})$. Let $G^{(t)} = (V^{(t)}, E^{(t)}, L^{(t)})$ denote the graph occurred up to and including time t, where $V^{(t)}$, $E^{(t)}$, and $L^{(t)}$ are the sets of nodes, edges, and edge labels respectively. We use $l^{(t)}_{u,v} \in \{\rightarrow, \leftarrow, \leftrightarrow\}$ to denote the label of an edge (u, v) in $E^{(t)}$. Denote $V^{(0)} = E^{(0)} = L^{(0)} = \emptyset$. Then we observe that $G^{(t)}$ can be computed incrementally as: $V^{(t)} = V^{(t-1)} \cup \{u^{(t)}, v^{(t)}\}$ and $E^{(t)} = E^{(t-1)} \cup \{(u^{(t)}, v^{(t)})\}$. We easily find that $l^{(t)}_{u^{(t)},v^{(t)}} = l^{(t)}$ when edge $(u^{(t)}, v^{(t)})$ has no duplicate occurred before time t, i.e., $(u^{(t)}, v^{(t)}) \notin E^{(t-1)}$. Otherwise, $l^{(t)}_{u^{(t)},v^{(t)}}$ is computed based on both $l^{(t-1)}_{u^{(t)},v^{(t)}}$ and $l^{(t)}$. For example, when the t^{th} edge in Π is $e^{(t)} = (u, v, \rightarrow)$ and the label of edge (u, v) is $l^{(t-1)}_{u,v} = \leftarrow$ at time $t-1$, then at the end of time t we update the label of edge (u, v) from $l^{(t-1)}_{u,v} = \leftarrow$ to $l^{(t)}_{u^{(t)},v^{(t)}} = \leftrightarrow$.

[1]http://nskeylab.xjtu.edu.cn/dataset/phwang/code/PartitionCT.zip

In summary, edges in the graph streaming model studied in this paper have: **1) duplicates**, i.e., an edge in Π may appear more than once, which is similar to the data streaming model in [19, 22]; **2) time-variant edge direction labels**, i.e., each edge label $l^{(t)}_{u^{(t)},v^{(t)}}$ may change with t, which results in existing streaming algorithms failing to approximately count directed triangle patterns over time.

Let $\Delta^{(t)}$ denote the set of triangles in undirected graph $G^{(t)}$. Let $\tau^{(t)} = |\Delta^{(t)}|$ be the triangle count, where $|\Delta^{(t)}|$ is the cardinality of set $\Delta^{(t)}$. For directed graphs, as shown in figure 1, there exist 7 different types of triangles. For the i^{th} type triangle, $1 \le i \le 7$, similarly, we let $\Delta^{(i,t)}$ denote the set of its instances (i.e., induced 3-node subgraphs in $G^{(t)}$ isomorphic to the i^{th} type triangle) in $G^{(t)}$. Let $\tau^{(i,t)} = |\Delta^{(i,t)}|$. In this paper, we aim to design a fast one-pass streaming algorithm to solve two tasks for undirected graphs: **task 1**) estimate $\tau^{(t)}$ over time, i.e., all $\tau^{(1)}, \ldots, \tau^{(t_{\max})}$; **task 2**) estimate only $\tau^{(t_{\max})}$. Our algorithm PartitionCT solves these two tasks in slightly different manners. That is, similar to state-of-the-art methods [22, 30, 44] for task 1, PartitionCT spends a significant time cost on dynamically keeping track of the number of triangles consisting of sampled edges. In contrary, the computational cost of task 2 is dominantly determined by the average time in the sampling process, i.e., the average time to determine whether to sample a coming edge and the average time to store a sampled edge. Task 2 is useful for time interval based applications such as network traffic anomaly detection. For example, Π is a network packet stream collected on a router in a time interval (e.g., one hour in a day), and one wants to compute the triangle count for each interval, i.e., $\tau^{(t_{\max})}$ for each interval. In addition to undirected graphs, we also extend our algorithm to solve two tasks for directed graphs: **task 3**) estimate $\tau^{(1,t)}, \ldots, \tau^{(7,t)}$ over time; **task 4**) estimate only $\tau^{(1,t_{\max})}, \ldots, \tau^{(7,t_{\max})}$. For ease of reading, we list notation used throughout the paper in Table 1.

Table 1: Table of notation.

Π	graph stream of interest
$G^{(t)} = (V^{(t)}, E^{(t)}, L^{(t)})$	graph at the end of time t, where $V^{(t)}$, $E^{(t)}$, and $L^{(t)}$ are node, edge edge label sets
$n^{(t)}$	the number of distinct edges presented at the end of time t
$d^{(t)}_u$	the number of edges in graph $G^{(t)}$ connecting to node $u \in V^{(t)}$
$t_{\max}$	the discrete time of the last edge
k	the size of user-specified memory space
$S[1], \ldots, S[k]$	buckets used in PartitionCT
$S^{(t)}$	edges sampled at the end of time t
$G^{(t)}_S$	graph consists of edges in $S^{(t)}$
$N^{(t)}_{u,S}$	the set of neighbors of u in $G^{(t)}_S$
$\Delta^{(t)}$	triangles in $G^{(t)}$
$\Delta^{(i,t)}, 1 \le i \le 7$	i^{th} type directed triangles in $G^{(t)}$
$\tau^{(t)} = \lvert\Delta^{(t)}\rvert$	the number of triangles
$\tau^{(i,t)} = \lvert\Delta^{(i,t)}\rvert$	the number of i^{th} type directed triangles

3. RELATED WORK

Global triangle count estimation. Compared to methods of exactly listing and counting triangles in a large graph [2, 3, 17, 18, 24, 28, 34–37, 39, 40, 46, 53], approximation methods require much less computational and memory resources. Bar-Yossef et al. [5] reduce

the problem of counting triangles to estimate zero-th, first, and second frequency moments for a stream of node triples, and propose the first one-pass streaming algorithm for estimating the number of triangles in a graph stream. Subsequently, [9, 21] present several sampling methods to further reduce the space complexity of the algorithm in [5]. Algorithms in [21] sample node triples based on random edge sampling, and algorithms in [9] sample triplets by combining random edge and node sampling techniques. These algorithms are far from practical for handling large graph streams due to unsatisfied performance (e.g., large estimation errors and high computational complexity) and unrealistic assumptions. For example, the one-pass streaming algorithm in [9] requires the entire node set is known in advance. To address these issues, very recently, [1, 20, 38, 44, 48] develop several one-pass streaming algorithms for estimating the number of triangles in a large graph stream. Jha et al. [20] develop a wedge sampling based algorithm to estimate the triangle count. Pavan et al. [38] present a neighborhood sampling method to sample and count triangles. Tsourakakis et at. [48] present a triangle sparsification method by sampling each and every edge with a fixed probability, which can be directly applied to estimate the number of triangles in a graph stream. Ahmed et al. [1] present a more general edge sampling framework for estimating a variety of graph statistics including the triangle count. De Stefani et al. [44] use a *fixed user-specified* memory space to sample edges by the reservoir sampling technique [49], and then estimate the triangle count from these edges sampled not-independently. All the above works assume there is no edge duplicate in the graph stream of interest. To deal with edge duplicates, Jha et al. [19] use a hash-based sampling technique to sample distinct edges with a fixed probability. Besides no guarantee on the amount of memory usage, their method also has a high computational cost because it enumerates each sampled wedge to check whether a coming edge is the closing edge or one edge of the wedges. In addition to streaming algorithms, [43, 51] present sampling algorithms for estimating the number of triangles in a large static graph that can be entirely fitted into memory space.

Local triangle count estimation. [6, 26, 30] focus on computing local (i.e., incident to each node) triangles counts for large graph streams. Becchetti et al. [6] develop a semi-streaming algorithm to estimate local triangle counts. However, their algorithm requires multiple ($\log n$) passes. Kutzkov and Ragh [26] develop a one-pass streaming algorithm. However, their algorithm requires the total number of edges and nodes in the graph stream of interest is known in advance. Lim and Kang [30] present an algorithm for counting local triangles based on sampling edges with a fixed probability, which results in no guarantee on the amount of memory usage. Recently, Jung et al. [22] also develop a reservoir sampling based method FURL to estimate local triangle counts. FURL samples distinct edges from the graph stream of interest with a fixed available memory usage by combining reservoir and hash-based sampling techniques. That is, it keeps track of k distinct edges with the smallest hash values at any time. FURL reduces the computational complexity $O(k)$ in each coming edge's sampling process to $O(\log k)$ by using a Fibonacci heap [13] (a fast priority queue algorithm) to keep track of k distinct edges with the smallest hash values and using a hash table to determine whether a coming edge's duplicate has been sampled. These fast-sorting and fast-searching algorithms require a large additional memory usage. For example, besides sampled edges, the Fibonacci heap also needs a large amount of memory for storing 7 kinds of auxiliary information (e.g., pointers to child and parent). *We would like to point out that most global triangle count estimation algorithms such as [1, 44, 48] can be easily extended to approximate local triangle counts, and vice versa.*

4. ESTIMATING THE TRIANGLE COUNT FOR UNDIRECTED GRAPH STREAMS

4.1 Basic Idea of PartitionCT

For each undirected edge (u, v), we assign it a random rank $r(u, v)$ independently drawn from a $Uniform(0, 1)$ distribution. Let $\mathbb{H}$ be a family of hash functions that map undirected edges into integers $1, \ldots, k$ uniformly. Let h be a hash function randomly selected from $\mathbb{H}$. Throughout this paper we set all hash functions such as $r(u, v)$ and $h(u, v)$ regardless of the order of u and v, i.e., $r(u, v) = r(v, u)$, $h(u, v) = h(v, u)$. It can be easily achieved. For example, when u and v are two 32-bit integers, we can define $r(u, v) = \hat{r}(x)$, where x is a 64-bit integer whose low and high 32-bit values are $\min\{u, v\}$ and $\max\{u, v\}$ respectively, and function $\hat{r}(x)$ maps a 64-bit integer into a random variable drawn from a $Uniform(0, 1)$ distribution. Using hash function h, we randomly hash edges in the graph stream Π into k buckets $S[1]$, ..., $S[k]$. At any time t, a bucket holds only one edge and keeps track of the edge with the smallest rank among all edges presented so far that are hashed into the bucket. For example, when $\Pi = e_1e_1e_2e_3e_1e_4e_5e_6e_7e_8e_9e_{10}$ and $k = 5$, we randomly partition distinct edges $E = \{e_1, e_2, \ldots, e_{10}\}$ in Π into 5 subsets $E_1, \ldots, E_5$ without overlapping by hash function h. The edge subset E_i, $1 \leq i \leq 5$, is formally defined as $\{e : h(e) = i, e \in E\}$. At any time, $S[i]$ only records the edge with the smallest rank value $r(e)$ among occurred edges in E_i. We can easily find that all duplicates of an edge are hashed into the same bucket and have the same rank value. All edges have different ranks, i.e., $r(u, v) \neq r(u', v')$ for any two different edges (u, v) and (u', v'), because $r(u, v) = r(u', v')$ happens with probability 0 when $r(u, v)$ and $r(u', v')$ are two independent random variables drawn from $Uniform(0, 1)$. This leads to the following property:

