Fast and Accurate Graph Stream Summarization

Xiangyang Gou[†,∓], Lei Zou[†,*,±], Chenxingyu Zhao[†], Tong Yang[†,*]
[†]*Peking University, China;* [∓] *Center of Data Science, Peking University;*
[±]*Beijing Institute of Big Data Research, China;*
[*] *National Engineering Laboratory for Big Data Analysis Technology and Application (PKU), China;*
{gxy1995,zoulei,dkzcxy}@pku.edu.cn, yangtongemail@gmail.com

***Abstract*—A graph stream is a continuous sequence of data items, in which each item indicates an edge, including its two endpoints and edge weight. It forms a dynamic graph that changes with every item. Graph streams play important roles in cyber security, social networks, cloud troubleshooting systems and more. Due to the vast volume and high update speed of graph streams, traditional data structures for graph storage such as the adjacency matrix and the adjacency list are no longer sufficient. However, prior art of graph stream summarization, like CM sketches, gSketches, TCM and gMatrix, either supports limited kinds of queries or suffers from poor accuracy of query results. In this paper, we propose a novel *G*raph *S*tream *S*ketch (GSS for short) to summarize the graph streams, which has linear space cost $O(|E|)$ (E is the edge set of the graph) and constant update time cost ($O(1)$) and supports most kinds of queries over graph streams with the controllable errors. Both theoretical analysis and experiment results confirm the superiority of our solution with regard to the time/space complexity and query results' precision compared with the state-of-the-art.**

***Index Terms*—graph, data stream, sketch, approximate query**

I. Introduction

A. Background and Motivations

In the era of big data, data streams propose some technique challenges for existing systems. Furthermore, the traditional data stream is modeled as a sequence of *isolated* items, and the connections between the items are rarely considered. However, in many data stream applications, the connections often play important roles in data analysis, such as finding malicious attacks in the network traffic data, mining news spreading paths among the social network. In these cases the data is organized as *graph streams*. A graph stream is an unbounded sequence of items, in which each item is a vector with at least three fields (denoted by $(\langle s, d\rangle, w)$), where $\langle s, d\rangle$ represents an edge between nodes s and d, and w is the edge weight. These data items together form a dynamic graph that changes continuously, and we call it *streaming graph* for convenience. Below we discuss three examples to demonstrate the usefulness of streaming graph problems.

Use case 1: *Network traffic.* The network traffic can be seen as a large dynamic graph, where each edge indicates the communication between two IP addresses. With the arrival of packets in the network, the network traffic graph changes rapidly and constantly. In the network traffic graph, various kinds of queries are needed, like performing node queries to find malicious attackers, or subgraph queries to locate certain topology structures in the dynamic networks.

Use case 2: *Social networks.* In a social network, the interactions among the users can form a graph. The edges between different nodes may be weighted by the frequencies of interactions. In such a graph, queries like finding the potential friends of a user and tracking the spreading path of a piece of news are often needed.

Use case 3: *Troubleshooting in data centers.* Cloud systems may need to analyze communication log stream to perform real time troubleshooting. In this situation the graph stream is the sequence of communication log entries where each entry is a description of a communication from a source machine to a destination machine. In such a graph, we may perform traversal queries to find out if massages created by a certain application on a source machine can reach a destination machine, or perform edge queries to find the detailed information of a communication log.

These streaming graphs are very large and change fast. For example, in Twitter, there are about 100 million user login data, with 500 million tweets posted per day. For another example, in large ISP or data centers [1], there could be millions of packets every second in each link. The large volume and high dynamicity make it hard to store the graph streams efficiently with traditional data structures like adjacency lists or adjacency matrices. In the context of graph streams, there are two requirements for designing a new data structure : (1) the linear space cost (2) the constant update time. To meet these two requirements, we can either apply approximated query data structures for data streams, like the CM sketch [2], the CU sketch [3] and other sketches [4], [5], or use specialized graph summarization techniques such as gSketches [6], TCM [7] and gMatrix [8]. However, existing solutions either support limited query types or have poor query accuracy. For example, CM sketches and gSketches fail to answer queries involved with topology like reachability queries, successor queries and so on. Though TCM and gMatrix can support these queries, they have poor accuracy. More details about the related work are given in Section II. In this paper, we design a novel data structure–*G*raph *S*tream *S*ketch (GSS for short), which can support most kinds of queries over streaming graphs with controllable errors in query results. Both theoretical analysis and experiment results show that the accuracy of our method outperforms state-of-the-art by orders of magnitudes.

B. Our Solution

In this paper we propose GSS, which is an approximate query data structure for graph streams with linear memory

usage, high update speed, high accuracy and supports most kinds of graph queries and algorithms like [9]–[11]. GSS can also be used in exiting distributed graph systems [12]–[15]

Like TCM, GSS uses a hash function $H(\cdot)$ to compress the streaming graph G into a smaller graph G_h which is named a *graph sketch*. Each node v in G is mapped into a hash value $H(v)$. Nodes with the same hash value are combined into one node in G_h, and the edges connected to them are also aggregated. An example of the graph stream and the graph sketch can be referred in Figure.1 and Figure.2. The compression rate can be controlled by the size of the value range of $H(\cdot)$, which we represent with M. The higher the compression rate is, the lower the accuracy is.

Different from TCM which uses an adjacency matrix to store the graph sketch G_h, GSS uses a novel data structure to store it. This data structure is specially designed for sparse graphs and stores a much bigger graph sketch with the same space. As the graph is sparse, the number of nodes is large, but each node is connected to few edges. Therefore, different from adjacency matrix which stores edges with the same source node / destination node in one row / column, we store edges with different source nodes / destination nodes in one row / column, and distinguish them with fingerprints. Each edge in the graph sketch is mapped to a bucket in the matrix depending on its endpoints, and marked with a fingerprint pair. If the mapped bucket is already occupied by other edges, we store this edge in a buffer B, which is composed of adjacency lists. With a $m \times m$ matrix we can represent a graph sketch with at most $m \times F$ nodes, where F is the size of the value range of the fingerprint (for example, a 16-bit fingerprint has $F = 65536$). On the other hand, the adjacency matrix can only store a graph sketch with at most m nodes. With a much larger graph sketch, the accuracy is also much higher compared to TCM.

In GSS, the memory cost and the update speed are greatly influenced by the size of the buffer B. As the buffer takes additional memory, and the update speed in an adjacency list is linear with its size. In order to restrict its size, we propose a technique called square hashing. In this technique each edge is mapped to multiple buckets, and stored in the first empty one among them. This enlarges the chance that an edge finds an empty bucket. Besides, a few nodes in a sparse graph may still have very high degrees. If one node emits a lot of edges, these edges have high probability to evict each other when stored in one row. To solve this problem, In square hashing edges with source node v are no longer mapped to one row, but r rows, sharing memory with other source nodes. The higher degree a node has, the more buckets it may take. It is similar in the view of columns and destination nodes. This helps to ease the congestion brought by the skewness in node degrees. Experiments show that after this modification the buffer only stores less than 0.01% of the edges in the graph stream.

The key contributions of this paper are as follows:

1) We propose GSS, a novel data structure for graph stream summarization. It has small memory usage, high update speed, and supports most kinds of queries for graphs. Moreover, it uses a combination of fingerprints and hash addresses to achieve very high accuracy.
2) We propose a technique called square hashing. It helps to decrease the buffer size in GSS, which improves update speed and reduces memory cost. It also eases the influence brought by the skewness in node degrees.
3) We define 3 graph query primitives and give details about how GSS supports them. Almost all algorithms for graphs can be implemented with these primitives.
4) We carry out theoretical analysis and extensive experiments to evaluate the performance of GSS, which show that when using 1/256 memory size of the state-of-the-art graph summarization algorithm, our algorithm still significantly outperforms it for most queries.

II. Related Work

In this part we give a brief introduction about the related works. The prior arts of graph stream summarization can be divided into two kinds. The first kind is composed of counter arrays, and stores each data item in these arrays independently, ignoring the connections between different items. They only support queries for edge weights, but do not support any queries involved with topology of the graph. This kind includes CM sketches [2], CU sketches [3], gSketches [6] and so on. The second kind supports multiple queries in the streaming graph, but suffers from poor accuracy. This kind includes TCM [7] and gMatrix [8]. Due to space limitation, in this section we focus on the second kind which is more relevant to GSS.

