Information Sciences 503 (2019) 469–492

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Labeled graph sketches: Keeping up with real-time graph streams

Chunyao Song[a,c,*], Tingjian Ge[b], Yao Ge[a], Haowen Zhang[a], Xiaojie Yuan[a]

[a] Tianjin Key Laboratory of Network and Data Science Technology, College of Computer Science, Nankai University, China
[b] University of Massachusetts Lowell, USA
[c] Jiangsu Key Laboratory of Big Data Security & Intelligent Processing, Nanjing University of Posts and Telecommunications, China

ARTICLE INFO

Article history:
Received 21 July 2018
Revised 4 July 2019
Accepted 5 July 2019
Available online 6 July 2019

Keywords:
Graph stream
Graph sketch
Labeled graph

ABSTRACT

Currently, graphs serve as fundamental data structures for many applications, such as road networks, social and communication networks, and web requests. In many applications, graph edges stream in and users are only interested in the recent data. In data exploration, the storage and processing of such massive amounts of graph stream data has become a significant problem. As the categorical attributes of vertices and edges are often referred to as labels, we propose a labeled graph sketch that stores real-time graph structural information using only sublinear space and that supports graph queries of diverse types. This sketch also works for sliding-window queries. We conduct extensive experiments on real-world datasets in six different domains and compare the results with a state-of-the-art method to show the accuracy, efficiency, and practicability of our proposed approach.

1. Introduction

Currently, graphs serve as fundamental data structures for many applications. Often, graph data is constantly produced at a fast rate, in the form of a graph stream—a stream of edges. Some examples include social networks such as Twitter and Weibo, real-time road traffic, telephone call networks, and web server requests. In data exploration, the storage and processing of such huge amounts of data has become a major problem.

Example 1. Fig. 1 shows a snapshot of an email network. Each vertex represents an email address, and each large oval encloses a group of email addresses belonging to the same domain server. An edge from A to B represents the event of user A sending an email to user B. The vertex identifier for a user is the email address, and each vertex also has a label, which is the domain server. Each edge may have a label as well, which is the email subject (e.g., "project" for edge AC). In such a situation, interesting queries include getting the discussion frequency over "project" between A and C during the last week or getting the top-k discussion subjects between A and B during the last month. We call these types of queries **aggregate edge frequency queries**. Likewise, **aggregate vertex flow queries** (e.g., the total number of emails of particular people on a certain subject within a given period) may be of interest for social media or business intelligence. In addition, **vertex label-based aggregate vertex/edge queries** are often needed in such applications. For example, querying the edge frequency between a specific email address and a domain server, the edge frequency between two domain servers, and the aggregate incoming/outgoing degrees of a domain server reveal useful information for various parties.

* Corresponding author at: College of Computer Science, Nankai University, 38 Tongyan Rd., Jinnan District, Tianjin 300000, China.
E-mail address: chunyao.song@nankai.edu.cn (C. Song).

https://doi.org/10.1016/j.ins.2019.07.019
0020-0255/

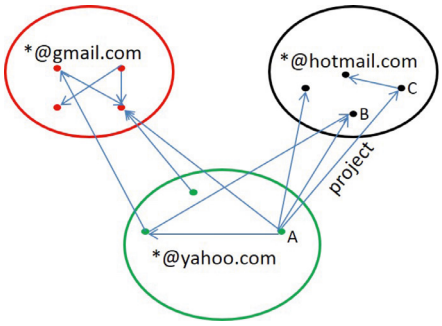

Fig. 1. Email Network.

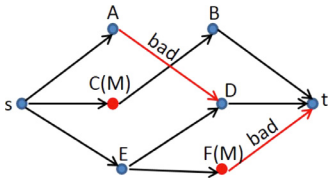

Fig. 2. Road Network.

Example 2. Fig. 2 shows a road traffic network. Each vertex is an intersection and has a label, indicating whether it is a major intersection, marked "M" in parentheses (unmarked vertices have label "secondary"). In heavy traffic, drivers may want to avoid M vertices to reduce delays. Each edge represents a road segment. The edge label "bad" indicates it is congested or under construction. A driver going from s to t asks whether there is a path that avoids all major intersections and bad roads. We call this a **path query**. Finally, given a subgraph query pattern (e.g., densely congested roads may reveal areas that are prone to congestion, thereby facilitating smart traffic management), **subgraph pattern match queries** are also often needed in practice.

In all these applications, data are continuously produced at a fast rate. It is reported that there are approximately 500 million tweets per day and over 300 billion tweets in total as of 2013 [1]. Storing all the necessary data in memory for real-time queries is infeasible. It is also well-known that *fast but approximate* answers are often better than *accurate but slow* results in data stream applications [34]. We propose a labeled graph sketch that stores graph stream information in sublinear space, while supporting many types of graph queries, such as those mentioned above. Our sketch structure also supports sliding-window queries, which are often needed in practice. More types of queries are shown in Sections 4 and 5.

1.1. Related work

Some previous studies on graph sketches maintain a random linear projection that aims to infer relevant properties of the input from the sketch and to retain the sketch in a small space. McGregor [32] is an excellent survey for this line of work. However, each is only applicable to a specific problem, such as finding a spanning forest of a graph, testing k-connectivity, min-cut, and sparsification. Thus, each of them needs to preserve few data features to solve a specific problem. Moreover, they do not deal with labeled graphs. Ahn et al. [11], Kapron et al. [25], Reittu et al. [36] all fall in this category.

Other sketch work also applies linear projections of data, while trying to preserve salient features of data using the CountMin sketch [15]. The CountMin sketch was originally used to summarize general data streams. gSketch [49] extends it to support graph streams, by first treating each edge as an element with a unique identifier and then applying CountMin. gMatrix [26] and TCM [43] have been proposed, which both use a three-dimensional sketch. Instead of treating the graph edge set as an element set and mapping each edge onto a one-dimensional space, gMatrix and TCM apply hash functions that define a mapping of the graph vertex set to an integer range $1 \ldots w$. However, they do not handle vertex/edge labels. Additionally, they do not handle automatic edge expiration for a sliding window. TCM [43] is the state-of-the-art method most related to our work. Thus, we compare our experimental results against TCM.

We have briefly described some preliminary high-level ideas of our sketches in an ICDE 2018 poster [41]. By contrast, in this paper, we present the complete work by including the detailed algorithms for query processing, theoretical analyses of each type of query, examples that are more illustrative, more query types, and comprehensive experimental evaluations.