THEOREM 1. *At the end of any time t, the set of edges sampled by PartitionCT is independent with the order of edges occurred before and including time t.*

Proof. At the end of any time t, a bucket $S[i]$, $1 \leq i \leq k$, samples (i.e., stores) the edge $(u^*, v^*) \in E^{(t)}$ having the smallest rank among the edges that are mapped to the bucket by hash function h. Formally, we have

$$(u^*, v^*) = \underset{(u,v) \in E^{(t)} \wedge h(u,v) = i}{\arg\min} r(u, v).$$

Clearly, (u^*, v^*) is independent with the order of edges occurred before and including time t. □

From the edges stored in buckets $S[1]$, ..., $S[k]$ at the end of time t, we estimate the triangle count $\tau^{(t)}$ as well as the number of distinct edges $n^{(t)} = |E^{(t)}|$ online, which is used for correcting the error introduced by sampling when estimating $\tau^{(t)}$.

4.2 Update Procedure

The pseudo-code of PartitionCT is shown as Algorithm 1. PartitionCT uses k buckets $S[1]$, ..., $S[k]$ to store distinct edges sampled at random. $S[1]$, ..., $S[k]$ are initialized to be empty. For any edge $e^{(t)} = (u, v)$ coming at time t, we directly set bucket $S[h(u, v)] = (u, v)$ when $S[h(u, v)]$ is empty or (u, v) has a smaller rank than the edge (u^*, v^*) currently in $S[h(u, v)]$, and discard (u, v) otherwise. Let $S^{(t)} = \{S[i] : S[i] \neq \emptyset, 1 \leq i \leq k\}$ be the set of edges sampled at the end of time t. Denote by $G_S^{(t)}$ the graph consisting of all edges in $S^{(t)}$, and $N_{u,S}^{(t)}$ the set of neighbors of u in $G_S^{(t)}$. Define $\varphi^{(t)}$ as the number of triangles in $G_S^{(t)}$. For

task 1, similar to [30,44], we use a counter φ (initialized to zero) to keep track of $\varphi^{(t)}$. That is, function **UpdateTriCounts**$(\pm, (u, v))$ in Algorithm 1 is defined as: $\varphi \leftarrow \varphi \pm |N^{(t)}_{u,S} \cap N^{(t)}_{v,S}|$. Instead of computing $\varphi^{(t)}$ over time, which requires a large computational cost, we only compute $\varphi^{(t_{\max})}$ at the end of time $t_{\max}$ for task 2.

To estimate the number of distinct edges, i.e., $n^{(t)}$, we use a counter q in Algorithm 1 to keep track of $\frac{1}{k}\sum_{j=1}^{k} 2^{-y(S[j])}$ over time, where $y(S[j])$ is defined as

$$y(S[j]) = \begin{cases} \lfloor -\log r(S[j]) \rfloor, & S[j] \neq \emptyset, \\ 0, & \text{otherwise.} \end{cases}$$

In Algorithm 1, we use a counter $\hat{n}$ to keep track of an estimate of $n^{(t)}$ over time. At the end of time t, we update $\hat{n}$ as $\hat{n} \leftarrow \hat{n} + \frac{1}{q}$ when $\lfloor -\log r(u, v) \rfloor$ is larger than $\lfloor -\log r(u^*, v^*) \rfloor$, where (u, v) is the edge coming at time t and (u^*, v^*) is the edge sampled in $S[h(u, v)]$ at the end of time $t - 1$. The idea behind our method of estimating $n^{(t)}$ will be introduced in Section 4.3.

Algorithm 1: The pseudo-code of PartitionCT. **The lines starting with "###" are only for directed graphs.**

input : edge stream Π, integer k.

$q \leftarrow 1$; $\hat{n} \leftarrow 0$; $S[1, \ldots, k] \leftarrow [\emptyset, \ldots, \emptyset]$;
foreach $(u, v, l) \in \Pi$ **do**
  $g_{\max} \leftarrow 0$;
  $j \leftarrow h(u, v)$;
  if $S[j] \neq (u, v)$ **then**
    $(u^*, v^*) \leftarrow S[j]$;
    if $S[j] == \emptyset$ *or* $r(u, v) < r(u^*, v^*)$ **then**
      if $S[j] \neq \emptyset$ **then**
        `UpdateTriCounts` $(-, (u^*, v^*))$;
        $g_{\max} \leftarrow \lfloor -\log r(u^*, v^*) \rfloor$;
      end
      $y = \lfloor -\log r(u, v) \rfloor$;
      /* y might equals $g_{\max}$ when $S[j] \neq \emptyset$ and $r(u, v) < r(u^*, v^*)$ */
      if $y > g_{max}$ **then**
        $\hat{n} \leftarrow \hat{n} + \frac{1}{q}$;
        $q \leftarrow q + \frac{1}{k}(2^{-y} - 2^{-g_{\max}})$;
      end
      $S[j] \leftarrow (u, v)$;
      ###$l_{u,v} \leftarrow l$;
      `UpdateTriCounts` $(+, (u, v))$;
    end
  end
  ###`UpdateEdgeLabel` (u, v, l);
end

4.3 Estimating the Number of Distinct Edges

We first introduce the HyperLogLog algorithm [15], which estimates the number of distinct items (i.e., cardinality) in large data streams based on the Flajolet-Martin (FM) sketch. The FM sketch consists of a list of k integers $g_1, \ldots, g_k$, initialized to 0. Let $g_j^{(t)}$, $1 \leq j \leq k$, be the value of g_j at the end of time t. When the t^{th} item $e^{(t)}$ (e.g., an undirected edge) comes, it maps the item into a pair of two random variables $h(e^{(t)})$ and $y(e^{(t)})$, where $h(e^{(t)})$ is an integer uniformly selected from $\{1, \ldots, k\}$ at random and $y(e^{(t)})$ is drawn from a $Geometric(1/2)$ distribution, supported on the set $\{0, 1, 2, \ldots\}$, i.e., $P(y(e^{(t)}) = l) = \frac{1}{2^{l+1}}$ for $l = 0, 1, 2, \ldots$. Then, $g_{h(e^{(t)})}$ is updated as

$$g^{(t)}_{h(e^{(t)})} \leftarrow \max\{g^{(t-1)}_{h(e^{(t)})}, y(e^{(t)})\}.$$

Let $q^{(t-1)}$ denote the probability of $e^{(t)}$ changing any $g_1, \ldots, g_k$. Formally, $q^{(t-1)}$ is defined as

$$q^{(t-1)} = \sum_{j=1}^{k} P(h(e^{(t)}) = j \wedge y(e^{(t)}) > g_j^{(t-1)}) = \frac{\sum_{j=1}^{k} 2^{-g_j^{(t-1)}}}{k}.$$

At the end of time t, Flajolet et al. [15] estimate the number of distinct items that have been presented in the data stream as $\breve{n}^{(t)} = \frac{\rho}{q^{(t)}}$, where ρ is an appropriate constant to correct for bias. Compared to the HyperLogLog method in [15], Ting [47] gives a more accurate estimator named *streaming HyperLogLog*, that is

$$\hat{n}^{(t)} \leftarrow \hat{n}^{(t-1)} + \frac{\mathbf{1}(g^{(t)}_{h(e^{(t)})} \neq g^{(t-1)}_{h(e^{(t)})})}{q^{(t-1)}}.$$

where $\mathbf{1}(\mathbb{P})$ is the indicator function that equals 1 when the predicate $\mathbb{P}$ is true and 0 otherwise. Next, we demonstrate that our algorithm PartitionCT estimates the number of distinct edges (i.e., $n^{(t)}$) following the same steps as the streaming HyperLoglog method. We can easily find that PartitionCT has similar data structure as HyperLogLog. It differs from HyperLogLog by storing a sampled edge (u, v) in bucket $S[h(u, v)]$ instead of using a variable $g_{h(u,v)}$ to record $y(u, v)$. Suppose we apply HyperLogLog to estimate $n^{(t)}$ of the edge stream Π. Given hash functions h and y, then we have

$$g_i^{(t)} = \max_{(u,v) \in E^{(t)} \wedge h(u,v) = i} y(u, v), \quad 1 \leq i \leq k.$$

As mentioned, at the end of time t, the sampled edge in bucket $S[i]$ is $(u^*, v^*) = \arg\min_{(u,v) \in E^{(t)} \wedge h(u,v) = i} r(u, v)$. Then, we easily have $g_i^{(t)} = y(S[i]) = y(u^*, v^*)$. In addition, PartitionCT uses a function $y(e^{(t)}) = \lfloor -\log r(e^{(t)}) \rfloor$. Then, we easily find that $y(e^{(t)}) \sim Geometric(1/2)$ because $r(e^{(t)}) \sim Uniform(0, 1)$. Based on the above observations, we easily find that the method of PartitionCT estimating $\hat{n}^{(t)}$ in Algorithm 1 is equivalent to streaming HyperLoglog [47]. Last, we would like to point out that **PartitionCT does not require any extra memory to store** $g_i^{(t)}$, because $g_i^{(t)}$ can be directly computed based on the sampled edge in bucket $S[i]$ at the end of time t. Define

$$Q^{(t)} = \{q^{(l-1)} : g^{(l)}_{h(e^{(l)})} \neq g^{(l-1)}_{h(e^{(l)})}, 1 \leq l \leq t\},$$

and $q^{(t)}_{\min} = \min_{q \in Q^{(t)}} q$. Then, we have

THEOREM 2. *[47] For any $t > 0$, the expectation and variance of $\hat{n}^{(t)}$ given $Q^{(t)}$ are*

$$\mathbb{E}(\hat{n}^{(t)} | Q^{(t)}) = n^{(t)},$$

$$Var(\hat{n}^{(t)} | Q^{(t)}) = \sum_{q \in Q^{(t)}, q \neq q^{(t)}_{min}} \frac{1 - q}{q^2} + \frac{1}{q^{(t)}_{min}}.$$

The variance of $\hat{n}^{(t)}$ is $Var(\hat{n}^{(t)}) \approx \frac{(n^{(t)})^2}{1.4426k}$ when $k \geq 128$.