TCM [7] is the state-of-the-art for graph stream summarization. It is composed of an adjacency matrix that stores the compression of the streaming graph. It uses a hash function $H(\cdot)$ to compress the streaming graph $G = (V, E)$ into a smaller graph sketch G_h. For each node v in G, TCM maps it to node $H(v)$ in G_h. For each edge $e = \overrightarrow{s, d}$ in G, TCM maps it to edge $\overrightarrow{H(s), H(d)}$ in G_h. The weight of an edge in G_h is an aggregation of the weights of all edges mapped to it. A hash table that stores pairs of the hash value and the original ID can be built to retrieve the original node ID for some queries. Then TCM uses an adjacency matrix to represent the graph sketch. If we represent the size of the value range of $H(\cdot)$ with M, we need to build an $M \times M$ adjacency matrix. Each bucket in the matrix contains a counter. The weight of edge $\overrightarrow{H(s), H(d)}$ in the graph sketch is added to the counter in the bucket in row $H(s)$, column $H(d)$. When the memory is sufficient, we can also build multiple sketches with different hash functions, and report the most accurate value in queries.

In order to satisfy the demand on memory usage, the size of the adjacency matrix, $M \times M$ has to be within $O(|E|)$, which means $M \ll |V|$ for a sparse streaming graph where $\frac{|E|}{|V|}$ is usually within 10. This means the graph sketch G_h is usually much smaller than G, a lot of nodes and edges will be aggregated. As a result, the accuracy of TCM is poor.

The gMatrix [8] is a variant of TCM. Its structure is similar to TCM. But it uses reversible hash functions to generate graph sketches. It also extends TCM to more queries like edge heavy hitters and so on. However, different from the accurate hash

tables, the reversible hash functions introduce additional errors in the reverse procedure. Therefore the accuracy of gMatrix is no better than TCM, sometimes even worse.

There are some graph algorithms for statistic graph compression [16]–[18] or specific queries in graph stream processing [19]–[21]. However, they are either not suitable for high dynamic graph streams or too limited in functions. We do not introduce them in detail due to space limit.

III. Problem Definition

Definition 1: **Graph Stream**: A graph stream is an unbounded timing evolving sequence of items $S = \{e_1, e_2, e_3.....e_n\}$, where each item $e_i = (\overrightarrow{s, d}; t; w)$ indicates a directed edge from node s to node d, with wight w. The timepoint t_i is also referred as the timestamp of e_i. Thus, the edge streaming sequence S forms a dynamic directed graph $G = (V, E)$ that changes with the arrival of every item e_i, where V and E denote the set of nodes and the set of edges in the graph, respectively. We call G a *streaming graph* for convenience.

In a graph stream S, an edge $\overrightarrow{s, d}$ may appear multiple times with different timestamps. The weight of such an edge in the streaming graph G is SUM of all edge weights sharing the same endpoints. The weight w can be either positive or negative. An item with $w < 0$ means deleting a former item.

Example 1: An example of the graph stream, S, and the corresponding streaming graph G are both shown in Figure. 1. Each node has an ID that uniquely identifies itself. If an edge appears multiple times, its weights are added up as stated above.

In practice, G is usually a large, sparse and high speed dynamic graph. The large volume and high dynamicity make it hard to store graph streams using traditional data structures such as adjacency lists and adjacency matrices. The large space cost of $O(|V|^2)$ rules out the possibility of using the adjacency matrix to represent a large sparse graph. On the other hand, the adjacency list has $O(|E|)$ memory cost, which is acceptable, but the time cost of inserting an edge is $O(|V|)$, which is unacceptable due to the high speed of the graph stream.

The goal of our study is to design a *linear* space cost data structure with efficient query and update algorithms over high speed graph streams. To meet that goal, we allow some approximate query results but with small and controllable errors. However, traditional graph stream summarization approaches either cannot answer graph topology queries such as reachability queries (such as CM sketches [2] and gSketches [6]) or fail to provide accurate query results (such as TCM [7] and gMatrix [8]). Therefore, in this paper, we design a novel graph stream summarization strategy.

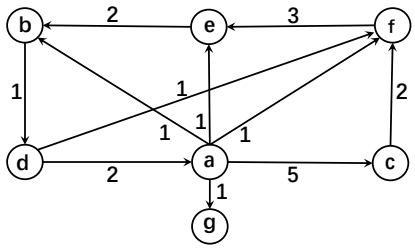
In order to give a definition of the graph stream summarization problem, First we define the *graph sketch* as follows:

Definition 2: **Graph Sketch**: a graph sketch of $G = (V, E)$ is a samller graph $G_h = (V_h, E_h)$ where $|V_h| \leqslant |V|$ and $|E_h| \leqslant |E|$. A map function $H(\cdot)$ is used to map each node in V to a node in V_h, and edge $e = \overrightarrow{s, d}$ in E is mapped to edge $\overrightarrow{H(s), H(d)}$ in E_h. The weight of an edge in E_h is the SUM of the weights of all edges mapped to it.

Formally, we define our *graph stream summarization* problem as follows.

(a, b; t_1; 1)	(a, c; t_2; 1)	(b, d; t_3; 1)	(a, c; t_4; 1)	(a, f; t_5; 1)
(c, f; t_6; 1)	(a, e; t_7; 1)	(a, c; t_8; 3)	(c, f; t_9; 1)	(d, a; t_{10}; 1)
(d, f; t_{11}; 1)	(f, e; t_{12}; 3)	(a, g; t_{13}; 1)	(e, b; t_{14}, 2)	(d, a; t_{15}; 1)

Graph stream S



Streaming graph G

Fig. 1. An example of the graph stream

Definition 3: **Graph Stream Summarization**: Given a streaming graph $G = (V, E)$, the *graph stream summarization* problem is to design a graph sketch $G_h = (V_h, E_h)$, and the corresponding data structure DS to represent G_h, where the following conditions hold:

1) There is a function $H(\cdot)$ that map nodes in V to nodes in V_h;
2) The space cost of DS is $O(|E|)$;
3) DS changes with each new arriving data item in the streaming graph and the time complexity of updating DS should be $O(1)$;
4) DS supports answering various queries over the original streaming graph G with small and controllable errors.

In the context of streaming graphs, G changes with every data item in the graph stream S, which is mapped to updating the graph sketch G_h, and conducted in data structure DS. For every new item $(\overrightarrow{s, d}; t; w)$ in S, we map edge $\overrightarrow{s, d}$ in G to edge $\overrightarrow{H(s), H(d)}$ in G_h with weight w and then insert it into G_h. Similarly, queries over G are also mapped to the same kind of queries over the graph sketch G_h. In order to support various kinds of graph queries, we first define three graph query primitives as follows, since many kinds of graph queries can be answered using these primitives.

Definition 4: **Graph Query Primitives**: Given a graph $G(V, E)$, the three graph query primitives are:

- **Edge Query**: given an edge $e = \overrightarrow{s, d}$, return its weight $w(e)$ if it exists in the graph and return -1 if not.
- **1-hop Successor Query**: given a node v, return a set of nodes that are 1-hop reachable from v, and return $\{-1\}$ if there is no such node;
- **1-hop Precursor Query**: given a node v, return a set of nodes that can reach node v in 1-hop, and return $\{-1\}$ if there is no such node.

With these primitives, we can re-construct the entire graph. We can find all the node ID in the hash table. Then by carrying

The approach in this paper can be easily extended to handle undirected graphs.

out 1-hop successor queries or 1-hop precursor queries for each node, we can find all the edges in the graph. The weight of the edges can be retrieved by the edge queries. As the graph is reconstructed, all kinds of queries and algorithms can be supported. In fact, in many situations, it is not necessary to re-construct the entire graph. We can just follow the specific algorithm and use the primitives to get the information when needed. Therefore, The data structure DS needs to support these 3 query primitives.

IV. GSS: Basic Version

In this section, we describe a conceptually simple scheme to help to illustrate intuition and benefit of our approach. The full approach, presented in Section V, is designed with more optimizations. As stated above, to produce a graph stream summarization, we first need to design a graph sketch $G_h = (V_h, E_h)$ for the streaming graph G. Initially, we use the same strategy as TCM to generate the graph sketch. We choose a hash function $H(\cdot)$ with value range $[0, M)$, then G_h is generated as following:

1) **Initialization**: Initially, $V_h = \varnothing$, and $E_h = \varnothing$.
2) **Edge Insertion**: For each edge $e = (s, d)$ in E with weight w, we compute hash values $H(s)$ and $H(d)$. If either node with ID $H(s)$ or $H(d)$ is not in V_h yet, we insert it into V_h. Then we set $H(e) = \overrightarrow{H(s), H(d)}$. If $H(e)$ is not in E_h, we insert $H(e)$ into E_h and set its weight $w(H(e)) = w$. If $H(e)$ is in E_h already, we add w to the weight.