Other graph compression/summarization work includes [18], where Fan et al. propose query preserving graph compression, especially for path reachability queries and bounded simulation pattern matching. However, this method needs to know the whole data graph in advance. Shah et al. propose TimeCrunch [39] to summarize important temporal structures for dynamic graphs and try to find patterns that agree with intuition. Their target is different from ours. Khandelwa et al. propose ZipG [27] to execute a larger fraction of queries in main memory. Some other work includes [12,31,40]. Previous work on

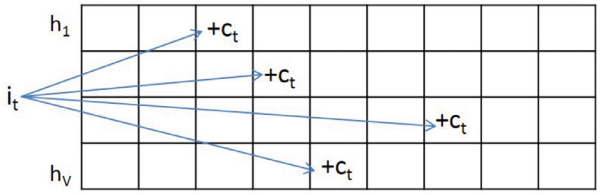

Fig. 3. Count-Min Sketch Insertions.

graph queries includes reachability queries such as [13,23,24,44,50], graph pattern matching queries [17,35,37,42,45,48], and top-k pattern queries [14,19,21,47]. As we show in Sections 3 and 4, our sketch can serve as a black box for most of the above algorithms.

Other graph stream sampling methods include [9,10,22,29,30]. GPS [10] provides an order-based reservoir sampling method for massive graph streams, which aims at estimating the frequency of occurrence of certain subgraphs, i.e., triangle and wedge counting, from graph streams. Jha et al. [22] also propose a space-efficient algorithm to estimate the global clustering coefficient (computed from the wedge count and triangle count) and the total triangle count, using the birthday paradox. Lim and Kang [29] and Lim et al. [30] only target the triangle counting problem. Ahmed et al. propose gSH [9] to build a generic graph stream sampling framework to estimate various graph properties, using the Horvitz–Thompson construction [20]. However, none of the above-mentioned methods can handle queries on labeled graphs.

1.2. Our contributions

First, we formulate the problem and propose a CountMin-based labeled graph sketch (LGS) that can store the original data graph in only sublinear space while maintaining the structural information and supporting many types of graph queries. A challenge of this problem, we need to store the label information without using significant extra storage space. To solve this problem, we utilize the characteristics of prime numbers to encode the label information into a product of prime numbers. The proposed LGS can handle query semantics that involve vertex or edge labels, edge expiration, and sliding windows, which enable it to perform better than traditional sketches for analytics over graph streams. We then devise algorithms to answer various types of queries, including aggregate vertex queries (on a specific vertex or a vertex type), aggregate edge queries (with or without a specified edge label), path reachability queries (with or without a specified edge type), and subgraph matching queries (with or without a specified edge label). We analyze the accuracy and complexity of our proposed algorithms. Importantly, we show that our sketch can work as a black box for many existing algorithms to support more graph queries. Comprehensive experiments have been conducted based on real-world datasets from six different domains. We compare our method with a state-of-the-art method, TCM [43], and find that our method shows better accuracy than TCM. Additionally, we verify our theoretical analysis and show that answering queries on our sketch takes significantly less time than on a raw data graph. Our contributions can be summarized as follows:

- We show the concept of labeled graph streams and provide the background of the CountMin sketch algorithm (Section 2).
- We introduce the framework of the proposed LGS (Section 3).
- We devise algorithms to answer various types of queries, and demonstrate that the proposed framework can work as a black box for many existing graph algorithms (Section 4).
- We perform comprehensive experiments to show the superiority of our proposed framework (Section 5).

2. Preliminaries

Graph Streams. A *graph stream* G is a sequence of elements $e = (A, B; t)$, where A is the identifier of e's starting vertex and is associated with a vertex label $L_v(A)$, and B is the identifier of e's ending vertex with a vertex label $L_v(B)$. Vertex labels may be used for different vertex types (e.g., email domain servers in Example 1). Each edge e also has an edge label $L_e(e)$ (e.g., different email subjects in Example 1). A timestamp t indicates the incoming time of edge e. Such a stream naturally defines a graph $G = (V, E)$, where V is a set of vertices $\{v_1, v_2, \ldots, v_n\}$ and E is a set of edges $\{e_1, e_2, \ldots, e_m\}$. Each vertex v_i has a vertex label $L_v(v_i)$ from a vertex label set Σ_N and each edge e_i has an edge label $L_e(e_i)$ from an edge label set Σ_E. A graph stream G is sorted by its edge incoming times, in nondecreasing order. We use the sliding-window model. Suppose the sliding window size is W time units (e.g., seconds) and the current time is t; we automatically discard edges with incoming time older than $t - W$. Our work applies to nonwindow settings as well.

CountMin Sketch (CMS). CMS was originally proposed by Cormode et al. [15], and is used for data stream summarization. It has a two-dimensional array of counts with width w and depth v: $count[1, 1] \ldots count[v, w]$. Suppose there are n distinct items in the data stream and v hash functions $[h_1, \ldots h_v : 1 \ldots n \rightarrow 1 \ldots w]$ are used, chosen uniformly at random from a pairwise-independent family to map the arriving items to matrix cells. When an item i arrives at time t with update c_t, the corresponding counts are updated in v rows, as shown in Fig. 3. The update c_t could be either positive or negative. c_t equals 1 for an unweighted arriving item. This model does not handle item expiration, because it does not store the arriving time