The above theorem shows that $\hat{n}$ is an unbiased estimate of n, and its estimation error decreases as k increases. For example, when $k \geq 10^6$, we easily have $P(|\frac{\hat{n}}{n} - 1| \geq 0.01) \leq \frac{1}{144.26}$ from Chebyshev's inequality [33].

4.4 Estimating the Number of Triangles

For any $1 \leq j \leq k$ different edges $e_1, \ldots, e_j \in E^{(t)}$, we first compute the probability of PartitionCT sampling these edges at the end of time t, i.e. $P(e_1, \ldots, e_j \in S^{(t)} | e_1, \ldots, e_j \in E^{(t)}, n^{(t)})$. In what follows we drop two conditions $e_1, \ldots, e_j \in E^{(t)}$ and $n^{(t)}$ for brevity. To compute $P(e_1, \ldots, e_j \in S^{(t)})$, we first define

$$\gamma_j^{(t)} = \frac{k(k-1)\cdots(k-j+1)}{n^{(t)}(n^{(t)}-1)\cdots(n^{(t)}-j+1)}, \tag{1}$$

$$\beta_j^{(t)} = \gamma_j^{(t)} \left(1 - \sum_{i=1}^{j} (-1)^{i-1} \binom{j}{i} \left(1 - \frac{i}{k}\right)^{n^{(t)}}\right). \tag{2}$$

Then we have

THEOREM 3. *For $1 \leq j \leq k$ different edges $e_1, \ldots, e_j \in E^{(t)}$, PartitionCT samples them at the end of time t with probability*

$$P(e_1, \ldots, e_j \in S^{(t)}) = \beta_j^{(t)}.$$

Proof. For any j different edges $e_1, \ldots, e_j$ in $E^{(t)}$, there exist $k(k-1)\cdots(k-j+1)$ different ways to select j different buckets $S[c_1], \ldots, S[c_j]$ to sample these edges, i.e., $S[c_1] = e_1 \wedge \cdots \wedge S[c_j] = e_j$, where $c_1, \ldots, c_j \in \{1, \ldots, k\}$ are different to each other. Thus, we have

$$\begin{aligned} &P(e_1, \ldots, e_j \in S^{(t)}) \\ &= \sum_{c_1,\ldots,c_j \in \{1,\ldots,k\}} P(S[c_1] = e_1 \wedge \cdots \wedge S[c_j] = e_j). \end{aligned} \tag{3}$$

For any j different buckets $S[c_1], \ldots, S[c_j]$, we first compute the probability that none of them is empty at the end of time t as

$$\begin{aligned} &P(S[c_1] \neq \emptyset \wedge \cdots \wedge S[c_j] \neq \emptyset) \\ =&1 - P(S[c_1] = \emptyset \vee \cdots \vee S[c_j] = \emptyset) \\ =&1 - \sum_{i=1}^{j} (-1)^{i-1} \sum_{1 \leq b_1 < \cdots < b_i \leq j} P(S[c_{b_1}] = \cdots = S[c_{b_i}] = \emptyset) \\ =&1 - \sum_{i=1}^{j} (-1)^{i-1} \binom{j}{i} \left(1 - \frac{i}{k}\right)^{n^{(t)}}. \end{aligned}$$

The second last equation is obtained by the inclusion-exclusion principle, and the last equation holds because $P(S[c_{b_1}] = \cdots = S[c_{b_i}] = \emptyset) = \left(1 - \frac{i}{k}\right)^{n^{(t)}}$.

Each edge is hashed into a bucket at random, and the ranks of edges are also random variables and different to each other. Based on these properties, we can easily find that any j different edges in $E^{(t)}$ have the same chance to reside in nonempty buckets $S[c_1]$, ..., $S[c_j]$ at the end of time t. There exist $n^{(t)}(n^{(t)}-1)\cdots(n^{(t)}-j+1)$ ways to select j *different* edges from all $n^{(t)}$ edges in $E^{(t)}$ and put them into j *different* buckets $S[c_1], \ldots, S[c_j]$. Thus, we easily have the following equation at the end of time t.

$$\begin{aligned} &P(S[c_1] = e_1 \wedge \cdots \wedge S[c_j] = e_j | S[c_1] \neq \emptyset \wedge \cdots \\ &\wedge S[c_j] \neq \emptyset) = \frac{1}{n^{(t)}(n^{(t)}-1)\cdots(n^{(t)}-j+1)}. \end{aligned} \tag{4}$$

Then, we have

$$\begin{aligned} &P(S[c_1] = e_1 \wedge \cdots \wedge S[c_j] = e_j) \\ &= \frac{1 - \sum_{i=1}^{j} (-1)^{i-1} \binom{j}{i} \left(1 - \frac{i}{k}\right)^{n^{(t)}}}{n^{(t)}(n^{(t)}-1)\cdots(n^{(t)}-j+1)}. \end{aligned} \tag{5}$$

From Eqs. (3) and (5), we have $P(e_1, \ldots, e_j \in S^{(t)}) = \beta_j^{(t)}$. □

From Theorem 3, we easily find that a triangle in $G^{(t)}$ is sampled by PartitionCT at the end of time t with probability $\beta_3^{(t)}$. We can easily find that $\beta_3^{(t)} \approx \gamma_3^{(t)}$ and $\beta_3^{(t)} \approx 1$ for $n^{(t)} \gg k$ and $n^{(t)} \ll k$ respectively. Later in Section 4.7, we show that PartitionCT samples triangles with a larger probability than the state-of-the-art method FURL [22] under the same memory usage, which results in a smaller estimation error. We easily find that $\mathbb{E}(\varphi^{(t)}) = \beta_3^{(t)} \tau^{(t)}$. Thus, we can approximate $\beta_3^{(t)}$ as $\hat{\beta}_3^{(t)}$ by substituting $n^{(t)}$ with $\hat{n}^{(t)}$ in Eq. (2), and then estimate $\tau^{(t)}$ as $\hat{\tau}^{(t)} = \frac{\varphi^{(t)}}{\hat{\beta}_3^{(t)}}$. However, it is difficult to bound the error of $\hat{\beta}_3^{(t)}$. To solve this challenge, at the end of time t, we compute $C^{(t)} = \{c : S[c] \neq \emptyset\}$, the set of non-empty buckets' IDs, and use this knowledge to build a more accurate and simpler probabilistic sampling model of PartitionCT. For example, when $k = 10,000$ and there exist $|C^{(t)}| = 9,000$ non-empty buckets at the end of time t, our method PartitionCT can be simply modeled as a method of sampling $9,000$ distinct edges from all occurred edges before and including time t at random. Formally, we have

THEOREM 4. *At the end of time t, the probability of PartitionCT sampling different edges $e_1, \ldots, e_j \in E^{(t)}$ given $C^{(t)}$ is*

$$P(e_1, \ldots, e_j \in S^{(t)} | C^{(t)}) = \gamma_{j,C}^{(t)},$$

where $\gamma_{j,C}^{(t)}$ is defined as

$$\gamma_{j,C}^{(t)} = \frac{|C^{(t)}|(|C^{(t)}|-1)\cdots(|C^{(t)}|-j+1)}{n^{(t)}(n^{(t)}-1)\cdots(n^{(t)}-j+1)}. \tag{6}$$

Proof. From Eq. (4), we easily have

$$\begin{aligned} &P(e_1, \ldots, e_j \in S^{(t)} | C^{(t)}) \\ &= \sum_{c_1,\ldots,c_j \in \{1,\ldots,k\}} P(S[c_1] \neq \emptyset \wedge \cdots \wedge S[c_j] \neq \emptyset | C^{(t)}) \times \\ &P(S[c_1] = e_1 \wedge \cdots \wedge S[c_j] = e_j | S[c_1] \neq \emptyset \wedge \cdots \wedge S[c_j] \neq \emptyset) \\ &= \gamma_{j,C}^{(t)}. \end{aligned}$$ □

From the above theorem, we estimate $\tau^{(t)}$ as

$$\hat{\tau}_C^{(t)} = \frac{\varphi^{(t)}}{\hat{\gamma}_{3,C}^{(t)}},$$

where $\hat{\gamma}_{3,C}^{(t)}$ is an estimate of $\gamma_{3,C}^{(t)}$ by substituting $n^{(t)}$ with $\hat{n}^{(t)}$ in Eq. (6).

4.5 Error Analysis

To compute the error of our estimate $\hat{\tau}_C^{(t)}$, we first define a variable $\tau_C^{(t)}$

$$\tau_C^{(t)} = \frac{\varphi^{(t)}}{\gamma_{3,C}^{(t)}},$$

and derive its expectation and variance given $C^{(t)}$, which are shown in the following theorem.

THEOREM 5. *Define $\vartheta_{3,C}^{(t)} = \frac{1}{\gamma_{3,C}^{(t)}} - 1$, $\vartheta_{5,C}^{(t)} = \frac{\gamma_{5,C}^{(t)}}{(\gamma_{3,C}^{(t)})^2} - 1$, and $\vartheta_{6,C}^{(t)} = \frac{\gamma_{6,C}^{(t)}}{(\gamma_{3,C}^{(t)})^2} - 1$. Then, we have*

$$\mathbb{E}(\tau_C^{(t)} | C^{(t)}) = \tau^{(t)},$$

$$Var(\tau_C^{(t)} | C^{(t)}) = \tau^{(t)} \vartheta_{3,C}^{(t)} + 2\zeta^{(t)} \vartheta_{5,C}^{(t)} + 2\eta^{(t)} \vartheta_{6,C}^{(t)}, \tag{7}$$

where $\zeta^{(t)}$ is the number of unordered pairs of distinct triangles in $\Delta^{(t)}$ sharing an edge, and $\eta^{(t)} = \frac{1}{2}\tau^{(t)}(\tau^{(t)}-1) - \zeta^{(t)}$ is the number of unordered pairs of distinct triangles in $\Delta^{(t)}$ sharing no edge.