G_h is empty at the beginning and expands with every data item in the graph stream. We can store $\langle H(v), v\rangle$ pairs with hash tables to make this mapping procedure reversible. This needs $O|V|$ additional memory, as $|V| \leqslant |E|$, the overall memory requirement is still within $O(|E|)$.

Node	a	b	c	d	e	f	g
H(v)	2	15	5	28	10	18	5

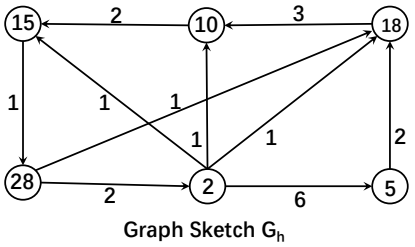


Fig. 2. An example of the graph sketch

Example 2: A graph sketch G_h for the streaming graph G in Figure 1 is shown in Figure 2. The value range of the hash function $H(\cdot)$ is $[0, 32)$. In the example, nodes c and g are mapped to the same node with ID 5 in G_h. In G_h, the weight of edge $(2, 5)$ is 6, which is the summary of the weight of edge (a, c) and edge (a, g) in G.

Obviously, the size of the value range of the map function $H(\cdot)$, which we represent with M, will significantly influence the accuracy of the summarization, especially in the 1-hop successor / precursor query primitives. In a uniform mapping with the hash function, each node in G has the probability $\frac{1}{M}$ to collide with another, which means they are mapped to the same node in G_h. When there are $|V|$ nodes, the probability that a node v does not collide with any other nodes is $(1 - \frac{1}{M})^{|V|-1} \approx e^{-\frac{|V|-1}{M}}$. In the 1-hop successor / precursor queries, if v collides with others, the query result about it will definitely have errors. Therefore we have to use a large M to maximize this probability.

Figure 3 shows the theoretical results of the relationship between M and the accuracy of the query primitives . The results are computed according to analysis in Section VI-B. (In the figure of the edge query, d_1 and d_2 means the in-degree of the source node and the out-degree of the destination node of the queried edge. In the figure of the 1-hop successor / precursor query, d_{in} and d_{out} means the in-degree and the out-degree of the queried node, respectively). The figure shows that we have to use a large M to achieve high accuracy in the query primitives, which is not possible in the prior works. According to Figure 3, only when $\frac{M}{|V|} > 200$, the accuracy ratio is larger than 80% in 1-hop successor / precursor queries. When $\frac{M}{|V|} \leq 1$, the accuracy ratio falls down to nearly 0, which is totally unacceptable.

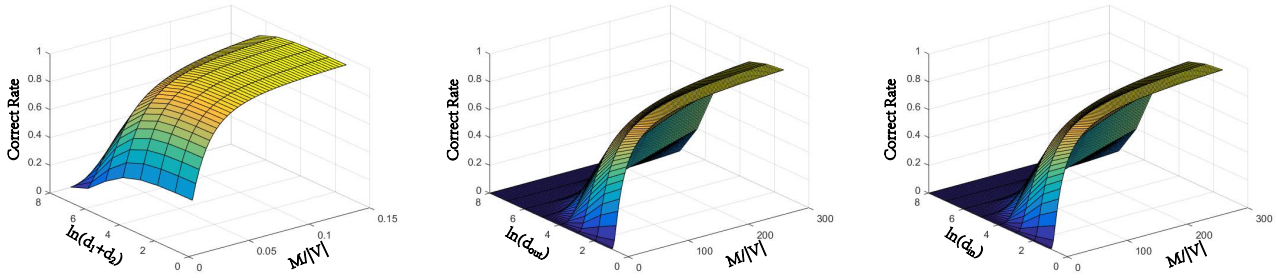
Both TCM and the gMatrix resort to an adjacency matrix to represent G_h. In this case, the matrix rank m equals to M, i.e, the value range of the map function. To keep the memory usage of the graph sketch within $O(|E|)$ (Condition 2 in Definition 3), m must be less than $\sqrt{E}$, that means $m = M < \sqrt{E} \ll |V|$ for a sparse streaming graph. According to our theoretical analysis in Figure 3, the query results' accuracy is quite low in them. Our experiments in Section VII also confirm the theoretical analysis.

Considering the above limitations, we design a novel data structure for *g*raph *s*tream *s*ummarization, called GSS.

Definition 5: **GSS**: Given a streaming graph $G = (V, E)$, we have a hash function $H(\cdot)$ with value range $[0, M)$ to map each node v in graph G to node $H(v)$ in graph sketch G_h. Then we use the following data structure to represent the graph sketch G_h:

1) GSS consists of a size $m \times m$ adjacency matrix X and an adjacency list buffer B for left-over edges.
2) For each node $H(v)$ in sketch graph G_h, we define an address $h(v)(0 \leqslant h(v) \leqslant m)$ and a fingerprint $f(v)(0 \leqslant f(v) \leqslant F)$ where $M = m \times F$ and $h(v) = \lfloor \frac{H(v)}{F} \rfloor$, $f(v) = H(v)\%F$.
3) Each edge $\overrightarrow{H(s), H(d)}$ in the graph sketch G_h is mapped to a bucket in the row $h(s)$, column $h(d)$ of the matrix X. We record $[\langle f(s), f(d)\rangle, w]$ in the corresponding bucket of the matrix, where w is the edge weight and $f(s)$, $f(d)$ are fingerprints of the two endpoints.
4) Adjacency list buffer B records all left-over edges in G_h, whose expected positions in the matrix X have been occupied by other previous inserted edges already.

The detailed analyses are given in Section VI-B



(a) Edge Query (b) 1-hop Successor Query (c) 1-hop Precursor Query

Fig. 3. Influence of M on Accuracy

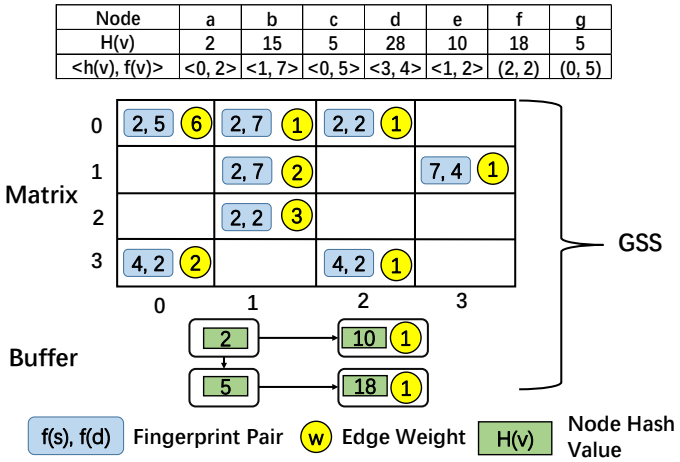


Fig. 4. An example of the basic version of data structure

When implementing a GSS for a graph stream, in order to satisfy the $O(|E|)$ memory cost requirement, we usually set $m = \alpha \times \sqrt{|E|}$, where α should be a constant approximate to 1. To achieve high accuracy, we set $M \gg |V|$. This can be achieved by setting a large F, in other words, using long fingerprints. When the memory is not sufficient, we can also set smaller M with smaller m and F, but this will decrease the accuracy.

Example 3: The basic version of GSS to store G_h in Figure 2 is shown in Figure 4. Here we set $F = 8$. The nodes in the original streaming graph and their corresponding $H(v)$, $h(v)$ and $f(v)$ are shown in the table. In this example, edge $\overrightarrow{2, 10}$ and edge $\overrightarrow{5, 18}$ in G_h are stored in the buffer because of collisions with other edges.

We discuss the insertion and primitive query operations over GSS as follows:

Edge Updating: When a new item $(s, d; t; w)$ comes in the graph stream S, we map it to an edge $\overrightarrow{H(s), H(d)}$ with weight w in graph sketch G_h. Then we find the bucket in row $h(s)$, column $h(d)$. If the bucket is empty, we store the fingerprints pair $\langle f(s), f(d)\rangle$ together with the edge weight w in the bucket. If it is not empty, we compare the fingerprint pair of this edge with the fingerprint pair $\langle f(s'), f(d')\rangle$ that is in the bucket already. If they are same, we add the weight w to the existing one; otherwise, it means this bucket has been occupied by other edges, and we store edge $\overrightarrow{H(s), H(d)}$ in the adjacency list in the buffer B. We call this kind of edges as *left-over* edges.