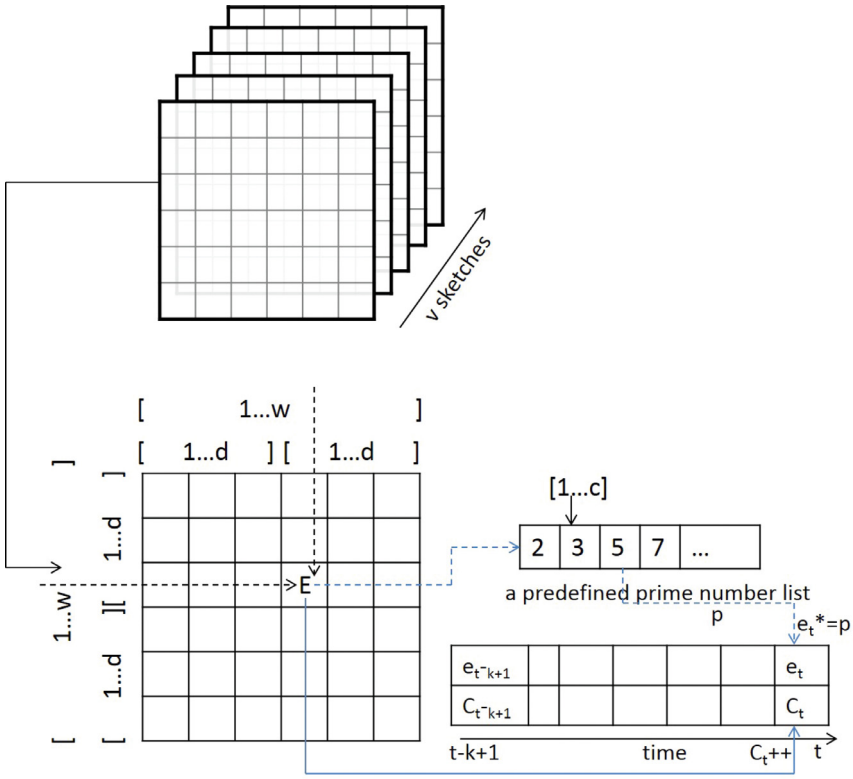

Fig. 4. Insertion Algorithm for LGS.

information. To remove an item from the sketch, we can simply set c_t to -1 for the unweighted case or some negative value for the weighted case. To retrieve the counts of an item, we again use v hash functions to map the query item to v cells and use the minimum value in those cells as the answer, i.e., $min\{count[1, h_1(i)], count[2, h_2(i)], \ldots, count[v, h_v(i)]\}$.

Pairwise Independent Hash Functions. A family of pairwise independent hash functions is used to reduce the collisions and, thus, improve the accuracy. It is defined as follows:

Definition 1. A family of functions $\mathscr{H} = \{h|h: [N] \rightarrow [M]\}$ is called a **family of pairwise independent hash functions** if for all $i \neq j \in N$ and $k, l \in M$,

$$Pr_{h \leftarrow \mathscr{H}}[h(i) = k \wedge h(j) = l] = 1/M^2$$

The notion of a pairwise independent hash family is also known as a "strongly universal hash family" [33,38].

In our implementation, we use the BKDR hash function [4] and generate functions with different seeds to achieve pairwise independence.

3. Labeled graph sketch

An edge e from vertex A to vertex B in a graph stream is identified as $(A, B, L_v(A), L_v(B), L_e(e))$. We consider three factors including the endpoint identifiers, endpoint labels, and edge labels for each incoming edge when we encode edges into the sketch. We use hash functions from a pairwise independent hash family to process these three factors. Because we deal with sliding windows, we need to handle item expiration. As we may not need a very fine time granularity, we divide the entire sliding window into k subwindows according to practical usage. We then maintain a counter for each subwindow. Fig. 4 illustrates how our algorithm works on an incoming edge, considering three factors. The upper part of Fig. 4 shows that v sketches are used to reduce the errors. As we deal with structured graph data, a single sketch is a two-dimensional matrix. Later, we will show how v affects the accuracy. The lower part of Fig. 4 shows how the insertion algorithm works for a single sketch.

Let a single sketch be a wd-by-wd matrix. That is, the whole matrix consists of w-by-w blocks, each of which is a d-by-d submatrix. Suppose we have an incoming edge $e(A, B)$ with an edge label $L_e(e)$. One endpoint A has a vertex label $L_v(A)$, and the other endpoint B has a vertex label $L_v(B)$. A hash function $\boldsymbol{h}$ is first used to map the labels $L_v(A)$ and $L_v(B)$ to the range $[1 \ldots w]$, giving one of the matrix blocks mentioned above. Then, $\boldsymbol{h}$ is used again to map the vertex *identifiers* A and B to the range $[1 \ldots d]$, reaching a single matrix cell E inside the chosen matrix block above. This two-level hashing (with

Algorithm 1: LabeledGraphSketch.

Input : A graph stream G, window size W, subwindow size W_s.
Output: LGS S

```
1  S ← an empty sketch
2  P ← a list of first c prime numbers
3  k ← W/W_s
4  for each new edge e = (A, B; t) in G do
5      m ← d * (h(L_v(A)) mod w)+h(A) mod d
6      n ← d * (h(L_v(B)) mod w)+h(B) mod d
7      E ← S[m][n]
8      if E[k].t + W_s equals t then
9          for i ← 2 to k do
10             E[i − 1] ← E[i]
11         E[k].t ← t
12     E[k].C ← E[k].C + 1
13     p_e ← h(L_e(e)) mod c
14     E[k].e ← E[k].e * P[p_e]
15     S[m][n] ← E
16 return S
```

vertex labels first, then vertex identifiers) of endpoints ensures that our sketch support aggregate vertex queries based on vertex labels (and types). Next, we process the edge label $L_e(e)$. Each matrix cell E contains a list of k subwindow counters. Each subwindow counter has two parts. The first part C_i is simply a count of edges hitting this hash table cell in the i-th subwindow. The second part is a product of prime numbers containing information from the edge labels, as follows. We have a list of unique prime numbers for our choice. For example, it may be the 172 prime numbers under 1024. We then use the hash function $\boldsymbol{h}$ to map the edge label $L_e(e)$ to the range $[1 \dots c]$, where c is the length of the prime number list. Next, we multiply the c-th prime number by the second part of the subwindow cell value e_i. The choice of c is application-dependent. Our experiments in Section 5 show that $c = 20$ already gives results with good accuracy (higher values will yield better accuracy but increase the computation cost, as larger prime numbers need to be multiplied). Compared to hash functions, the prime number list reduces the space consumption when compressing edge labels, because we do not need to allocate space to store the results initially. This is more obvious under the nonsliding-window scenario.

As we adopt the sliding-window model for graph streams, we need to maintain the sketches upon edge insertions and deletions. Edge insertion happens upon each edge arrival, and edge deletion happens upon each edge expiration.

Algorithm 1 is used to build the sketch online as each edge arrives, and we summarize the notations in Table 1 for reference.

Edge expirations are handled automatically with the help of the k subwindows. If an edge deletion is explicitly needed, we can simply adopt the opposite method of edge insertion. Any edge can be located within O(1) time using hash functions, and thus there is no need to maintain any indexes.

In short, for each arriving edge, we first locate the matrix cell that is associated with the insertion according to the two endpoints' identifiers and labels, and then we update the corresponding information of that cell according to the edge label. Edge expirations are handled automatically at the same time.