Proof. For a triangle σ in $G^{(t)}$, let $\zeta_\sigma^{(t)}$ be a random variable that equals 1 when all three edges of σ are sampled (i.e., stored in three different buckets) at the end of time t and 0 otherwise. From Theorem 4, we have $\mathbb{E}(\zeta_\sigma^{(t)}|C^{(t)}) = \gamma_{3,C}^{(t)}$. Thus, we obtain

$$\mathbb{E}(\tau_C^{(t)}|C^{(t)}) = \mathbb{E}\left(\frac{\sum_{\sigma\in\Delta^{(t)}}\zeta_\sigma^{(t)}}{\gamma_{3,C}^{(t)}}|C^{(t)}\right) = \tau^{(t)}.$$

We compute the variance of $\tau_C^{(t)}$ given $C^{(t)}$ as

$$\begin{aligned}
&\mathrm{Var}(\tau_C^{(t)}|C^{(t)}) = \mathrm{Var}\left(\frac{\sum_{\sigma\in\Delta^{(t)}}\zeta_\sigma^{(t)}}{\gamma_{3,C}^{(t)}}|C^{(t)}\right)\\
&= \frac{\sum_{\sigma,\sigma^*\in\Delta^{(t)}}\mathrm{Cov}(\zeta_\sigma^{(t)},\zeta_{\sigma^*}^{(t)}|C^{(t)})}{(\gamma_{3,C}^{(t)})^2}\\
&= \frac{\sum_{\sigma\in\Delta^{(t)}}\mathrm{Var}(\zeta_\sigma^{(t)}|C^{(t)})}{(\gamma_{3,C}^{(t)})^2} +\\
&\frac{\sum_{\sigma,\sigma^*\in\Delta^{(t)},\sigma\neq\sigma^*}\mathbb{E}(\zeta_\sigma^{(t)}\zeta_{\sigma^*}^{(t)}|C^{(t)}) - \mathbb{E}(\zeta_\sigma^{(t)}|C^{(t)})\mathbb{E}(\zeta_{\sigma^*}^{(t)}|C^{(t)})}{(\gamma_{3,C}^{(t)})^2}.
\end{aligned}$$

From Theorem 4, we easily have

$$\mathrm{Var}(\zeta_\sigma^{(t)}|C^{(t)}) = \gamma_{3,C}^{(t)} - (\gamma_{3,C}^{(t)})^2,$$

$$\mathbb{E}(\zeta_\sigma^{(t)}|C^{(t)})\mathbb{E}(\zeta_{\sigma^*}^{(t)}|C^{(t)}) = (\gamma_{3,C}^{(t)})^2,$$

$$\mathbb{E}(\zeta_\sigma^{(t)}\zeta_{\sigma^*}^{(t)}|C^{(t)}) = \begin{cases}\gamma_{5,C}^{(t)}, & \text{triangles } \sigma \text{ and } \sigma^* \text{ share one edge,}\\ \gamma_{6,C}^{(t)}, & \text{triangles } \sigma \text{ and } \sigma^* \text{ share no edge.}\end{cases}$$

Recalling the definition of $\vartheta_{3,C}^{(t)}$, $\vartheta_{5,C}^{(t)}$, $\vartheta_{6,C}^{(t)}$, $\zeta^{(t)}$, and $\eta^{(t)}$, we easily obtain Eq. (7) based on the above equations. □

THEOREM 6. *For any $\epsilon > 0$ and $\varepsilon > 0$, define*

$$\epsilon_3 = (1+\epsilon)\cdot\frac{(1+\epsilon)n^{(t)}-1}{n^{(t)}-1}\cdot\frac{(1+\epsilon)n^{(t)}-2}{n^{(t)}-2} - 1,$$

$$\epsilon^* = \epsilon_3 + \varepsilon + \epsilon_3\varepsilon.$$

Then, we have

$$\begin{aligned}
&P(|\frac{\hat{\tau}_C^{(t)}}{\tau^{(t)}} - 1| < \epsilon^*|C^{(t)}, Q^{(t)})\\
&> 1 - \frac{Var(\hat{n}^{(t)}|Q^{(t)})}{(\epsilon n^{(t)})^2} - \frac{Var(\tau_C^{(t)}|C^{(t)})}{(\varepsilon\tau^{(t)})^2},
\end{aligned}$$

where $Var(\hat{n}^{(t)}|Q^{(t)})$ and $Var(\tau_C^{(t)}|C^{(t)})$ are given in Theorems 2 and 5 respectively. When $n^{(t)} \gg 1$, and ϵ and ε are close to 0, we have $\epsilon^* \approx \varepsilon + 3\epsilon$.

Proof. Theorem 4 indicates that $P(S^{(t)}|C^{(t)})$ is independent with $Q^{(t)}$, so we have $P(\tau^{(t)}|C^{(t)}, Q^{(t)}) = P(\tau^{(t)}|C^{(t)})$. From Chebyshev's inequality [33], then we have

$$\begin{aligned}
P(|\frac{\tau_C^{(t)}}{\tau^{(t)}} - 1| < \varepsilon|C^{(t)}, Q^{(t)}) &> 1 - \frac{\mathrm{Var}(\tau_C^{(t)}|C^{(t)}, Q^{(t)})}{(\varepsilon\tau^{(t)})^2}\\
&= 1 - \frac{\mathrm{Var}(\tau_C^{(t)}|C^{(t)})}{(\varepsilon\tau^{(t)})^2}. \quad (8)
\end{aligned}$$

Let $s_3^{(t)} = \frac{\hat{n}^{(t)}(\hat{n}^{(t)}-1)(\hat{n}^{(t)}-2)}{n^{(t)}(n^{(t)}-1)(n^{(t)}-2)}$. We easily find that $\hat{\tau}_C^{(t)} = \tau_C^{(t)}s_3^{(t)}$. When $\hat{n}^{(t)} \approx n^{(t)}$, we have $s_3^{(t)} \approx 1$ and $\hat{\tau}_C^{(t)} \approx \tau_C^{(t)}$. To get a confidence interval of $\hat{\tau}_C^{(t)}$ with respect to $\tau_C^{(t)}$, we first bound the value of $s_3^{(t)} - 1$. From Chebyshev's inequality [33], we have

$$\begin{aligned}
P(|\frac{\hat{n}^{(t)}}{n^{(t)}} - 1| < \epsilon|C^{(t)}, Q^{(t)}) &> 1 - \frac{\mathrm{Var}(\hat{n}^{(t)}|C^{(t)}, Q^{(t)})}{(\epsilon n^{(t)})^2}\\
&= 1 - \frac{\mathrm{Var}(\hat{n}^{(t)}|Q^{(t)})}{(\epsilon n^{(t)})^2},
\end{aligned}$$

where the last equation holds because $\hat{n}^{(t)}$ only depends on the values of elements in $Q^{(t)}$. When $|\frac{\hat{n}^{(t)}}{n^{(t)}} - 1| < \epsilon$, we easily have $-\epsilon_3 < s_3^{(t)} - 1 < \epsilon_3$. Thus, we obtain

$$P(-\epsilon_3 < s_3^{(t)} - 1 < \epsilon_3|C^{(t)}, Q^{(t)}) > 1 - \frac{\mathrm{Var}(\hat{n}^{(t)}|Q^{(t)})}{(\epsilon n^{(t)})^2}. \quad (9)$$

Since $1 - \epsilon^* < (1-\varepsilon)(1-\epsilon_3)$, we have

$$\begin{aligned}
&P(|\frac{\hat{\tau}_C^{(t)}}{\tau^{(t)}} - 1| < \epsilon^*|C^{(t)}, Q^{(t)})\\
&= P(1 - \epsilon^* < \frac{\tau_C^{(t)}s_3^{(t)}}{\tau^{(t)}} < 1 + \epsilon^*|C^{(t)}, Q^{(t)})\\
&> P((1-\varepsilon)(1-\epsilon_3) < \frac{\tau_C^{(t)}s_3^{(t)}}{\tau^{(t)}} < (1+\epsilon)(1+\epsilon_3)|C^{(t)}, Q^{(t)})\\
&> P(|\frac{\tau_C^{(t)}}{\tau^{(t)}} - 1| < \varepsilon \wedge |s_3^{(t)} - 1| < \epsilon_3|C^{(t)}, Q^{(t)}).
\end{aligned}$$

From the inclusion-exclusion principle and inequalities (8)-(9), we have

$$\begin{aligned}
&P(|\frac{\tau_C^{(t)}}{\tau^{(t)}} - 1| < \varepsilon \wedge |s_3^{(t)} - 1| < \epsilon_3|C^{(t)}, Q^{(t)})\\
&= P(|\frac{\tau_C^{(t)}}{\tau^{(t)}} - 1| < \varepsilon|C^{(t)}, Q^{(t)}) + P(|s_3^{(t)} - 1| < \epsilon_3|C^{(t)}, Q^{(t)})\\
&- P(|\frac{\tau_C^{(t)}}{\tau^{(t)}} - 1| < \varepsilon \vee |s_3^{(t)} - 1| < \epsilon_3|C^{(t)}, Q^{(t)})\\
&> 1 - \frac{\mathrm{Var}(\hat{n}^{(t)}|Q^{(t)})}{(\epsilon n^{(t)})^2} - \frac{\mathrm{Var}(\tau_C^{(t)}|C^{(t)})}{(\varepsilon\tau^{(t)})^2}.
\end{aligned}$$

□

4.6 Memory and Computational Complexities

Similar to [22, 30, 44], we use a hash table to store each pair of u and $N_{u,S}^{(t)}$ to speed up the calculation of $N_{u,v,S}^{(t)} = N_{u,S}^{(t)} \cap N_{v,S}^{(t)}$. Besides $S[1], \ldots, S[k]$, PartitionCT also needs about $\frac{2}{l_f}$ times more memory for the hash table with load factor l_f, which is usually set to 0.7. Next, we discuss the computational complexity of PartitionCT. To determine whether to sample a coming edge (u, v), PartitionCT only requires to compute $h(u, v)$ and the ranks of both (u, v) and the edge currently in bucket $S[h(u, v)]$, so the computational complexity in each edge's sampling process is $O(1)$.