Graph Query Primitives: The three primitives (defined in Definition 4) are all supported with our proposed data structure GSS.

Edge Query: Given an edge query $e = \overrightarrow{s, d}$, we work as follows. We check the bucket in row $h(s)$, column $h(d)$ in the matrix. Let $\langle f(s'), f(d')\rangle$ be the fingerprint pair stored at the bucket. If $\langle f(s'), f(d')\rangle$ equals to the the fingerprint pair $\langle f(s), f(d)\rangle$ of edge $\overrightarrow{s, d}$, we return the weight in the bucket. Otherwise we search the buffer B for edge $\overrightarrow{H(s), H(d)}$ using the adjacency list. If we cannot find it in the matrix X or in the buffer B, we return -1, i.e. reporting that the edge $e = \overrightarrow{s, d}$ does not exists.

1-hop Successor Query: To find the 1-hop successors of node v, we work as follows. First, we search all buckets in row $h(v)$ of the matrix X. If a bucket in row $h(v)$ and column c has a fingerprint pair $\langle f(v), f(v_s)\rangle$, we add node $H(v_s) = c \times F + f(v_s)$ to the 1-hop successors set SS. After that, we also need to search the buffer area to find all edges with source node $H(v)$, and add its destination node to the 1-hop successors set S. We return -1 if we find no result, i.e., $|SS| = 0$. Otherwise, for each $H(s)$ in successors set SS, we obtain the original node ID by accessing the hash table.

1-hop Precursor Query: To find the 1-hop precursors of node v, we have the analogue operations with 1-hop Successor Query if we switch the columns and the rows in the matrix X. The details are omitted due to space limit.

In GSS, we store edges with different source nodes in G_h in one row of the matrix, because the graph is sparse and each node is usually connected to very few edges. We can use fingerprints to distinguish them. For example, edge $\overrightarrow{15, 28}$ and edge $\overrightarrow{10, 15}$ are all stored in row 1, but they have different source node fingerprints, namely 2 and 7, thus we know exactly which nodes they are from. It is similar in columns. Fingerprints also help us to distinguish edges when they are mapped into the same bucket. This enables us to apply a map function with a much larger value range, and generate a much larger graph sketch with the same size of matrix as TCM. With

a 4×4 matrix as in Figure 4, TCM can only support a map function with $M = 4$, and the number of nodes in the graph sketch will be no more than 4, thus the accuracy will he much poorer.

V. GSS: Augmented Algorithm

As we know, GSS has two parts: a size $m \times m$ matrix X and an adjacency list buffer B for left-over edges. Obviously, we only need $O(1)$ time to insert an edge into X, but linear time $O(|B|)$ if the edge must goto the buffer B, where $|B|$ represents the number of all left-over edges. Therefore $|B|$ influences both the memory and the time cost. In this section, we design several solutions to reduce the size of buffer B.

A. Square Hashing

In the basic version, an edge is pushed into buffer B if and only if its mapped position in the matrix X has been occupied. The most intuitive solution is to find another bucket for it. Then where to find an empty bucket? We further notice the skewness in node degrees. In the real-world graphs, node degrees usually follow the power law distribution. In other words, a few nodes have very high degrees, while most nodes have small degrees. Consider a node v that has A out-going edges in the graph sketch G_h. For a $m \times m$ adjacency matrix X in GSS (see Definition 5), there are at least $A - m$ edges that should be inserted into buffer B, as these A edges must be mapped to the same row (in X) due to the same source vertex v. These high degree nodes lead to crowed rows and result in most left-over edges in buffer B. On the other hand, many other rows are uncrowded. We have the same observation for columns of matrix X. *Is it possible to make use of the unoccupied positions in uncrowded rows / columns?* It is the motivation of our first technique, called *square hashing*.

For each node with ID $H(v) = \langle h(v), f(v) \rangle$ in G_h, we compute a sequence of hash addresses $\{h_i(v)|1 \leqslant i \leqslant r\}, (0 \leqslant h_i(v) < m)$ for it. Edge $\overrightarrow{H(s), H(d)}$ is stored in the first empty bucket among the $r \times r$ buckets with addresses

$$\{(h_i(s), h_j(d))|(1 \leqslant i \leqslant r, 1 \leqslant j \leqslant r)\}$$

where $h_i(s)$ is the row index and $h_j(d)$ is the column index. We call these buckets *mapped buckets* for convenience. Note that we consider row-first layout when selecting the first empty bucket.

Example 4: An example of square hashing is shown in Figure 5. The inserted edge is mapped to 9 buckets, and the first 2 with address $(h_1(s), h_1(d))$ and $(h_1(s), h_2(d))$ have been already occupied. Therefore the edge is inserted in the third mapped bucket. In the bucket, we store the weight, the fingerprint pair, together with an index pair $\langle 1, 3 \rangle$ which indicates the position of this bucket in the mapped buckets sequence. We will talk about the use of the index pair later.

The following issue is how to generate a *good* hash address sequence $\{h_i(v)|1 \leq i \leq r\}$ for a vertex v. There are two requirements:

Independent: For two nodes v_1 and v_2, we use P to represent the probability that $\forall 1 \leq i \leq r, h_i(v_1) = h_i(v_2)$. Then we have $P = \prod_{i=1}^{r} Pr(h_i(v_1) = h_i(v_2))$. In other words, the randomness of each address in the sequence will not be influenced by others. This requirement will help to maximize the chance that an edge finds an empty bucket among the $r \times r$ mapped buckets.

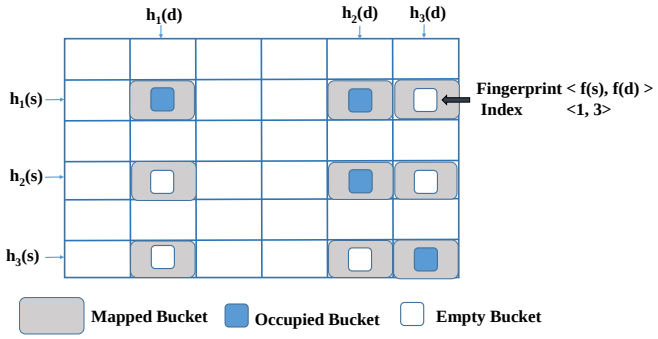


Fig. 5. The square hashing

Reversible: Given a bucket in row R and column C and the content in it, we are able to recover the representation of the edge e in the graph sketch G_h: $\overrightarrow{H(s), H(d)}$, where e is the edge in that bucket. This property is needed in the 1-hop successor query and the 1-hop precursor query. As in these queries, we need to check the potential buckets to see if they contain edges connected to the queried node v and retrieve the other end point in each qualified bucket.

To meet the above requirements, we propose to use *linear congruence method* [22] to generate a sequence of r random values $\{q_i(v)|1 \leqslant i \leqslant r\}$ with $f(v)$ as seeds. We call this sequence the linear congruential (LR) sequence for convenience. The linear congruence method is as following: select a timer a, small prime b and a module p, then

$$\begin{cases} q_1(v) = (a \times f(v) + b)\%p \\ q_i(v) = (a \times q_{i-1}(v) + b)\%p, (2 \leqslant i \leqslant r) \end{cases} \quad (1)$$

By choosing a, b and p carefully, we can make sure the cycle of the sequence we generate is much larger than r, and there will be no repetitive numbers in the sequence [22]. Then we generate a sequence of hash addresses as following:

$$\{h_i(v)|h_i(v) = (h(v) + q_i(v))\%m, 1 \leqslant i \leqslant r\} \quad (2)$$

When storing edge $\overrightarrow{H(s), H(d)}$ in the matrix, besides storing the pair of fingerprints and the edge weight, we also store an index pair (i_s, i_d), supposing that the bucket that contains this room has an address $(h_{i_s}(s), h_{i_d}(d))$. As the length of the sequence, r, is small, the length of each index will be less than 4 bits. Therefore storing such a pair will cost little.

Note that the hash sequence $\{q_i(v)|1 \leqslant i \leqslant r\}$ generated by the linear congruence method are both *independent* and *reversible*. The independence property has been proved in [8]. We show how to recover the original hash value $H(v)$ based on the $f(v)$, $h_i(v)$ and the index i as follows. First, we compute the LR sequence $\{q_i(v)\}$ with $f(v)$ following equation 4. Second we use the equation $(h(v) + q_i(v))\%m = h_i(v)$ to

compute the original hash address $h(v)$. As $h(v) < m$, the equation has unique solution. At last we use $H(v) = h(v) \times F + f(v)$ to compute $H(v)$. Given a bucket in the matrix, the fingerprint pair $(f(s), f(d))$ and the index pair (i_s, i_d) are all stored in it, and we have $h_{i_s}(s) = R$, $h_{i_d}(d) = C$, where R and C are the row index and the column index of the bucket in the matrix, respectively. Therefore we can retrieve both $H(s)$ and $H(d)$ as above.