Let S be a wd-by-wd matrix. Each matrix cell E maintains two lists: one has an edge count for each subwindow, and each entry in the other list contains a product of prime numbers corresponding to the labels of edges in a subwindow. Lines 5–7 are to locate the matrix cell according to the two endpoints' identifiers and vertex labels of the incoming edge. Lines 8–11 are to check whether we need to start a new subwindow. If the starting time of the latest subwindow plus the size of the subwindow equals the current time t, then we need to start a new subwindow. Note that window sliding and item expirations are handled in this way. We do not need to keep the starting time of every subwindow; rather, we only need to keep the starting time of the last subwindow. Lines 12–14 are to update the counter and product fields of the subwindow. For sliding-window queries, the product of the prime numbers typically does not get so large as to exceed the word size (e.g., 64-bit). In case it does, especially for nonsliding-window queries, we break down the product into multiple words, as detailed in Section 4.5.

Furthermore, to distinguish collisions caused by hash functions, we use v groups of hash functions to get v sketches $\{S_1, \dots S_v\}$. Then, our query answer results are computed from these v sketches.

Example 3. Fig. 5 shows a running example for building the graph sketches under the stream scenario. Fig. 5(a) shows the original labeled graph stream, where A, B, D, E, and F are node identifiers; C (for core people) and P (for periphery people) are node labels; edges are labeled with L (for long calls) or S (for short calls); the numbers enclosed in parentheses next to

Table 1
Notations Used in this Paper.

Symbols	Meaning
W	Window size
W_s	Subwindow size
t	Current time
S	LGS
v	Number of sketch numbers
w	Vertex label hash value range size
d	Vertex identifier hash value range size
A, B	Vertex identifier
$L_v(A)$, $L_v(B)$	Vertex label
$L_e(e)$	Edge label
k	Number of subwindows
E	A cell in S storing edge information
$E[i].C$	The counter field of subwindow i in cell E
$E[i].e$	The prime number product field for subwindow i of cell E, encoding edge label information
$E[i].t$	The starting time of subwindow i

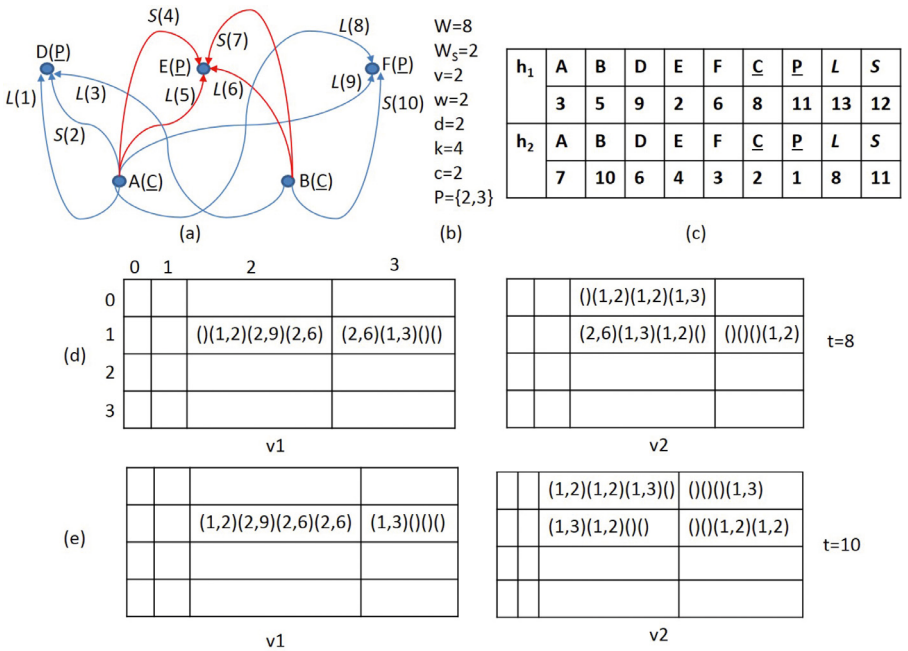

Fig. 5. Illustrative example of LGS.

an edge label indicate the arrival timestamp of that edge. Fig. 5(b) shows the parameters used in this example. The window size is 8; the subwindow size is 2; the number of sketches is 2; $w \cdot d$ equals 4, resulting in a 4*4 matrix; and the prime number list that we use is {2, 3}. Fig. 5(c) shows the hash values for each identifier/label of two pair-wise independent hash functions. Fig. 5(d) and (e) show the two sketches at times 8, and 10, respectively. The first number in parentheses stands for the counter field and the second number is the prime number product result. An empty pair of parentheses means that there is no edge coming into this subwindow. An empty matrix cell also means there is no corresponding edge. The column and row numbers of the matrix are also given in sketch 1 of Fig. 5(d). At time 9, the first subwindow expires, and thus the first (and only) empty pair of parentheses in [1, 2] and (2,6) in [1, 3] from v1 expire. Similarly, three subwindows from v2 expire. At time 9, $A(\underline{C})$ to $F(\underline{P})$ with edge label L arrives, and the information should be inserted in $[2 * (8\%2) + 3\%2,\ 2 * (11\%2) + 6\%2]$, which is [1, 2] for v1, and inserted in $[2 * (2\%2) + 7\%2,\ 2 * (1\%2) + 3\%2]$, which is [1, 3] for v2. To represent the edge label, (1,3) is added as a subwindow in [1, 2] for v1, and (1,2) is added as a subwindow in [1, 3] for v2. At time 10, $B(\underline{C})$ to $F(\underline{P})$ with edge label S arrives. The information should be inserted into [1, 2] for v1, and [0, 3] for v2. Therefore, the last subwindow in [1, 2] of v1 is updated to (1+1,3*2), which is (2,6), and (1,3) is added into [0, 3] for v2.

Algorithm 2: GetEdgeCount.

Input : An edge label l (optional), hash function $\boldsymbol{h}$ (optional), prime number list P (optional), specific matrix cell $S[m][n]$.

Output: i: The number of edges with label l in $S[m][n]$, j: the number of edges in $S[m][n]$ regardless of edge labels

```
1  p_e ← P[h(l) mod c] //get the prime number representation of label l if it is provided
2  i ← 0 //number of edges with label l
3  j ← 0 //number of edges regardless of labels
4  currentE ← S[m][n]
5  for s_t ← 1 to k do
6  |  j ← j + currentE[s_t].C
7  |  while currentE[s_t].e mod p_e = 0 do
8  |  |  i ← i + 1
9  |  |  currentE[s_t].e ← currentE[s_t].e/p_e
10 return i, j
```

4. Answering queries

Before we discuss in detail the queries that our sketch supports, let us first introduce an auxiliary process used to get the number of edges from a given vertex to another. All edges with the same two endpoints are hashed to the same matrix cell. We use GetEdgeCount (shown in Algorithm 2) to get the edge count in a matrix cell with or without a particular edge label. For a non-edge label-based query, we simply need to sum all the counters of the k subwindows, whereas for an edge label-based query, we sum the number of factors that represent the prime number of that edge label from the k subwindows.