For task 1, we also need to update the triangle counter when sampling an edge. Suppose the t^{th} edge in $G^{(t)}$ is $e^{(t)} = (u, v)$, and $d_u^{(t)}$ and $d_v^{(t)}$ edges presented so far are incident to nodes u and v respectively. Then, $d_u^{(t)}k/n^{(t)}$ and $d_v^{(t)}k/n^{(t)}$ sampled edges are expected to be incident to u and v respectively. Therefore, the computational complexity of updating the triangle counter is $O\left((d_u^{(t)} + d_v^{(t)})k/n^{(t)}\right)$ when inserting $e^{(t)}$ into $S[h(u, v)]$. We compute the number of insertions by the end of time t as: When $l-1$ distinct edges were presented so far, the probability of sampling the l^{th} distinct edge is $\frac{k}{l}(1-(1-\frac{1}{k})^l)$. Thus, the expected number of insertions is at most $\sum_{l=1}^{n^{(t)}} k/l \approx k \ln n^{(t)}$. For task 2, we compute the number of triangles in a graph consisting of k sampled edges *only* at the end of t_{max}.

4.7 Discussion

PartitionCT vs MinHash Sketches. To some extent, our method PartitionCT and FURL [22] can be viewed as variants of MinHash sketches [11, 16] and inherit their properties, which have been extensively used for estimating the number of distinct items in a large data stream [4, 11, 16, 47], intersections of item sets [11], and distances between nodes in a large graph [12]. To sample k items, MinHash comes in three flavors with respect to sampling schemes: 1) **k-min sketch** [16] uses k independent random rank assignments, and it samples the data item with the smallest rank for each rank assignment; 2) **bottom-k sketch** [11] uses only one random rank assignment and samples k data items with the smallest ranks; 3) **k-partition sketch** [16] also uses only one random rank assignment. It hashes data items into k buckets at random. Among all data items hashed into a bucket, the bucket only keeps track of the data item with the smallest rank. We can easily find that k-partition sketch is faster than k-min and bottom-k sketches. k-min sketch is prohibitive for processing high-speed data streams when k reaches to thousands or millions, because it requires to compute k rank values for processing each data item. PartitionCT (resp. FURL) extends k-partition (resp. bottom-k) sketch to estimate the number of triangles in a large graph stream.

PartitionCT vs FURL. Jung et al. [22] develop two algorithms FURL-M_B and FURL-MX_B to solve the problem studied in this paper, which directly extend TRIÉST-BASE [44] by combining reservoir and hash-based sampling techniques. We only compare PartitionCT with FURL-M_B, because it is unknown how to choose the optimal decay parameter of FURL-MX_B and their experimental results indicate that the accuracy of FURL-MX_B is slightly better (sometimes even worse) than FURL-M_B. FURL-M_B is simply called FURL in this paper. As mentioned in Section 3, FURL [22] with the same k (i.e., the maximum number of sampled edges) requires about 3 times more memory space than PartitionCT. PartitionCT samples each coming edge with a small computational complexity $O(1)$. In contrary, FURL has a high computational complexity $O(\log k)$ when deleting an edge from the Fibonacci heap used for storing sampled edges. We compute the number of edge deletions as: When $l-1 > k$ distinct edges were presented so far, we obtain the probability of sampling the l^{th} distinct edge (trigger an edge deletion) is $\frac{k}{l}$ from [22, 44]. Thus, the expected number of edge deletions is $\sum_{l=k+1}^{n^{(t)}} \frac{k}{l} \approx k \ln \frac{n^{(t)}}{k}$. Our method PartitionCT reduces the computational complexity of sampling these edges from $O(k \ln \frac{n^{(t)}}{k} \log k)$ to $O(k)$. Next, we compare the probabilities of PartitionCT and FURL sampling triangles under the same memory usage. At the end of any time t, the probability of FURL sampling a triangle in $G^{(t)}$ is $\alpha_3^{(t)} = \min\{1, \gamma_3^{(t)}\}$, where $\gamma_3^{(t)}$ has the same definition as in Eq. (1). As mentioned in Theorem 3, PartitionCT samples a triangle in $G^{(t)}$ with probability $\beta_3^{(t)}$ at the end of time t. Figure 2 shows the values of $\alpha_3^{(t)}$ and $\beta_3^{(t)}$ for different $n^{(t)}$. We observe $\alpha_3^{(t)} \approx \beta_3^{(t)}$ when $k \gg n^{(t)}$ and $k \ll n^{(t)}$. As mentioned, FURL requires several times (about 3 times) as much memory space as PartitionCT for the same k. Figure 2 shows that $\beta_3^{(t)}$ with $k = 1,000$ is 8 and 9 times larger than $\alpha_3^{(t)}$ with $k = 500$ and $k = 333$ respectively, when $n^{(t)} > 2,000$. It indicates that PartitionCT samples more edges and tends to sample more triangles than FURL under the same memory usage. This results in a smaller estimation error, which is consistent to our experimental results in Section 6.

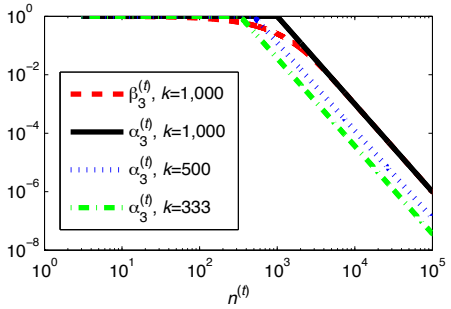


Figure 2: PartitionCT ($\alpha_3^{(t)}$) vs FURL ($\beta_3^{(t)}$): the probability of sampling a triangle in $G^{(t)}$.

5. ESTIMATING THE TRIANGLE COUNT FOR DIRECTED GRAPH STREAMS

In this section, we extend our algorithm PartitionCT to estimate directed triangle counts (i.e., $\tau^{(1,t)}, \ldots, \tau^{(7,t)}$).

Update procedure. As shown in Algorithm 1, PartitionCT can be easily extended to estimate the number of directed triangles in a large graph stream. The major difference is: the labels of edges in graph $G_S^{(t)}$ may vary with time. For example, for an edge $(u, v, \rightarrow)$ coming at time t, suppose an edge $(u, v, \leftarrow)$ has already occurred and been stored in one of buckets $S[1], \ldots, S[k]$, then $l_{u,v}$ turns from $\leftarrow$ to $\leftrightarrow$ at the end of time t. Therefore, we keep track of the labels of edges in $S[1], \ldots, S[k]$ over time, which is achieved by a function **UpdateEdgeLabel** in Algorithm 2. For $i = 1, \ldots, 7$, let $\phi^{(i,t)}$ denote the number of i^{th} type directed triangles in $G_S^{(t)}$ at the end of time t. We can easily observe that $\phi^{(i,t)}$ may change when the label of edge (u, v) changes. For task 3, similar to undirected PartitionCT, we maintain a counter $\phi^{(i)}$ to track $\phi^{(i,t)}$ dynamically. That is, function **UpdateTriCounts**$(\pm, (u, v))$ in Algorithm 1 is defined as: For each node w in $N_{u,v,S}^{(t)} = N_{u,S}^{(t)} \cap N_{v,S}^{(t)}$, we update counter $\phi^{(i)} \leftarrow \phi^{(i)} \pm 1$ when the induced subgraph consisting of nodes u, v, and w in $G_S^{(t)}$ is a triangle of the i^{th} type. Each counter $\phi^{(i)}$ is initialized as zero. Instead of computing $\phi^{(i,t)}$ over time, we only compute $\phi^{(i,t_{\text{max}})}$ at the end of t_{max} for task 4.

Directed triangle count estimation. For $i = 1, \ldots, 7$, we define $\varphi^{(i,t)}$ as the number of directed triangles of the i^{th} type in $S^{(t)}$. Then, we estimate $\tau^{(i,t)}$ as

$$\hat{\tau}_C^{(i,t)} = \frac{\varphi^{(i,t)}}{\hat{\gamma}_{3,C}^{(t)}},$$

where $\hat{\gamma}_{3,C}^{(t)}$ has the same definition as in Section 4.

Algorithm 2: UpdateEdgeLabel(u, v, l).

```
l* ← ComputeEdgeLabel(l_{u,v}, l);
if l_{u,v} ≠ l* then
    UpdateTriCounts (−, (u, v));
    l_{u,v} ← l*;
    UpdateTriCounts (+, (u, v));
end
```

Error analysis. Similar to our analysis for undirected graphs in the last section, we first define a variable

$$\tau_C^{(i,t)} = \frac{\varphi^{(i,t)}}{\gamma_{3,C}^{(t)}}$$

to compute the error of our estimate $\hat{\tau}_C^{(i,t)}$. Let $\zeta^{(i,t)}$ denote the number of unordered pairs of distinct triangles in $\Delta^{(i,t)}$ sharing an edge, and $\eta^{(i,t)} = \frac{1}{2}\tau^{(i,t)}(\tau^{(i,t)} - 1) - \zeta^{(i,t)}$ denote the number of unordered pairs of distinct triangles in $\Delta^{(i,t)}$ that do not share any edge. Then we have

THEOREM 7. *For any $1 \le i \le 7$, we have*

$$\mathbb{E}(\tau_C^{(i,t)}|C^{(t)}) = \tau^{(i,t)},$$

$$Var(\tau_C^{(i,t)}|C^{(t)}) = \tau^{(i,t)}\vartheta_{3,C}^{(t)} + 2\zeta^{(i,t)}\vartheta_{5,C}^{(t)} + 2\eta^{(i,t)}\vartheta_{6,C}^{(t)}, \quad (10)$$

where variables $\vartheta_{3,C}^{(t)}$, $\vartheta_{5,C}^{(t)}$, and $\vartheta_{6,C}^{(t)}$ have the same definitions as in Theorem 5.

THEOREM 8. *For any $1 \le i \le 7$, $\epsilon > 0$, and $\varepsilon > 0$, we have*

$$P(|\frac{\hat{\tau}_C^{(i,t)}}{\tau^{(i,t)}} - 1| < \epsilon^*|C^{(t)}, Q^{(t)}) > 1 - \frac{Var(\hat{n}^{(t)}|Q^{(t)})}{(\epsilon n^{(t)})^2} - \frac{Var(\tau_C^{(i,t)}|C^{(t)})}{(\varepsilon\tau^{(i,t)})^2},$$

where $Var(\hat{n}^{(t)}|Q^{(t)})$ and $Var(\tau_C^{(i,t)}|C^{(t)})$ are given in Theorems 2 and 7 respectively, and ϵ^* has the same definition as in Theorem 6.

We omit the proofs of Theorems 7 and 8, which are similar to the proofs of Theorems 5 and 6 respectively.