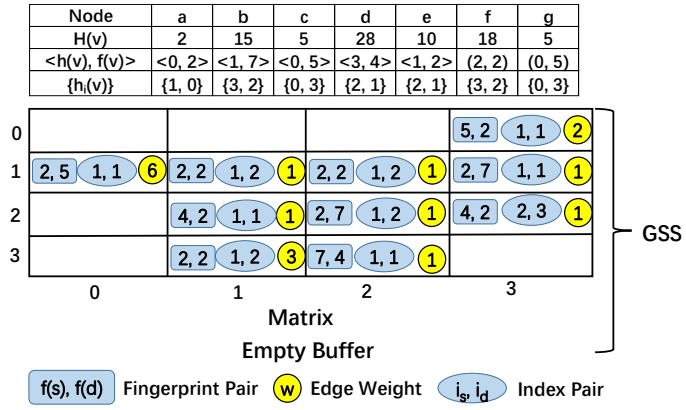


Fig. 6. An example of the modified version of data structure

Example 5: An example of the modified version is shown in Figure. 6. In the matrix we stored G_h in Figure. 2, which is a graph sketch of G in Figure.1. In this example we set $F = 8$, $m = 4$, $r = 2$, and the equation in the linger congruence method is

$$\begin{cases} q_1(v) = (5 \times f(v) + 3)\%8 \\ q_i(v) = (5 \times q_{i-1}(v) + 3)\%8, (2 \leqslant i \leqslant r) \end{cases} \tag{3}$$

Compared to the basic version, in the modified version all edges are stored in the matrix, and the number of memory accesses we need to find an edge in the matrix is within $2^2 = 4$. In fact in the example we only need one memory access to find most edges, and 2 for a few ones.

In the following, we illustrate the four basic operators in this data structure GSS.

Edge Updating: When a new item $(s, d, t; w)$ comes in the graph stream S, we map it to edge $\overrightarrow{H(s), H(d)}$ in the graph sketch G_h with weight w. Then we compute two hash address sequences $\{h_i(s)\}$ and $\{h_i(d)\}$ and check the r^2 mapped buckets with addresses $\{(h_i(s), h_j(d)) | 1 \leqslant i \leqslant r, 1 \leqslant j \leqslant r\}$ one by one. For a bucket in row $h_{i_s}(s)$ and column $h_{i_d}(d)$, if it is empty, we store the fingerprint pair $(f(s), f(d))$ and the index pair (i_s, i_d) and weights w in it, and end the procedure. If it is not empty, we check the fingerprint pair $(f(s'), f(d'))$ and the index pair (i'_s, i'_d) stored in the bucket. If the fingerprint pair and the index pair are all equal to the corresponding pairs of the new inserted edge $\overrightarrow{H(s), H(d)}$, we add w to the weights in it, and end the procedure. Otherwise it means this bucket has been occupied by other edges and we consider other hash addresses following the hash sequence. If all r^2 buckets have been occupied, we store edge $\overrightarrow{H(s), H(d)}$ with weight w in the buffer B, like the basic version of GSS.

Graph Query Primitives: The three graph query primitives are supported in the modified data structure as follows:

Edge Query: When querying an edge $e = \overrightarrow{s, d}$, we map it to edge $\overrightarrow{H(s), H(d)}$ in the graph sketch, and use the same square hashing method to find the r^2 mapped buckets and check them one by one. Once we find a bucket in row $h_{i_s}(s)$ and column $h_{i_d}(d)$ which contains the fingerprint pair $(f(s), f(d))$ and the index pair (i_s, i_d), we return its weight as the result. If we find no results in the r^2 buckets, we search the buffer for edge $\overrightarrow{H(s), H(d)}$ and return its weights. If we still can not find it, we return -1.

1-hop Successor Query: to find the 1-hop successors of node v, we map it to node $H(v)$ in G_h. Then we compute its hash address sequence according to $H(v)$, and check the r rows with index $h_i(v), (1 \leqslant i \leqslant r)$. If a bucket in row $h_{i_s}(v)$, column C contains fingerprint pair $((f(v), f(x))$ and index pair (i_s, i_d) where $f(x)$ is any integer in range $[0, F)$ and i_d is any integer in range $[1, r]$, we use $f(x)$, i_d and C to compute $H(x)$ as stated above. Then we add $H(x)$ to the 1-hop successor set SS. After searching the r rows, we also need to check the buffer to see if there are any edges with source node $H(v)$ and add their destination nodes to SS. We return -1 if we find no result, otherwise we obtain the original node ID from SS by accessing the hash table $\langle H(v), v \rangle$.

1-hop Precursor Query: to answer an 1-hop precursor query, we have the analogue operations with 1-hop Successor Query if we switch the columns and the rows in the matrix X. The details are omitted due to space limit.

After applying square hashing, the edges with source node $H(v)$ in G_h are on longer stored in a single row, but spread over r rows with addresses $h_i(v), (1 \leqslant i \leqslant r)$. Similarly, edges with destination node $H(v)$ are stored in the r different columns. These rows or columns are shared by the edges with different source nodes or destination nodes. The higher degree a node has, the more buckets its edges may take. This eases the congestion brought by the skewness in node degrees. Moreover, as each bucket has multiple mapped buckets, it has higher probability to find an empty one. Obviously, square hashing will reduce the number of *left-over edges*.

B. Further Improvements

There are some other improvements which can be implemented to GSS.

1) Mapped Buckets Sampling: In the modified version of GSS, each edge has r^2 mapped buckets. We usually set r to integers from 4 to 16. When the skewness of node degrees is serious, r can be larger. If we check all the r^2 buckets when inserting an edge, it will be time consuming. To improve the updating speed, which is very important for graph stream summarization, we can use a sampling technique to decrease the time cost. Instead of check all the r^2 buckets, we select k buckets as a sample from the mapped buckets, we call these buckets *candidate buckets* for short. For each edge we only check these k buckets in updating and queries, and the operations are the same as above. The method to select these k buckets for an edge e is also a linear congruence method.

We add the fingerprint of the source node and the destination node of e to get a seed $seed(e)$, then we compute a k length sequence as

$$\begin{cases} q_1(e) = (a \times seed(e) + b)\%p \\ q_i(e) = (a \times q_{i-1}(e) + b)\%p, (2 \leqslant i \leqslant k) \end{cases} \tag{4}$$

where a, b and p are the same integers used above. We choose the k buckets with address

$$\left\{ (h_{\lfloor \frac{q_i(e)}{r} \rfloor \%r}(s), h_{(q_i(e)\%r)}(d)) | 1 \leqslant i \leqslant k \right\} \tag{5}$$

$\{h_i(s)\}$ and $\{h_i(d)\}$ are the hash address sequence of the source node and the destination node, respectively.

2) Multiple Rooms: When the memory is sufficient, we do not need to use multiple matrices to increase accuracy as TCM, because the accuracy is already very high. Instead, in order to further decrease the buffer size, we can separate each bucket in the matrix into l segments, and each segments contains an edge, including the weight, the fingerprint pair and the index pair. We call each segment a *room* for convenience. When performing the basic operators, we use the same process as above to find the buckets we need to check, and search all the rooms in them to find qualified edges or empty rooms.

However, when the rooms in each bucket are stored separately, the speed will probably decrease. Because we can not fetch the l rooms in one memory access in most cases, and multiple memory accesses increase the time cost. As shown in Figure. 7, we separate the bucket into 3 area: the index area, the fingerprint area, and the weight area. Each area contains the corresponding parts of the l rooms. When we check this bucket to find certain edges, we can first check all the index pairs. If we find a matched index pair, we check the corresponding fingerprint pair, and if the fingerprint pair is also matched, we fetch the corresponding weight. If we do not find any matched index pair, we can just move on and do not need to check the fingerprint pairs any more. As the index pairs are very small, usually no more than 1 byte, we can fetch all the index pairs in one memory access. This will omit a lot of unnecessary memory accesses.