The input to GetEdgeCount includes an optional edge label l to match, the corresponding optional hash function, and the prime number list. Lines 1, 2, and 7–9 compute the edge count in a matrix cell with an edge label if this optional input is provided. The function returns two counts: the number of edges in $S[m][n]$ that match the input label (if it is provided), and the number of all edges in $S[m][n]$. Lines 7–9 are due to the fact that each edge with label l contributes a factor p_e to the product. Line 4 clones the current queried matrix cell in the global sketch, and any subsequent query processing is performed on this cloned matrix cell; hence, the global sketch is not affected.

With the help of GetEdgeCount, LGS supports various types of vertex queries, edge queries, and path queries. When a query comes in, we apply $\boldsymbol{h}$ on the vertex labels (and vertex identifiers) to reach certain cell(s). We then apply GetEdgeCount to get the intended result for each cell. Next, we sum up the count of the final cells as the output for vertex queries and edge queries. For path queries, our sketch can be used as a black box for any existing path reachability algorithms. Let us now examine the details of how each type of query is answered.

4.1. Vertex queries

Aggregate vertex query: An aggregate vertex query is to estimate the overall in/out edge frequency of a vertex identifier/type $A/L_v(A)$, denoted as $\tilde{f}_v(A, \leftarrow)/\tilde{f}_v(L_v(A), \leftarrow)$ for in-flow queries and $\tilde{f}_v(A, \rightarrow)/\tilde{f}_v(L_v(A), \rightarrow)$ for out-flow queries. A vertex type $L_v(A)$ represents a set of all vertices with label $L_v(A)$. In particular, each vertex query could be associated with an optional edge label constraint, where all edges contributing to the resulting frequency should have a predefined edge label $L_e(e)$.

Our sketch supports all types of aggregate vertex queries, including edge label-based and non-edge label-based aggregate single-vertex queries (e.g., the in-degree of user B in Example 1), as well as edge label-based or non-edge label-based aggregate queries on a certain label of vertices (e.g., the total out-degree of all major intersections in Example 2, considering only *bad* roads). To get a specific vertex's incoming degree, we apply $\boldsymbol{h}$ twice to map this vertex to a particular column. We then sum up all the cell values in this column. The cell value is computed from GetEdgeCount, either with or without an edge label. Similarly, for a specific vertex's out-degree, we apply $\boldsymbol{h}$ twice to map this vertex to a specific row and get the sum values. Note that we use v groups of hash functions to distinguish collisions. The final value is the minimum we find among all these v sketches. Because we group vertices of the same type together, we also support aggregate vertex queries for different vertices with the same label. The detailed algorithm is shown in AggV. Algorithm AggV covers all four types of queries discussed earlier, and the input to the algorithm includes a vertex label $L_v(A)$, an optional vertex identifier A, and an optional edge label l. When querying a certain label of vertices, there is no need to provide a vertex identifier.

For a vertex query, we first locate the corresponding column(s) or row(s), and then sum all the necessary information to get the result. The error bound of vertex queries follows that of CMS, which is shown in detail in Theorems 1 and 3.

The AggV (Algorithm 3) algorithm handles all types of aggregate vertex queries on in-degrees. Lines 1–4 set four counters of each type of aggregate-vertex query to 0. Line 5 finds the column for an aggregate vertex query on a given vertex. Line 6 determines the starting column for an aggregate vertex query on a single vertex label/type, e.g., a large oval in Fig. 1.

Algorithm 3: AggV.

Input : A vertex identifier A, a vertex label $L_v(A)$, an edge label l

Output: i_c: The incoming number of edges with label l of vertex A, j_c: the incoming number of edges of vertex A, p_c: the number of incoming edges with label l of all vertices with label $L_v(A)$, q_c: the number of incoming edges of all vertices with label $L_v(A)$

```
1  i_c ←0
2  j_c ←0
3  p_c ←0
4  q_c ←0
5  m ← d * (h(L_v(A)) mod w)+h(A) mod d
6  n ← d * (h(L_v(A)) mod w)
7  for a ← 1 to d * w do
8      i_c ←i_c+i returned by GetEdgeCount (l, h, P, S[a][m])
9      j_c ←j_c+j returned by GetEdgeCount (S[a][m])
10 for b ← n to n + d do
11     for a ← 1 to d * w do
12         p_c ←p_c+i returned by GetEdgeCount (l, h, P, S[a][b])
13         q_c ←q_c+j returned by GetEdgeCount (S[a][b])
14 return i_c, j_c, p_c, q_c
```

Lines 7 and 8 together compute edge label-based aggregate vertex queries for a given vertex. A sample query of this type is "retrieve the number of emails on subject SALARY that user B has received" in Fig. 1. Line 9 deals with non-edge label-based aggregate vertex queries for a given vertex. A sample query of this type is "retrieve the number of emails that user B has received" in Fig. 1. Lines 10–12 process edge label-based aggregate vertex queries for a given vertex label/type. A sample query of this type is "retrieve the number of emails where the receiver's email server is Gmail and the email subject is HELLO" in Example 1. Lines 10, 11, and 13 manage non-edge label-based aggregate vertex queries on a certain vertex label/type. A sample query of this type is "retrieve the number of emails in which the receivers' email server is yahoo" in Example 1.

Having presented AggV for all aggregate vertex queries on in-degrees, we can obtain a similar algorithm for out-degrees. We only need to change the summation of a single column for a certain vertex or the summation of several columns for a given vertex type to the summation of a single row for a particular vertex or the summation of several rows for an individual vertex type.

Example 4. Take the graph stream shown in Fig. 5 as an example. At time 10, there is an in-flow query of $F(\underline{P})$. Then, for v1, we locate column 2 and get a result of 7. For v2, we locate column 3, and get a result of 3. The final result is min{7, 3} which is 3.

Theorem 1. *For non-edge label-based aggregate vertex queries on a given vertex, the estimated aggregate edge count $\hat{a}_i$ has the following guarantees: $a_i \le \hat{a}_i$, and with probability at least $1-\delta$,*

$$\hat{a}_i \le a_i + \epsilon E_w$$

where the number of sketches is $v = \lceil \ln \frac{1}{\delta} \rceil$, and the vertex identifier hash value range size d is $\lceil \frac{e}{\epsilon} \rceil$; a_i is the ground truth answer and E_w is the total number of edges within the sliding window W.