6. EVALUATION

We evaluate the performance of our method PartitionCT on several real-world graphs with up to a billion edges for the four tasks in Section 2. The algorithms are implemented in C++, and run on a computer with a Quad-Core Intel(R) Xeon(R) CPU E3-1226 v3 CPU 3.30GHz processor. We compare our method PartitionCT with four state-of-the-art methods: TRIÉST [44], MASCOT [30], FURL [22], and MG-Triangle [19]. For TRIÉST, both of its basic TRIÉST-BASE and improved TRIÉST-IMPR variants are considered. MASCOT has three variants: MASCOT-C, MASCOT-A, and MASCOT-I (MASCOT-I has no suffix and is simply called MASCOT in [30], while here we add the -I suffix to avoid confusion). [44] reveals that the variant MASCOT-A may be forced to store the entire graph stream, so we do not consider it in this paper.

6.1 Datasets

We perform our experiments on a variety of publicly available real-world directed graphs, which are summarized in Table 2. Unlike the first four graph files, a directed edge (e.g. $(u, v, \leftarrow)$) may occur more than once in the last four graph files. In the experiment on undirected graphs (Section 4.3), we discard edge directions of these graphs.

Table 2: Graph datasets used in our experiments.

graph	nodes	edges	distinct edges	triangles
Flickr [32]	2.3M	33M	23M	838M
LiveJournal [32]	4.3M	69M	43M	286M
YouTube [32]	1.1M	4.9M	3.0M	3.1M
Twitter [27]	42M	1.5B	1.2B	34.8B
wiki-Talk [29]	0.2M	1.2M	0.6M	1.9M
Actor [25]	0.4M	33M	15M	346M
Baidu [25]	0.4M	3.3M	2.4M	14M
DBLP-M [25]	1.3M	18M	5.2M	12M

6.2 Comparison Models

In this subsection, we introduce a model used to compare our method with state-of-the-art methods.

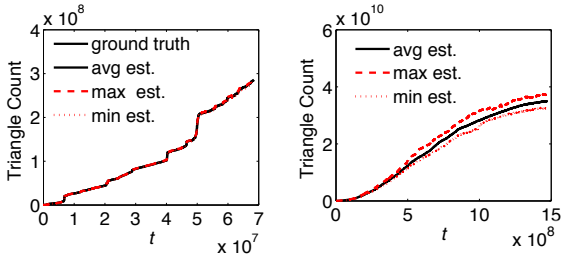
Modify MASCOT and TRIÉST to deal with edge duplicates. Except MG-Triangle and FURL, MASCOT and TRIÉST are not designed for estimating the number of triangles in a large graph stream including edge duplicates. As mentioned in Section 1, we can easily modify MOSCOT-C by using the hash-based sampling to uniformly sample distinct edges from the graph stream. Methods MASCOT-I, TRIÉST-BASE, and TRIÉST-IMPR cannot be extended in this way, therefore we simply combine them with a Bloom filter to handle edge duplicates, which is also recommended in [30].

Initialize all methods under the same memory space. As mentioned, MASCOT-C, MASCOT-I, and MG-Triangle sample edges (or wedges) with a fixed probability, so they have no guarantee on the amount of memory used when the graph stream Π comes continuously and the amount of edges is unpredictable. To make a fair comparison, we devise the following experiment. First, we run MG-Triangle for 100 times with the same values of parameters α (edge sampling probability) and β (wedge sampling probability) and compute the average number of edges and wedges sampled at the end of Π, which are denoted as M_1 and M_2 respectively. To achieve the same memory usage, we set $k = M_1 + 2M_2$ for PartitionCT. As mentioned in Section 4.7, FURL requires three times as much memory space as PartitionCT when setting the same k. Therefore, we set $k = \frac{1}{3}(M_1 + 2M_2)$ for FURL to achieve the same memory usage. In our experiments, we also compare PartitionCT and FURL with the same k. For MASCOT-C, the expected number of edges it sampled at the end of Π is $p|E^{(t_{\max})}|$, where p is the edge sampling probability specified by users in advance. Therefore, we set $p = \frac{M_1+2M_2}{|E^{(t_{\max})}|}$ for MASCOT-C. For MASCOT-I combined with a Bloom filter, we set $p = \frac{(M_1+2M_2)(1-f_{\text{Bloom}})}{|E^{(t_{\max})}|}$, where f_{Bloom} is the fraction of memory space allocated for the Bloom filter. For TRIÉST-BASE and TRIÉST-IMPR, similarly, we set both their reservoir sizes as $k = (M_1 + 2M_2)(1 - f_{\text{Bloom}})$, where k refers to the parameter M in the original work [44].

In our experiments, we set $\alpha = 0.01$, $\beta = 0.1$, and $f_{\text{Bloom}} = 0.7$ as the default values. The default memory usage of a sampling method such as PartitionCT, MASCOT, FURL, TRIÉST, and MG-Triangle is determined by these default values according to the above comparison model.

6.3 Performance on Undirected Edge Streams

Estimation of the triangle count. Figure 3 shows the evolution (over time) of the estimation computed by our method PartitionCT



(a) LiveJournal (b) Twitter

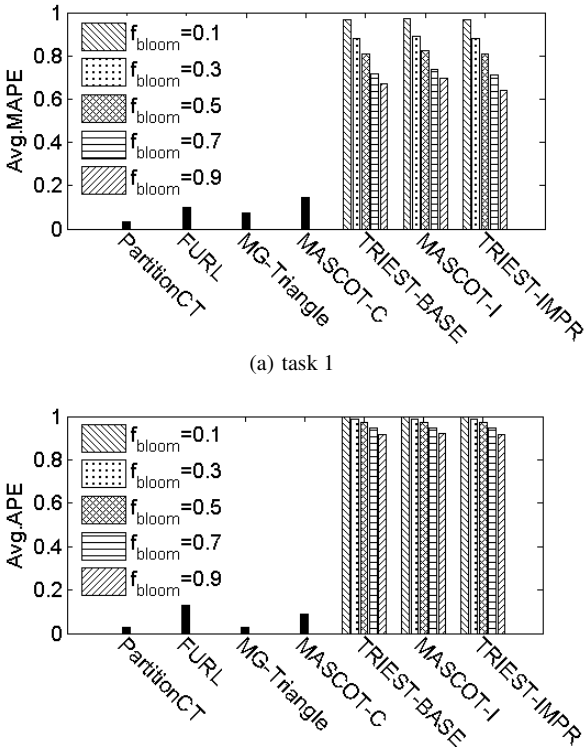
Figure 3: Estimation of the triangle count over time by our method PartitionCT with $k = 5,000,000$. The max/min/avg estimation is computed over twenty runs.

with $k = 5,000,000$. The curve of our estimation is almost indistinguishable from the ground truth of LiveJournal. For the very large graph Twitter, using less than 0.5% sampled edges, our method also has small relative variances. Due to the extensive computation, we do not provide the ground truth of $\tau^{(t)}$ over time for Twitter in figure 3. For task 2, we compute the ground truth $\tau^{(\tau_{\max})} = 3.5 \times 10^{10}$. Each of our twenty estimations has a relative error less than 0.08, i.e., $|\hat{\tau}_{\max} - \tau_{\max}| < 0.08\tau_{\max}$.

Comparison with the state-of-the-art methods. For task 1, similar to [44], we use a metric *MAPE* (Mean Average Percentage Error) to assess the accuracy of triangle count estimations over time. Formally, the MAPE of estimate $\hat{\tau}^{(t)}$ with respect to its true value $\tau^{(t)}$ is defined as $\frac{1}{t_{\max}} \sum_{t=1}^{t_{\max}} | \frac{\tau^{(t)} - \hat{\tau}^{(t)}}{\tau^{(t)}} |$. For task 2, similarly, we use a metric *APE* (Absolute Percentage Error) $| \frac{\tau^{(t_{\max})} - \hat{\tau}^{(t_{\max})}}{\tau^{(t_{\max})}} |$ to assess the accuracy of $\hat{\tau}^{(t_{\max})}$. We run each method twenty times to compute the average MAPE and APE for tasks 1 and 2 respectively.

Figure 4 shows the accuracy of our method PartitionCT in comparison with state-of-the-art methods under the same memory usage, where we set $\alpha = 0.01$ and $\beta = 0.1$ for MG-Triangle (later we will also compare PartitionCT with MG-Triangle with different α and β). On average, all these methods sample up to 1.3% of distinct edges in LiveJournal. For task 1, as shown in figure 4(a), TRIÉST-BASE, TRIÉST-IMPR, and MASCOT-I fail to provide an accurate estimation, and their estimates have average MAPEs larger than 0.6 for different f_{Bloom}. The average MAPEs of MASCOT-C and FURL are about 0.15 and 0.08 respectively. MG-Triangle further reduces the average MAPE to 0.05. PartitionCT is the best one, reducing the average MAPE to 0.03. For task 2, PartitionCT and MG-Triangle almost have the same APE, and they are several times more accurate than the other methods. Table 3 shows the average MAPE and APE of our method PartitionCT in comparison with MOSCOT-C, MG-Triangle, and FURL on more real-world graphs for both tasks 1 and 2. For task 1 (resp. task 2), PartitionCT has an average MAPE (resp. APE) up to 6, 3, and 3 times (resp. 3, 4, and 3 times) smaller than MOSCOT-C, MG-Triangle, and FURL respectively.