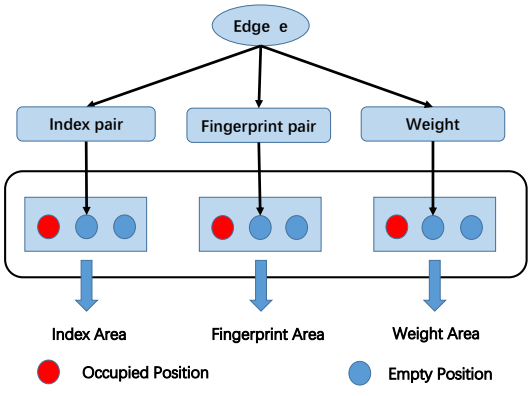


Fig. 7. Bucket Separation

VI. Analysis

A. Memory and Time Cost Analysis

As stated above, GSS has $O(|E|)$ memory cost and constant update speed. The memory cost of GSS is $O(|E_h|+|B|)$, to be precise, where $|E_h|$ is the number of edges in the graph sketch G_h and $|B|$ is the size of buffer. When we use a hash table to store the original ID, additional $O(|V|)$ memory is needed, but the overall memory cost is still $O(|E|)$. The update time cost is $O(k + \frac{|B|}{|E_h|}|B|)$, where k is the number of sampled buckets and is a small constant. When an edge is stored in the matrix, we only need to check at most k candidate buckets, which takes $O(k)$ time. Each edge has probability $\frac{|B|}{|E_h|}$ to be stored in the buffer. When it is stored in the buffer, the update takes additional $O(|B|)$ time, as the buffer is an adjacency list. In implementations the buffer stores 0 edges in most cases, which will be shown in Section VII-G. (We also carry out theoretical analysis about the buffer size, which can be seen in the technical report [23]. We omit it here because of space limitation). Therefore $\frac{|B|}{|E_h|}|B|$ is also a small constant. When it is necessary to store the ID of nodes in applications, one insertion to the hash table is needed, which also takes constant time. Overall, the update time cost is $O(1)$.

The time cost of queries is based on the algorithms we use. We consider the time cost of the primitives as an evaluation. The time cost of the edge query primitive is the same as the update, and the time cost of the 1-hop successor query and 1-hop precursor query is $O(rm + |B|)$, where m is the side length of the matrix and r is the length of the hash address sequence.

B. Accuracy Analysis

In this part we evaluate the accuracy of GSS. Before we analyze the probability of errors, we first propose the following theorem:

Theorem 1: The storage of the graph sketch G_h in the data structure of GSS is accurate. Which means for any edge $e_1 = \overrightarrow{H(s_1), H(d_1)}$ and $e_2 = \overrightarrow{H(s_2), H(d_2)}$ in G_h, the weights of them will be added up if and only if $H(s_1) = H(s_2), H(d_1) = H(d_2)$.

The proof of the theorem can be referred in the technical report [23]. This theorem means we only need to consider the procedure of mapping G to G_h, as all errors happen in this procedure. We use $\hat{P}$ to represent the probability of the following event:

Definition 6: **Edge Collision**: An edge collision means that given an edge e, there is at least one e' in G and $e' \neq e$ which satisfies $H(e) = H(e')$ in the graph sketch G_h.

We set $P = 1-\hat{P}$, and P is the main component of the error rate of all the 3 graph query primitives. In the edge query, P is just the correct rate. In the 1-hop successor query for a node v, the correct rate is $P^{|V|-d}$, where $|V|$ is the number of nodes in G, and d is the out-degree of the queried node. Because we will get a correct answer if and only if for each v' in G which is not a 1-hop successor of v, (v, v') does not collide with any existing edges, and there are totally $|V| - d$ such nodes as v'.

The 1-hop precursor query is similar. Therefore, we need to compute P to evaluate the accuracy of GSS.

C. Collision Rate

Now we show the probability that an edge e suffers from edge collision, $\hat{P}$. For $e = \overrightarrow{s, d}$ in G, we assume there are D edges with source node s or destination node d in G besides e, and there are totally $|E|$ edges in G. We represent the size of the value range of the map function $H(\cdot)$ with M.

Then we have

$$P = e^{-\frac{|E|+(M-1)\times D}{M^2}} \tag{6}$$

And $\hat{P} = 1 - P$. For an edge not adjacent to e, it will collide with e when both its source node and destination node collide with the corresponding node of e, which probability is $\frac{1}{M^2}$. For an edge adjacent to e, the collision probability increases to $\frac{1}{M}$ as one node is already the same. P is the probability that none edge collide with e, which can be computed by multiplication of probabilities. Detailed derivation is shown in the technical report [23]. In GSS we have $M = m \times F$, where m is the length of the matrix, and F is the maximum size of the fingerprints. On the other hand, in TCM the accuracy analysis is the same as GSS, but we have $M = m$. This lead to the difference on accuracy.

VII. Experimental Evaluation

In this section, we show our experimental studies of GSS. We compare GSS with TCM on the three graph query primitives: edge query , 1-hop successor query, 1-hop precursor query (VII-D) and two compound queries, node queries (VII-E) and reachability queries (VII-F). We also evaluate the size of buffer (VII-G)and update speed of GSS (VII-H). Then we further compare GSS with the state-of-the-art task specific algorithms on triangle counting and subgraph matching VII-I.

All experiments are performed on a server with dual 6-core CPUs (Intel Xeon CPU E5-2620 @2.0 GHz, 24 threads) and 62 GB DRAM memory, running Ubuntu. All algorithms including GSS and TCM are implemented in C++.

A. Data Sets

We choose three real world data sets. Details of three data sets are described as follows:

1)**web-NotreDame**.The first data set is a web graph collected from the University of Notre Dame. Nodes represent web pages and directed edges represent hyperlinks between pages. The data set contains 325729 nodes and 1497134 edges. We use the Zipfian distribution to generate weights for the edges in the data set, and insert the edges into the data structure one by one to simulate the procedure of real-world incremental updating. 2)**lkml-reply**.The second data set is a collection of communication records in the network of the Linux kernel mailing list. It contains 63399 email addresses (nodes) and 1096440 communication records (edges). Each edge is weighted by its frequency in the data set, and has a timestamp indicating the communication time. We feed the data items to the data structure according to their timestamps to simulate a graph stream. 3)**networkflow**. The third data set is a collection of network packets downloaded from a backbone router. It contains 445440480 communication records (edges) concerning 2601005 different IP addresses (nodes). Each edge is weighted by its frequency in the data set, and has a timestamp indicating the communication time. We feed the data items to the data structure according to their timestamps to simulate a graph stream.

The function we use to cumulate the edge weights is addition. In this case, TCM and GSS only have over-estimations. The codes are open sourced

B. Metrics

In this part we give a definition of the metrics we use in experiments.

Average Relative Error (ARE): ARE measures the accuracy of the reported weights in edge queries and node queries. Given a query q, the *relative error* is defined as:

$$RE(q) = \frac{\hat{f(q)}}{f(q)} - 1$$

$f(q)$ and $\hat{f(q)}$ are the real answer and the estimated value of q. When giving a query set, the *average relative error* (ARE) is measured by averaging the relative errors over all queries int it. A more accuracy data structure will have smaller ARE.

Average Precision: We use average precision as the evaluation metric in 1-hop successor queries and 1-hop precursor queries. Given such a query q, we use SS to represent the accurate set of 1-hop successors / precursors of the queried node v, and $\hat{SS}$ to represent the set we get by q. As TCM and GSS have only false positives, which means $SS \subseteq \hat{SS}$, we define the precision of q as:

$$Precision(q) = \frac{|SS|}{|\hat{SS}|}$$

Average precision of a query set is the average value of the precision of all queries in it. A more accuracy data structure will have higher *Average Precision*

True Negative Recall: It measures the accuracy of the reachability query. Because connectives of all edges are kept, there is no false negatives in TCM and GSS, which means if we can travel to d from s in the streaming graph, the query result of these data structures will be definitely yes. Therefore in experiments we use reachability query sets $Q = \{q_1, q_2, ..., q_k\}$ where $\forall q_i \in Q$, source node s and destination node d in q_i are unreachable. *True negative recall* is defined as the number of queries reported as unreachable divided by the number of all queries in Q.