Proof. Our proof extends the one for CMS [15]. We introduce indicator variables $I_{i,j,k}$, which are 1 if $(i \neq k) \wedge (h_j(i) = h_j(k))$ (for vertex queries, two vertices are hashed to the same row/column) for any hash functions of the v sketches, and 0 otherwise. There are two cases for $I_{i,j,k} = 1$.

Case 1: Different vertices with *different* vertex labels are hashed to the same column/row, which means the hash collision has happened twice. By the pairwise independence of the hash functions, the probability is

$$P_1(I_{i,j,k}) = 1/(w * d)$$

Case 2: Different vertices with the *same* vertex labels are hashed to the same column/row. By the pairwise independence of hash functions, the probability is

$$P_2(I_{i,j,k}) = 1/d$$

Because both cases 1 and 2 are conditional probabilities for two subcases,

$$E[I_{i,j,k}] \le max(P_1(I_{i,j,k}), P_2(I_{i,j,k})) \le \frac{1}{d} \le \frac{\epsilon}{e}$$

Define a random variable $X_{i,j} = \sum_{k=1}^{n} I_{i,j,k} a_k$, where n is the total number of distinct vertices, and a_k represents the number of edges incident to a certain vertex k. Observe that

$$E[X_{i,j}] = E\left[\sum_{k=1}^{n} I_{i,j,k} a_k\right] \leq \sum_{k=1}^{n} a_k E[I_{i,j,k}] \leq \frac{\epsilon}{e} E_w$$

by the pairwise independence of h_j, and the linearity of expectation. By the Markov inequality,

$$\begin{aligned} Pr[\hat{a}_i < a_i + \epsilon E_w] &= Pr[\forall j \cdot count[j, h_j(i)] < a_i + \epsilon E_w] = Pr[\forall j \cdot a_i + X_{i,j} < a_i + \epsilon E_w] \\ &= Pr[\forall j \cdot X_{i,j} < eE(X_{i,j})] > 1 - e^{-d} \geq 1 - \delta. \end{aligned}$$

□

We can quickly verify that when v is greater than 5, the algorithm has a probability greater than 0.99 to fall in the theoretical error range. We also validate this in our experiments. In practical use, we can run a small window to obtain the statistics about the incoming graph stream and set the corresponding parameters according to the space requirement.

Theorem 2. *The time complexity of answering non-edge-label aggregate vertex queries on a given vertex is $O(w \cdot d \cdot k)$, which is $O(\lceil \frac{e}{\epsilon} \rceil \cdot k)$, where k is the number of subwindows.*

Proof. In this case, we only need to sum up all the numbers in one column/row. There are $w \cdot d$ cells in a column/row. Then, for each cell, we need to add up all k counters of this cell's counter list. □

Theorem 3. *For edge label-based aggregate vertex queries on a certain vertex, the estimated aggregate edge count $\hat{a}_i$ has the following guarantees: $a_i \leq \hat{a}_i$, and with probability at least $1 - \delta$,*

$$\hat{a}_i \leq a_i + \epsilon E_w$$

where the number of sketches is $v = \lceil \ln \frac{1}{\delta} \rceil$; the vertex identifier hash value range is d; the number of prime numbers used is c, and $d \cdot c$ is set to $\lceil \frac{e}{\epsilon} \rceil$; a_i is the ground truth answer, and E_w is the number of edges within the sliding window W.

Proof. The proof is similar to the proof of Theorem 1, and hence we omit it here. □

Theorem 4. *The time complexity of answering an edge label-based aggregate vertex query on a given vertex is $O(w \cdot d \cdot k + i)$, which is $O(\lceil \frac{e}{\epsilon} \rceil \cdot k + i)$, where i is the query result.*

Proof. In this case, we again need to sum up all the numbers in one column/row. Moreover, for each matrix cell, we need to check whether each subwindow includes the specific edge label for all k subwindows. If so, we check how many edges it includes, and this process takes at most i time units. Otherwise, we check the next subwindow. Thus, compared to non-edge label-based aggregate vertex queries, edge label-based aggregate vertex queries consume at most i more time units. □

As we have discussed earlier, our sketch also supports aggregate vertex queries on a specific vertex type/label. The error bounds for such queries are similar to those of aggregate vertex queries on a given vertex. The time complexity for non-edge label-based aggregate vertex queries on a certain vertex label/type and the time complexity for edge label-based aggregate vertex-queries on a given vertex label/type are $O(w^2 d)$ and $O(w^2 d + i)$, respectively, where i is the query result.

4.2. Edge queries

Aggregate edge query: An aggregate edge query is to estimate the edge frequency from a node identifier/type $A/L_v(A)$ to a node identifier/type $B/L_v(B)$, which is denoted as $\tilde{f}_e(\mathscr{A}, \mathscr{B})$, where $\mathscr{A}$ represents either a node identifier A, or a node type $L_v(A)$, and similar for $\mathscr{B}$. In particular, each aggregate edge query could retain an optional edge label constraint, where all edges contributing to the final frequency should carry a predefined edge label $L_e(e)$.

Our sketch supports various types of aggregate edge queries, including the following six:

- Aggregate edge queries between two vertices, without a required edge label.
- Aggregate edge queries between two vertices, with a required edge label.
- Aggregate edge queries between a vertex and vertices of a certain label, without a required edge label.
- Aggregate edge queries between a vertex and vertices of a certain label, with a required edge label.
- Aggregate edge queries between vertices of two types, without a required edge label.
- Aggregate edge queries between vertices of two types, with a required edge label.

For all the query types above, we only need to locate the corresponding matrix cells from the sketch and retrieve the required information. The error bound of edge queries follows that of CMS as well, as shown in detail in Theorem 5. The detailed algorithm is shown in AggEdge (Algorithm 4). The final result is the minimum value among all v sketches.

Lines 1–3 compute aggregate edge queries between two vertices with a particular edge label. A sample query of this type is "retrieve the number of emails that A sends to B, where the email subject is work" in Example 1. Lines 1, 2, and 4 process aggregate edge queries between two vertices without a specific edge label. A sample query of this type is "retrieve

Algorithm 4: AggEdge.