Figure 5 shows the performance of our method PartitionCT on LiveJournal in comparison with FURL with the same $10^6 \leq k \leq 8 \times 10^6$ and MASCOT-C with the same memory usage. To avoid storing duplicate edges, we use a hash table to speed up the computation of determining whether a coming edge has been sampled for both MASCOT-C and FURL. PartitionCT is slightly more accurate than FURL for both tasks 1 and 2. However it reduces the computational cost of FURL by up to 2 and 4 times for tasks 1 and 2 respectively. Note that here FURL (with the same k as PartitionCT)



(a) task 1

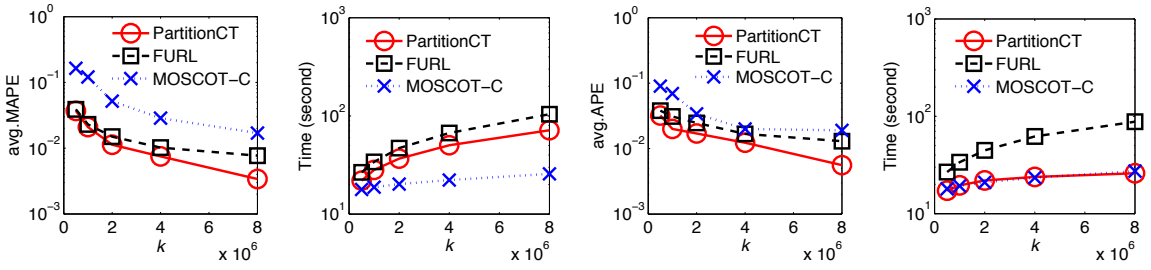
(b) task 2

Figure 4: (LiveJournal) Average MAPE of our method PartitionCT in comparison with state-of-the-art methods MASCOT, TRIÉST, FURL, and MG-Triangle with $\alpha = 0.01$ and $\beta = 0.1$ under the same memory usage (about $0.01|E^{(t_{\max})}|$). To deal with duplicate edges, we modify MASCOT-I, TRIÉST-BASE, and TRIÉST-IMPR by using a Bloom filter, and modify MASCOT-C by using hash-based sampling.

requires 3 times as much memory usage as PartitionCT. Compared to MOSCOT-C, PartitionCT is 4 times more accurate but 1.2-2.7 times slower for task 1, and is 2 times more accurate and has the same computational cost for task 2. MOSCOT-C outperforms PartitionCT and FURL in terms of computational time for task 1, but it exhibits much larger errors and has no guarantee on the amount of memory usage for evolving graph streams.

It is unknown how to compute the optimal values of α and β for MG-Triangle. Therefore, we also compare our method PartitionCT with MG-Triangle with different α and β. The result is shown in figure 6. We omit results for pairs of α and β above the black dashed line in figure 6, because MG-Triangle with any of these pairs requires more memory usage than that is required for storing the entire distinct edges. Compared to MG-Triangle with different α and β, our method PartitionCT is 5 to 50 times more accurate and 30 to 1,200 times faster than MG-Triangle for task 1. We omit similar results of task 2.

Performance vs the order of edges and duplication ratio. To assess the impact of the order of edges in the graph stream and different duplication ratios on the accuracy of our algorithm, we generate a graph by randomly repeating each edge in WikiTalk a few times drawn from a distribution $Geometric(1/b)$, where b is

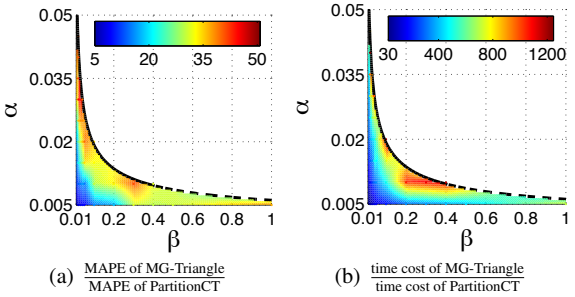


(a) task 1, average MAPE (b) task 1, time cost (c) task 2, average APE (d) task 2, time cost

Figure 5: (LiveJournal) PartitionCT vs FURL (with the same k as PartitionCT) vs MOSCOT-C (with the same memory usage as PartitionCT). In this experiment, the memory usage of FURL is 3 times larger than PartitionCT.

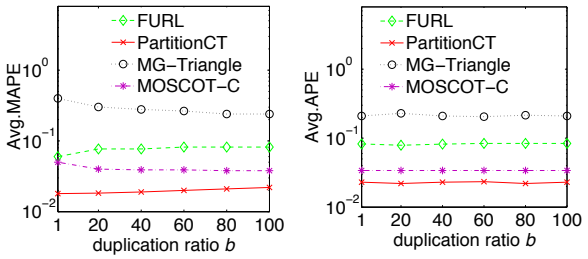
Table 3: Average MAPE and APE of triangle count estimation under the same memory usage (we set $\alpha = 0.01$ and $\beta = 0.1$ for MG-Triangle) for tasks 1 and 2 respectively.

graph	task 1				task 2			
	MOSCOT-C	MG-Triangle	FURL	PartitionCT	MOSCOT-C	MG-Triangle	FURL	PartitionCT
Flickr	0.092	0.057	0.038	**0.017**	0.054	0.022	0.058	**0.02**
LiveJournal	0.14	0.062	0.080	**0.027**	0.054	0.026	0.13	**0.025**
YouTube	0.32	0.28	0.25	**0.10**	0.21	0.21	0.34	**0.18**
Wiki-Talk	0.054	0.40	0.025	**0.019**	0.035	0.21	0.047	**0.025**
Actor	0.12	0.035	0.034	**0.021**	0.07	0.016	0.051	**0.015**
Baidu	0.037	0.36	0.027	**0.021**	0.020	0.095	0.043	**0.017**
DBLP-M	0.36	0.14	0.33	**0.10**	0.20	0.11	0.48	**0.11**



(a) $\frac{\text{MAPE of MG-Triangle}}{\text{MAPE of PartitionCT}}$ (b) $\frac{\text{time cost of MG-Triangle}}{\text{time cost of PartitionCT}}$

Figure 6: (WikiTalk, Task 1) PartitionCT vs MG-Triangle with different α and β under the same memory usage. We omit results for pairs of α and β above the black dashed line, because MG-Triangle with any of these pairs requires more memory usage than that is required for storing the entire distinct edges.



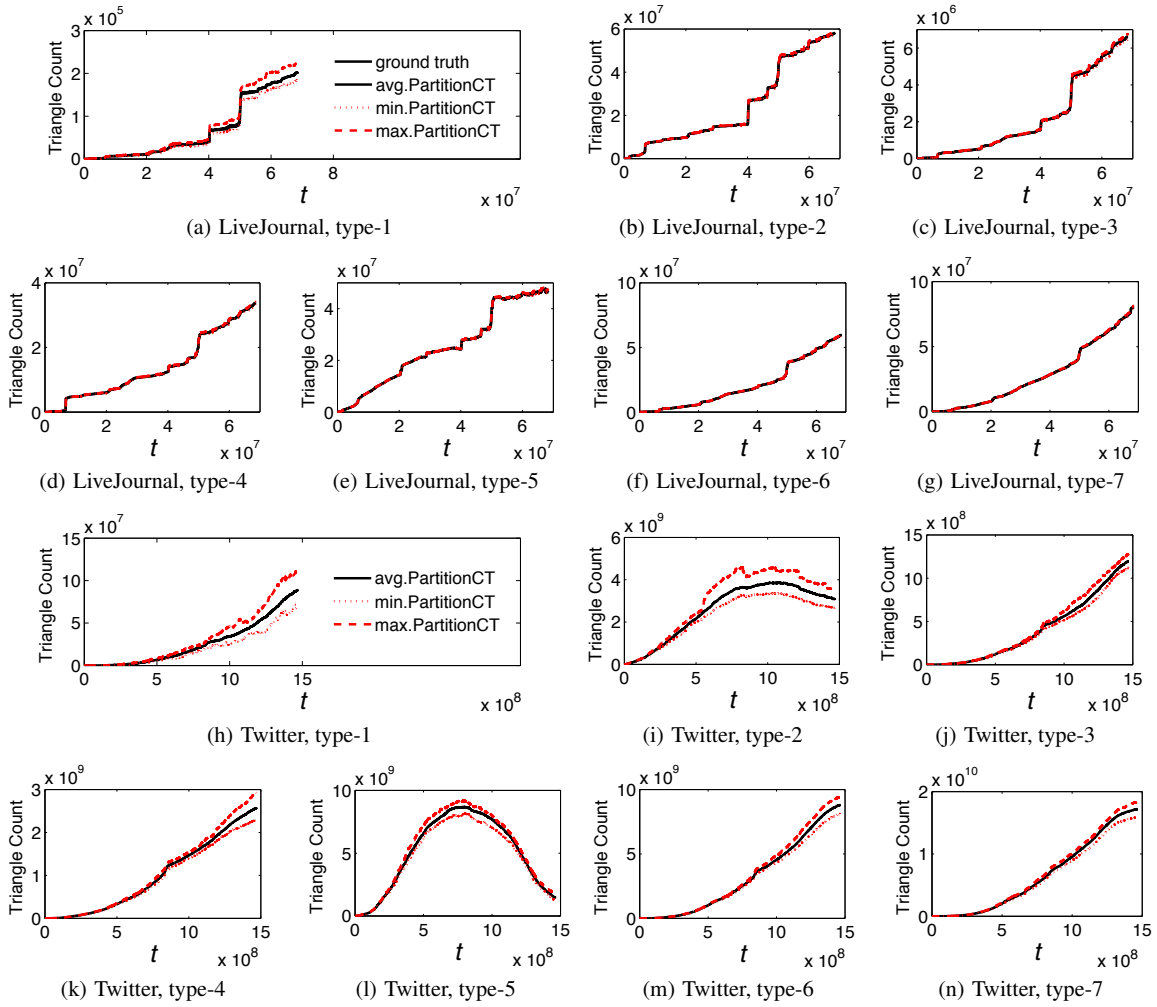
(a) task 1 (b) task 2

Figure 7: (WikiTalk) Average MAPE and APE of our method PartitionCT in comparison with MOSCOT-C, MG-Triangle, and FURL with different duplication ratio b under the same memory usage. We set $\alpha = 0.01$ and $\beta = 0.1$ for MG-Triangle.

the duplication ratio (i.e., the expected number of duplicates generated for each edge), then randomly shuffle the generated edge stream. From figure 7, we can see that our method PartitionCT is consistently more accurate than MOSCOT-C, MG-Triangle and FURL for different b.

6.4 Performance on Directed Edge Streams

Next, we evaluate the performance of our methods on directed graphs. To the best of our knowledge, no sampling method has been given to estimating the number of directed triangles from large directed graph streams. Figure 8 shows the estimations by PartitionCT on LiveJournal and Twitter. We set $k = 5,000,000$ and $k = 10,000,000$ for LiveJournal and Twitter respectively. For LiveJournal, the curve of estimation is very close to the ground truth for each type of directed triangles. The estimation of the type-1 triangle (i.e., the directed cycle) count exhibits a larger error than the other types of directed triangles, because triangles of type-1 are about 2 orders of magnitude less than triangles of the other types in LiveJournal. The MAPE and APE are 0.007 and 0.045 for tasks 3 and 4. For the graph stream of Twitter, a user with more followers (incoming edges) has smaller node IDs and all of its incoming edges occur earlier than its outgoing edges in the stream. Therefore, there exist a large number of type-2 and type-5 triangles at the beginning of the graph stream, which are mainly generated by friend nodes with the same popular followings (e.g., Lady Gaga). Friends tend to have the same popular followings and common friends on Twitter, therefore, later a large fraction of type-2 triangles transfer to triangles of type-4, and a large fraction of type-5 triangles transfer to triangles of type-6 and type-7.