Buffer Percentage: It measures buffer size of GSS. *Buffer percentage* is defined as the number of edges in the buffer divided by the total number of edges in the graph stream.

http://snap.stanford.edu/data/web-NotreDame.html
http://konect.uni-koblenz.de/networks/lkml-reply

https://github.com/Puppy95/Graph-Stream-Sketch

C. Experiments settings

In experiments, we implement two kinds of GSS with different fingerprint sizes: 12 bits and 16 bits, and vary the matrix size. We use *fsize* to represent the fingerprint size by short. We apply all improvements to GSS, and the parameters are as follows. Each bucket in the matrix contains $l = 2$ rooms. The length of the address sequences is $r = 16$, and the number of *candidate buckets* for each edge is $k = 16$. As for TCM, we apply 4 graph sketches to improve its accuracy, and allow it to use larger memory, because otherwise the gap between it and GSS will be so huge that we can hardly compare them in one figure. In edge query primitives, we allow TCM to use 8 times memory, and in other queries we implement it with 256 times memory, as its accuracy is too poor in these queries (in networkflow, we implement it with 16 times memory because of the limitation of the memory of the server). This ratio is the memory used by all the 4 sketches in TCM divided by the memory used by GSS with 16 bit fingerprints. When the size of GSS varies, the size of matrix in TCM also varies correspondingly to keep the ratio unchanged.

D. Experiments on query primitives

In this section, we evaluate the performance of GSS in the 3 basic graph query primitives: the edge query, the 1-hop precursor query and the 1-hop successor query. Figure 8, Figure 9, and Figure 10 show that ARE of edge queries and average precision of 1-hop precursor / successor queries for the data sets web-NotreDame, lkml-reply and networkflow, respectively. To reduce random error introduced by the selection of the data sample [24], the edge query set contains all edges in the graph stream, and the 1-hop precursor / successor query set contains all nodes in the graph stream. The results tell us that GSS performs much better in supporting these query primitives than TCM, especially in the 1-hop precursor / successor query primitives. In both GSS and TCM, the ARE decreases, and the precision increases with the growth of the width of the matrix. This trend is not significant in GSS as the accuracy is high and there are no errors in most experiments. Also, when the length of fingerprint becomes longer, the accuracy of GSS increases.

E. Experiments on Node Query

In this section, we evaluate the performance of GSS in estimating the accuracy of node query. A node query for a node v is to compute the summary of the weights of all edges with source node v. For each dataset, node query set contains all nodes in the graph stream. Figure 11 shows the ARE of node queries in data sets web-NotreDame, lkml-reply and networkflow, respectively. The figure shows that although we unfairly fix the ratio of memory used by TCM and GSS, GSS still can achieve better performance than TCM.

F. Experiments on Reachability Query

In this section, we evaluate the performance of GSS in supporting reachability queries. Each reachability query set Q contains 100 unreachable pairs of nodes which are randomly generated from the graph. Figure 12 shows the true negative recall of reachability query for the data sets web-NotreDame, lkml-reply and networkflow, respectively. From the figure we can see that the accuracy of GSS is much higher than TCM even when TCM uses much larger memory. The gap varies with the size of the graph. Along with increasing the memory and the length of the fingerprint [25], GSS can achieve better performance. We can also see that the accuracy of TCM is so poor that it can barely support this query.

G. Experiments on Buffer Size

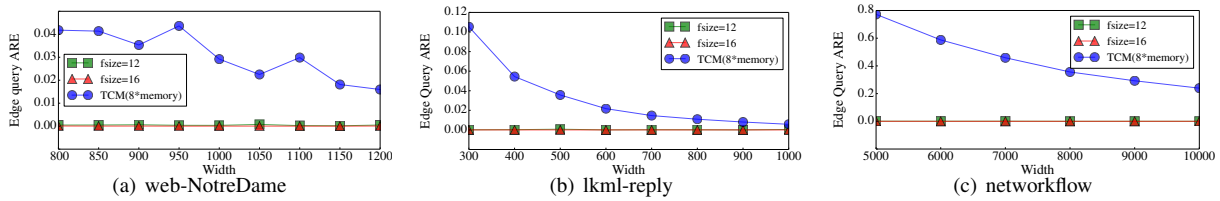
In this section, we evaluate the buffer size of GSS. Figure 13 shows the buffer percentage for networkflow. Results of other data sets can be found in the technical report [23]. The four curves in the figure represent 1)GSS with 1 room in each bucket and no square hashing. 2) GSS with 2 rooms in each bucket and no square hashing. 3) GSS with 1 room in each bucket and square hashing. 4) GSS with 2 rooms in each bucket and square hashing. The x-label, w, is the side length of the matrix for the schemes with 2 rooms in each bucket. When GSS has 1 room in each bucket, the width of the matrix is $2^{0.5}$ times larger to make the memory unchanged. The above results show that the decrement in buffer size brought by using square hashing and multiple rooms is significant, especially the square hashing. The results also show that the buffer percentage in the fully improved GSS (2 rooms each bucket, with square hashing) becomes 0 in most experiments when the matrix size is close to $|E|$. In this case, the overhead brought by the insertion failure in the matrix is nearly 0.

H. Experiment on update speed

In this section we evaluate the update speed of GSS. We compare the update speed of GSS, TCM and adjacency lists, the result is shown in Table I. The adjacency list is accelerated using a map that records the position of the list for each node. Because the update speed changes little with the matrix size, we only show the average speed here. The fingerprint size is 16-bit. TCM is still implemented with the same settings as above experiments. In each data set we insert all the edges into the data structure, repeat this procedure 100 times and calculate the average speed. The unit we use is Million Insertions per Second (Mips). From the table we can see that the speed of GSS is similar to TCM, because though more memory accesses are needed, GSS computes less hash functions. Both of them are much higher than the adjacency list. We also show the speed of GSS without candidate bucket sampling. We can see that the speed without candidate sampling is lower than the full optimized one. The gap is not very large because most edges find empty bucket in few searches.

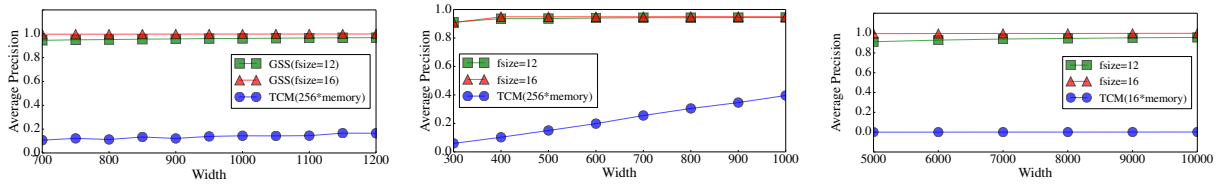
I. Experiment on Other Compound Queries

We compare GSS with state-of-the-art task specific algorithms on triangle counting and subgraph matching in this Section. We compare GSS with TRIEST [26] in triangle counting with the same memory. We use relative error between the reported results and the true value as evaluation metrics. TRIEST does not support multiple edges. Therefore we unique



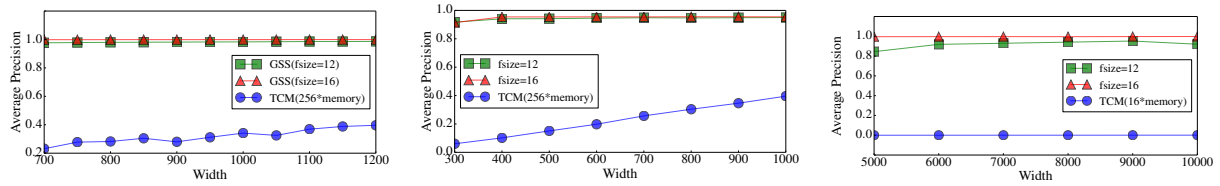
(a) web-NotreDame (b) lkml-reply (c) networkflow

Fig. 8. Average Relative Error of Edge Queries



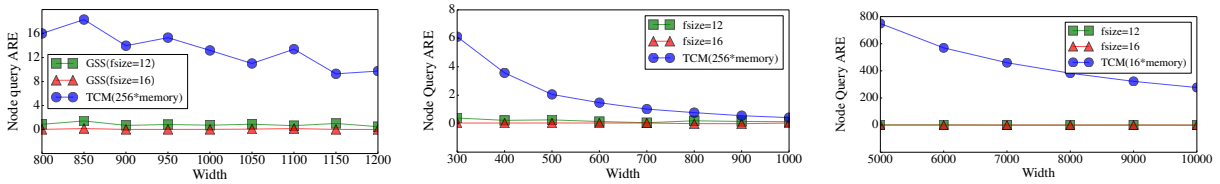
(a) web-NotreDame (b) lkml-reply (c) networkflow