Input : An edge $e = (A, B)$ with vertex labels $L_v(A)$ and $L_v(B)$, and an edge label $L_e(e)$

Output: i_{vv}: Aggregate edge count with label $L_e(e)$ between vertices A and B,j_{vv}: Aggregate edge count between vertices A and B,i_{vt}: Aggregate edge count with label $L_e(e)$ between vertex A and vertex type $L_v(B)$,j_{vt}: Aggregate edge count between vertex A and vertex type $L_v(B)$,i_{tt}: Aggregate edge count with label $L_e(e)$ between vertex type $L_v(A)$ and vertex type $L_v(B)$,j_{tt}: Aggregate edge count between vertex type $L_v(A)$ and vertex type $L_v(B)$

1 $m \leftarrow d * (\boldsymbol{h}(L_v(A)) \bmod w)$+$\boldsymbol{h}(A) \bmod d$
2 $n \leftarrow d * (\boldsymbol{h}(L_v(B)) \bmod w)$+$\boldsymbol{h}(B) \bmod d$
3 $i_{vv} \leftarrow i$ returned by GetEdgeCount($L_e(e)$, $\boldsymbol{h}$, P, $S[m][n]$)
4 $j_{vv} \leftarrow j$ returned by GetEdgeCount($S[m][n]$)
5 $i_{vt} \leftarrow 0$
6 $j_{vt} \leftarrow 0$
7 $i_{tt} \leftarrow 0$
8 $j_{tt} \leftarrow 0$
9 **for** $a \leftarrow d * (\boldsymbol{h}(L_v(B)) \bmod w)$ *to* $d * (\boldsymbol{h}(L_v(B)) \bmod w)$+$d$ **do**
10 $i_{vt} \leftarrow i_{vt} + i$ returned by GetEdgeCount($L_e(e)$, $\boldsymbol{h}$, P, $S[m][a]$)
11 $j_{vt} \leftarrow j_{vt} + j$ returned by GetEdgeCount($S[m][a]$)
12 **for** $a \leftarrow d * (\boldsymbol{h}(L_v(A)) \bmod w)$ *to* $d * (\boldsymbol{h}(L_v(A)) \bmod w)$+$d$ **do**
13 **for** $b \leftarrow d * (\boldsymbol{h}(L_v(B)) \bmod w)$ *to* $d * (\boldsymbol{h}(L_v(B)) \bmod w)$+$d$ **do**
14 $i_{tt} \leftarrow i_{tt} + i$ returned by GetEdgeCount($L_e(e)$, $\boldsymbol{h}$, P, $S[a][b]$)
15 $j_{tt} \leftarrow j_{tt} + j$ returned by GetEdgeCount($S[a][b]$)
16 return i_{vv}, j_{vv}, i_{vt}, j_{vt}, i_{tt}, j_{tt}

the number of emails that A sends to B" in Example 1. Lines 5–8 set the initial counter values for the remaining four types of queries. Lines 1, 2, 9, and 10 deal with aggregate edge queries between a vertex and a vertex label/type with a required edge label. A sample query of this type is "retrieve the number of emails that A sends to all receivers whose email server is Gmail, where the email subject is GREETINGS" in Example 1. Lines 1, 2, 9, and 11 manage aggregate edge queries between a vertex and a vertex label/type. A sample query of this type is "retrieve the number of emails that A sends to all receivers whose email server is Hotmail" in Example 1. Lines 1, 2, and 12–14 together resolve aggregate edge queries between two vertex labels/types with a required edge label. A sample query of this type is "retrieve the number of emails where senders use a Gmail server, and receivers use a Hotmail server, and the email subject is SALARY" in Example 1. Lines 1, 2, 12, 13, and 15 solve aggregate edge queries between two vertex labels/types. A sample query of this type is "retrieve the number of emails where senders use a Yahoo server, and receivers use a Yahoo server as well" in Example 1.

Example 5. Consider the graph stream shown in Fig. 5. At time 10, an edge query $f_e(A(\underline{C}), E(\underline{P}))$ with edge label L arrives. In v1, {(1, 2)(2, 9)(2, 6)(2, 6)} is retrieved from [1, 2] and L corresponds to 3, and thus we get $\tilde{f}_e$=3 from v1. In v2, {(1, 3)(1, 2)()()} is retrieved from [1, 2] and L corresponds to 2, and thus we get $\tilde{f}_e$=1. The final result is min{3, 1}=1.

Theorem 5. *For aggregate edge queries between two vertices without an edge label constraint, the estimated aggreagate edge count $\hat{a}_i$ has the following guarantees: $a_i \le \hat{a}_i$, and with probability at least $1 - \delta$,*

$$\hat{a}_i \le a_i + \epsilon E_w$$

where the number of sketches is $v = \lceil \ln \frac{1}{\delta} \rceil$; the vertex identifier hash value range size d^*d is $\lceil \frac{e}{\epsilon} \rceil$; a_i is the ground truth answer, and E_w is the number of edges within the sliding window W.

Proof. The proof is similar to the proof of Theorem 1, and hence we omit it here. □

It is easy to see that the time complexity of aggregate edge queries between two vertices without an edge label constraint is $O(k)$, for summing up the k subwindows in a matrix cell. Similarly, aggregate edge queries between two vertices with an edge label constraint achieve the same error bound as the non-edge-label case when setting d^*d^*c to $\lceil \ln \frac{1}{\epsilon} \rceil$. The time complexity of aggregate edge queries between two vertices with an edge label constraint is $O(k + i)$, where i is the query result.

The other four types of queries all achieve the similar error bounds. The time complexity for aggregate edge queries between a vertex and a vertex label/type without an edge label constraint is $O(d \cdot k)$. The time complexity for aggregate edge queries between a vertex and a vertex type with an edge label constraint is $O(d \cdot k + i)$, where i is the query result. The time complexity for aggregate edge queries between two vertex labels/types without edge label constraint is $O(d^2k)$, and the time complexity for aggregate edge queries between two vertex labels/types with an edge label constraint is $O(d^2k + i)$, where i is the query result.

Algorithm 5: PathReachability.

Input : Two vertices A, B
Output: whether there is a path from A to B

```
1 s ← d * (h(L_v(A)) mod w)+h(A) mod d
2 t ← d * (h(L_v(B)) mod w)+h(B) mod d
3 j_e ← j returned by GetEdgeCount(S[s][t])
4 if j_e > 0 then
5 |  return true
6 V ← ∅ //visited list, initially empty
7 return Reach(s, t)
```

4.3. Path queries

Path reachability query: given two node identifiers A and B, a path reachability query is a Boolean query, denoted as $\tilde{r}(A, B)$, indicating whether there is a path from A to B. In particular, edges on the path could be constrained with a predefined edge label $L_e(e)$.