(a) LiveJournal, type-1 (b) LiveJournal, type-2 (c) LiveJournal, type-3

(d) LiveJournal, type-4 (e) LiveJournal, type-5 (f) LiveJournal, type-6 (g) LiveJournal, type-7

(h) Twitter, type-1 (i) Twitter, type-2 (j) Twitter, type-3

(k) Twitter, type-4 (l) Twitter, type-5 (m) Twitter, type-6 (n) Twitter, type-7

Figure 8: Estimation of the number of directed triangles over time by PartitionCT. We set $k = 5,000,000$ and $k = 10,000,000$.

7. CONCLUSIONS AND FUTURE WORK

We develop a one-pass streaming method PartitionCT to fast approximate the number of triangles in a large graph stream including edge duplicates without storing the entire graph. To handle edge duplicates, our method PartitionCT randomly assigns different ranks for different edges and hashes edges in the graph stream into k buckets, where k is the maximum number of sampled edges specified in advance. All duplicates of an edge are hashed into the same bucket and have the same rank value. At any time, a bucket holds only one edge and keeps track of the edge with the smallest rank among all edges presented so far that are hashed into the bucket, which requires a small computational time $O(1)$ for processing each edge in the graph stream. We develop an accurate method to estimate the number of triangles based on edges stored in all buckets over time. We conduct experiments on a variety of real-world large graphs, and experimental results demonstrate that PartitionCT is several times more accurate than FURL and other state-of-the-art methods with the same memory usage, and is better than or comparable to FURL on estimation accuracy but significantly outperforms FURL in terms of running time and memory usage when setting the same maximum number of sampled edges. Our method PartitionCT is limited to handle graph streams (e.g., streams of network traffic and calling records) without edge deletions. In future, we plan to develop triangle counting methods for centralized (i.e., streams of edges observed at a single site) and distributed (i.e., streams of edges observed at multiple distributed sites) fully-dynamic graph streams including edge additions, deletions, and duplicates.

Acknowledgment

The research presented in this paper is supported in part by National Natural Science Foundation of China (U1301254, 61603290, 61602371), 111 International Collaboration Program of China, Ministry of Education&China Mobile Research Fund (MCM20160311), Natural Science Foundation of Jiangsu Province (SBK2014021758), Prospective Joint Research of Industry-Academia-Research Joint Innovation Funding of Jiangsu Province (BY2014074), Shenzhen Basic Research Grant (JCYJ20160229195940462), China Postdoctoral Science Foundation (2015M582663), Natural Science Basic Research Plan in Shaanxi Province of China (2016JQ6034).

8. REFERENCES

[1] N. Ahmed, N. Duffield, J. Neville, and R. Kompella. Graph sample and hold: A framework for big-graph analytics. In *SIGKDD*, pages 1446–1455, 2014.

[2] N. Alon, R. Yuster, and U. Zwick. Finding and counting given length cycles. *Algorithmica*, 17:354–364, 1997.

[3] S. Arifuzzaman, M. Khan, and M. Marathe. Patric: A parallel algorithm for counting triangles in massive networks. In *CIKM*, pages 529–538, 2013.

[4] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan. Counting distinct elements in a data stream. In *RANDOM*, pages 1–10, 2002.

[5] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *SODA*, pages 623–632, 2002.

[6] L. Becchetti, P. Boldi, C. Castillo, and A. Gionis. Efficient algorithms for large-scale local triangle counting. *TKDD*, 4(3):13, 2010.

[7] J. W. Berry, B. Hendrickson, R. A. LaViolette, and C. A. Phillips. Tolerating the community detection resolution limit with edge weighting. *Physical Review E*, 83(5):056119+, 2011.

[8] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13(7):422–426, 1970.

[9] L. S. Buriol, G. Frahling, S. Leonardi, A. Marchetti-Spaccamela, and C. Sohler. Counting triangles in data streams. In *PODS*, pages 253–262, 2006.

[10] H. Chun, Y. yeol Ahn, H. Kwak, S. Moon, Y. ho Eom, and H. Jeong. Comparison of online social relations in terms of volume vs. interaction: A case study of cyworld. In *IMC*, pages 57–70, 2008.

[11] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *J. Comput. Syst. Sci.*, 55(3):441–453, 1997.

[12] E. Cohen. All-distances sketches, revisited: Hip estimators for massive graphs analysis. In *PODS*, pages 2320–2334, 2014.

[13] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[14] J.-P. Eckmann and E. Moses. Curvature of co-links uncovers hidden thematic layers in the world wide web. *PNAS*, 99(9):5825–5829, 2002.

[15] P. Flajolet, E. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm. In *AOFA*, pages 127–146, 2007.

[16] P. Flajolet and G. N. Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[17] I. Giechaskiel, G. Panagopoulos, and E. Yoneki. PDTL: parallel and distributed triangle listing for massive graphs. In *ICPP*, pages 370–379, 2015.

[18] X. Hu, Y. Tao, and C.-W. Chung. Massive graph triangulation. In *SIGMOD*, pages 325–336, 2013.

[19] M. Jha, A. Pinar, and C. Seshadhri. Counting triangles in real-world graph streams: Dealing with repeated edges and time windows. In *ACSSC*, pages 1507–1514, 2015.

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

[21] H. Jowhari and M. Ghodsi. New streaming algorithms for counting triangles in graphs. In *COCOON*, pages 710–716, 2005.

[22] M. Jung, S. Lee, Y. Lim, and U. Kang. FURL: fixed-memory and uncertainty reducing local triangle counting for graph streams. *CoRR*, abs/1611.06615, 2016.

[23] U. Kang, B. Meeder, E. E. Papalexakis, and C. Faloutsos. Heigen: Spectral analysis for billion-scale graphs. *TKDE*, 26(2):350–362, 2014.

[24] J. Kim, W.-S. Han, S. Lee, K. Park, and H. Yu. Opt: A new framework for overlapped and parallel triangulation in large-scale graphs. In *SIGMOD*, pages 637–648, 2014.

[25] J. Kunegis. Handbook of network analysis [KONECT - the koblenz network collection]. *CoRR*, abs/1402.5500:1343–1350, 2014.

[26] K. Kutzkov and R. Pagh. On the streaming complexity of computing local clustering coefficients. In *WSDM*, pages 677–686, 2013.

[27] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW*, pages 591–600, 2010.

[28] M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *TCS*, 407(1-3):458–473, 2008.

[29] J. Leskovec, D. Huttenlocher, and J. Kleinberg. Predicting positive and negative links in online social networks. In *WWW*, pages 641–650, 2010.

[30] Y. Lim and U. Kang. MASCOT: memory-efficient and accurate sampling for counting local triangles in graph streams. In *SIGKDD*, pages 685–694, 2015.

[31] R. Milo, E. Al, and C. Biology. Network motifs: Simple building blocks of complex networks. *Science*, 298(5549):824–827, 2002.

[32] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Measurement and analysis of online social networks. In *IMC*, pages 29–42, 2007.

[33] M. Mitzenmacher and E. Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, New York, NY, USA, 2005.

[34] R. Pagh and F. Silvestri. The input/output complexity of triangle enumeration. In *PODS*, pages 224–233, 2014.

[35] H. Park, S. Myaeng, and U. Kang. PTE: enumerating trillion triangles on distributed systems. In *SIGKDD*, pages 1115–1124, 2016.

[36] H.-M. Park and C.-W. Chung. An efficient mapreduce algorithm for counting triangles in a very large graph. In *CIKM*, pages 539–548, 2013.

[37] H.-M. Park, F. Silvestri, U. Kang, and R. Pagh. Mapreduce triangle enumeration with guarantees. In *CIKM*, pages 1739–1748, 2014.

[38] A. Pavany, K. Tangwongsan, S. Tirthapuraz, and K.-L. Wu. Counting and sampling triangles from a graph stream. In *PVLDB*, 6(14):1870–1881, 2013.

[39] T. Schank. Algorithmic aspects of triangle-based network analysis. *Phd in Computer Science*, 2007.

[40] T. Schank and D. Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *WEA*, pages 606–609, 2005.

[41] D. Schiöberg, F. Schneider, S. Schmid, S. Uhlig, and A. Feldmann. Evolution of directed triangle motifs in the google+ OSN. *CoRR*, abs/1502.04321, 2015.

[42] C. Seshadhri, A. Pinar, N. Durak, and T. G. Kolda. Directed closure measures for networks with reciprocity. *J. Complex*

Networks, 5(1):32–47, 2017.

[43] C. Seshadhri, A. Pinar, and T. G. Kolda. Wedge sampling for computing clustering coefficients and triangle counts on large graphs. *Statistical Analysis and Data Mining*, 7(4):294–307, 2014.

[44] L. D. Stefani, A. Epasto, M. Riondato, and E. Upfal. Trièst: Counting local and global triangles in fully-dynamic streams with fixed memory size. In *SIGKDD*, pages 825–834, 2016.

[45] L. D. Stefani, A. Epasto, M. Riondato, and E. Upfal. Trièst: Counting local and global triangles in fully-dynamic streams with fixed memory size. *CoRR*, abs/1602.07424, 2016.

[46] S. Suri and S. Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, pages 607–614, 2011.

[47] D. Ting. Streamed approximate counting of distinct elements: Beating optimal batch methods. In *SIGKDD*, pages 442–451, 2014.

[48] C. E. Tsourakakis, U. Kang, G. L. Miller, and C. Faloutsos. Doulion: Counting triangles in massive graphs with a coin. In *KDD*, pages 837–846, 2009.

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

[50] H. T. Welser, E. Gleave, D. Fisher, and M. Smith. Visualizing the signatures of social roles in online discussion groups. *JoSS*, 8(2):1–32, 2007.

[51] B. Wu, K. Yi, and Z. Li. Counting triangles in large graphs by random sampling. *TKDE*, 28(8):2013–2026, 2016.

[52] Z. Yang, C. Wilson, X. Wang, T. Gao, B. Y. Zhao, and Y. Dai. Uncovering social network sybils in the wild. *TKDD*, 8(1):1–29, 2014.

[53] H. Zhang, Y. Zhu, L. Qin, H. Cheng, and J. X. Yu. Efficient triangle listing for billion-scale graphs. In *BigData*, pages 813–822, 2016.