Fig. 9. Average Precision of 1-hop Precursor Queries



(a) web-NotreDame (b) lkml-reply (c) networkflow

Fig. 10. Average Precision of 1-hop Successor Queries



(a) web-NotreDame (b) lkml-reply (c) networkflow

Fig. 11. Average Relative Error of Node Queries

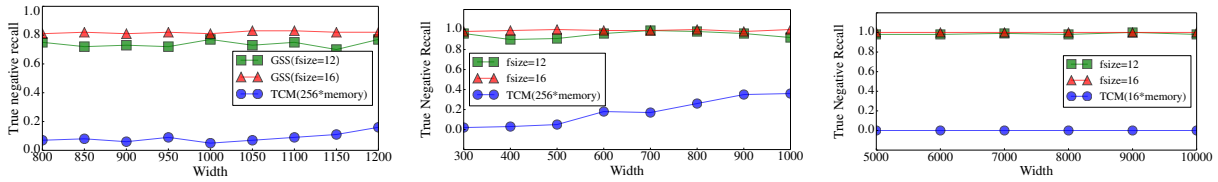
TABLE I
UPDATE SPEED (MIPS)

Data Structure	web-NotreDame	lkml-reply	networkflow
GSS	2.2887	2.61057	2.40976
GSS(no sampling)	2.1245	2.49191	2.32015
TCM	2.10417	2.5498	2.07403
Adjacency Lists	0.578596	0.3384	0.52147

the edges in the dataset for it. The results are shown in Figure 14. The results show that they achieve similarly high accuracy with relative error less than 1%. We compare GSS with SJ-tree [27] in subgraph matching. As SJ-tree is an accurate algorithm, we set GSS to $\frac{1}{10}$ of its memory. We use VF2 algorithm when querying in GSS, other algorithms can also be used. We use networkflow and search for subgraphs in windows of the data stream. The edges in the graph are labeled by the ports and the protocol. We carry out experiment on 5 window sizes, and for each window size, we randomly select 5 windows in the stream. In each window, we generate 4 kinds of subgraphs with 6, 9, 12 and 15 edges and 5 instances in each kind by random walk. We use the correct rate as evaluation metrics, which means the percentage of correct matches in the 100 matches for each window size. Experimental results are shown in Figure 15. We can see that GSS has nearly 100% correct rate. It should be noted that both TRIEST an SJ-tree have a much lower throughput (less than 2×10^5 edges per second) in order to achieve continuous query on specific problem. While GSS summarizes the graph with much higher speed (over 2×10^6 edges per second), supporting various kinds of queries, but needs additional processing in queries. The time complexity of queries depending on the algorithm we use. Technically, they are designed for different application scenarios. We carry out this experiment only to show the capability of GSS in supporting these compound queries.

VIII. CONCLUSION

Graph stream summarization is a problem rising in many fields. However, as far as we know, there are no prior work



(a) web-NotreDame (b) lkml-reply (c) networkflow

Fig. 12. True Negative Recall of Reachability Queries

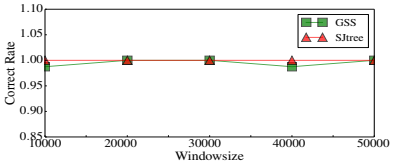


Fig. 15. Subgraph matching in networkflow

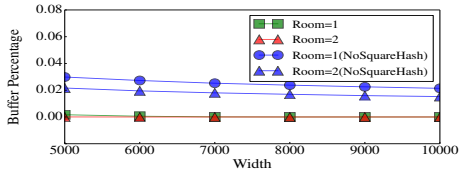


Fig. 13. Buffer percentage in networkflow

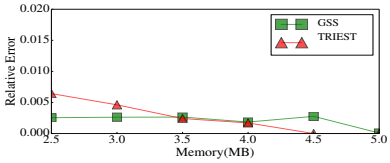


Fig. 14. Triangle count in lkml-reply

that can support most kinds of queries with high accuracy. In this paper, we propose graph stream summarization data structure Graph Stream Sketch (GSS). It has $O(|E|)$ memory usage where $|E|$ is the number of edges in the graph stream, and $O(1)$ update speed. It supports most queries based on graphs and has accuracy which is higher than state-of-the-art by magnitudes. Both mathematical analysis and experiment results confirm the superiority of our work.

ACKNOWLEDGEMENT

This work was supported by The National Key Research and Development Program of China under grant 2018YFB1003504 and NSFC under grant 61622201, 61532010 and 61672061. Xiangyang Gou finished this work under the guidance of his PhD supervisor: Lei Zou. Corresponding authors: Tong Yang (yangtongemail@gmail.com) and Lei Zou (zoulei@pku.edu.cn).

REFERENCES

[1] S. Guha and A. McGregor, "Graph synopses, sketches, and streams: A survey," *PVLDB*, vol. 5, no. 12, pp. 2030–2031, 2012.

[2] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," in *Latin American Symposium on Theoretical Informatics*, pp. 29–38, 2004.

[3] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM TOCS*, vol. 21, no. 3, pp. 270–313, 2003.

[4] P. Roy, A. Khan, and G. Alonso, "Augmented sketch: Faster and more accurate stream processing," in *SIGMOD*, pp. 1449–1463, ACM, 2016.

[5] D. Thomas, R. Bordawekar, C. C. Aggarwal, and S. Y. Philip, "On efficient query processing of stream counts on the cell processor," in *ICDE*, pp. 748–759, IEEE, 2009.

[6] P. Zhao, C. C. Aggarwal, and M. Wang, "gsketch: on query estimation in graph streams," *PVLDB*, vol. 5, no. 3, pp. 193–204, 2011.

[7] N. Tang, Q. Chen, and P. Mitra, "Graph stream summarization: From big bang to big crunch," in *SIGMOD*, pp. 1481–1496, 2016.

[8] A. Khan and C. Aggarwal, "Query-friendly compression of graph streams," in *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*, pp. 130–137, 2016.

[9] M. Elkin, "Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners," in *ICALP*, pp. 716–727, Springer, 2007.

[10] V. Braverman, R. Ostrovsky, and D. Vilenchik, "How hard is counting triangles in the streaming model?," in *ICALP*, Springer, 2013.

[11] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang, "On graph problems in a semi-streaming model," *Theoretical Computer Science*, vol. 348, no. 2-3, pp. 207–216, 2005.

[12] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "Graphx: Graph processing in a distributed dataflow framework.," in *OSDI*, vol. 14, pp. 599–613, 2014.

[13] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: distributed graph-parallel computation on natural graphs.," in *OSDI*, vol. 12, p. 2, 2012.

[14] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, pp. 135–146, ACM, 2010.

[15] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed graphlab: a framework for machine learning and data mining in the cloud," *PVLDB*, vol. 5, no. 8, 2012.

[16] S. Raghavan and H. Garcia-Molina, "Representing web graphs," in *ICDE*, pp. 405–416, IEEE, 2003.

[17] W. Fan, J. Li, X. Wang, and Y. Wu, "Query preserving graph compression," in *SIGMOD*, pp. 157–168, ACM, 2012.

[18] D. A. Spielman and N. Srivastava, "Graph sparsification by effective resistances," *SIAM Journal on Computing*, vol. 40, no. 6, 2011.

[19] J. Gao, C. Zhou, J. Zhou, and J. X. Yu, "Continuous pattern detection over billion-edge graph using distributed framework," in *IEEE ICDE*, 2014.

[20] C. Wang and L. Chen, "Continuous subgraph pattern search over graph streams," in *IEEE ICDE*, 2009.

[21] C. Song, T. Ge, C. Chen, and J. Wang, "Event pattern matching over graph streams," *PVLDB*, vol. 8, no. 4, 2014.

[22] P. LEcuyer, "Tables of linear congruential generators of different sizes and good lattice structure," *Mathematics of Computation*, vol. 68, no. 225, pp. 249–260, 1999.

[23] Code and technical report: https://github.com/Puppy95/Graph Stream-Sketch

[24] Z. Li, B. Chang, S. Wang, A. Liu, F. Zeng, and G. Luo, "Dynamic compressive wide-band spectrum sensing based on channel energy reconstruction in cognitive internet of things," *IEEE Transactions on Industrial Informatics*, 2018.

[25] F. Xiao, Z. Wang, N. Ye, R. Wang, and X.-Y. Li, "One more tag enables fine-grained rfid localization and tracking," *IEEE/ACM Transactions on Networking (TON)*, vol. 26, no. 1, pp. 161–174, 2018.

[26] L. D. Stefani, A. Epasto, M. Riondato, and E. Upfal, "Triest:counting local and global triangles in fully-dynamic streams with fixed memory size," in *ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 825–834, 2016.

[27] S. Choudhury, L. Holder, G. Chin, K. Agarwal, and J. Feo, "A selectivity based approach to continuous pattern detection in streaming graphs," *Computer Science*, vol. 93, no. 8, pp. 939–945, 2015.