Because our sketch maintains all structural information, it can be used as a black box for any existing path reachability algorithms. Here, we use the classical depth-first search (DFS) for an illustration. Again, employ GetEdgeCount, where a greater than 0 result indicates an edge exists, and 0 otherwise.

To check whether there is a path from vertex A to vertex B, we first perform an initialization in PathReachability (Algorithm 5). If it happens that (A, B) is already an edge in the graph, then B is reachable from A. Otherwise, we call a recursive function Reach and perform a DFS to check whether B is reachable from A. Note that there is no false negative and only false positive, i.e., if B is not reachable from A is returned by the algorithm, then B is indeed not reachable from A. On the contrary, if B is reachable from A is returned by the algorithm, it is possible that B is not in fact reachable from A.

Lines 1–2 in PathReachability map the starting and ending vertices to the corresponding row and column. Lines 3–5 check whether these two vertices are connected directly. Again, we use the auxiliary algorithm GetEdgeCount. The idea is simple; if the aggregate edge count between these two vertices is greater than 0, then they are certainly connected. If the two queried vertices are not connected initially, then we call Reach (Algorithm 6) to perform a depth-first recursive search.

We do not specify an edge label in our algorithm. However, it is possible to check whether vertex B is reachable from vertex A through edges of a certain type. In that case, we only need to change lines 3 and 4 of PathReachability and lines 2, 3, 6, and 7 of Reach to $i_e \leftarrow i$ **returned by GetEdgeCount**(l, $\boldsymbol{h}$, P, $S[s][t]$) and **if** $i_e > 0$. An example query of this type is "check whether t is reachable from s through all good roads" in Fig. 2. It is also easy to extend our algorithm to answer a query such as "is vertex B reachable from vertex A with at most two edges of type $L_e(e)$?" by maintaining a state variable

Algorithm 6: Reach.

Input : s: starting vertex row ID in matrix,t: ending vertex column ID in matrix
Output: whether there is a path from s to t

```
1 V ← V ∪ s
2 j_e ← j returned by GetEdgeCount(S[s][t])
3 if j_e > 0 then
4 |  return true
5 for i ← 1 to d * w do
6 |  j_e ← j returned by GetEdgeCount(S[s][i])
7 |  if j_e > 0 and i ∉ V then
8 |  |  if Reach(i, t) then
9 |  |  |  return true
10 return false
```

of how many edges of type $L_e(e)$ have been found so far in the algorithm. The proposed sketch is also able to handle path reachability queries that avoid certain node and/or edge types. To avoid a certain edge type, we only need to add $i_e \leftarrow i$ **returned by GetEdgeCount**(l, $\boldsymbol{h}$, P, $S[s][t]$) before lines 3 and 7, and then change lines 3 and 7 to **if** $j_e > i_e$.

We run the algorithm for all v sketches. A vertex is reachable from another vertex only if there is a path between these two vertices returned by all v sketches. As discussed earlier, when a vertex B is reachable from vertex A in the graph stream window, vertex B is always reachable from A in our sketch. The reason for this is that if there were indeed a path from vertex A to vertex B, because of the existence of each connecting edge, the corresponding hash cell value would be greater

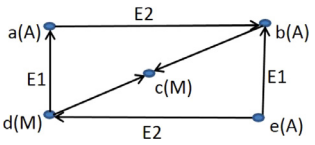

Fig. 6. Example of a subgraph.

than or equal to 1, such that the sketch could also find it as a path. However, if vertex B is not reachable from vertex A in the graph stream window, it is possible that the sketch might find a false path from A to B.

Example 6. Suppose at time 10, a path reachability query $r(A(\underline{C}), B(\underline{C}))$ comes for the graph stream shown in Fig. 5. In v1, the start point $A(\underline{C})$ corresponds to row 1, where column 2 and column 3 are not empty. The endpoint $B(\underline{C})$ corresponds to column 1. If we can reach a non-empty column 1 from row 1, then we find a path. However, in row 1, the first non-empty column we find is column 2. Then we develop row 2, which is all empty. We then go back to row 1, and find a non-empty column 3. Then we develop row 3, and find it to be all empty. We then go back to row 1, and all the non-empty columns have been developed. Therefore, we return NO REACH. In this case, there is no need to run the algorithm on v2, because there are no false negatives for path reachability queries. The query result is "there is no path from $A(\underline{C})$ to $B(\underline{C})$."

Theorem 6. *Suppose our algorithm finds a path (i.e., sequence) P of l positive edges in the sketch (line 6 in* REACH*). In a hypothesis-testing setup, let the null hypothesis be that no path in the graph stream corresponds to P. Then the false positive rate of our algorithm (i.e., the probability of finding P, and thus mistakenly rejecting the null hypothesis) is no more than* $1 - e^{-\frac{E_w l}{d^2 w^2}}$, *where* E_w *is the number of edges within the current window and d and w are vertex identifier hash value range size and vertex label hash value range size, respectively.*

Proof. Let us first get the false positive rate of a single cell in the sketch, i.e., it is set to positive by chance, and thus it is an upper bound of the false positive rate of a single edge in the sketch. The probability of a graph edge hitting a cell is $\frac{1}{d^2 w^2}$. Then, the probability that none of the E_w edges in the window hits that cell is $(1 - \frac{1}{d^2 w^2})^{E_w} \approx e^{-\frac{E_w}{d^2 w^2}}$, giving the upper bound of a single cell's false positive rate of

$$p_+ \leq 1 - e^{-\frac{E_w}{d^2 w^2}} \tag{1}$$

Next, we calculate the overall false positive rate. First, the probability that our algorithm does NOT make the false positive error is as follows:

$$\begin{aligned} Pr(correct) &\geq \Pi_{i=1}^{l}[p_i(1-p_+) + (1-p_i)] \\ &\geq \Pi_{i=1}^{l}[p_i(1-p_+) + (1-p_i)(1-p_+)] \\ &= \Pi_{i=1}^{l}(1-p_+) \\ &= (1-p_+)^l \end{aligned}$$

where p_i is the probability that the i-th edge/cell in P does not really correspond to an edge in the graph stream window. Thus, the overall false positive rate is

$$Pr(false\ positive) = 1 - Pr(correct) \leq 1 - (1-p_+)^l$$

Using the upper bound of p_+ in Eq. (1), we then get the upper bound of the false positive rate as stated in the theorem. □

Theorem 6 gives the result for one sketch. When there are v sketches, they may return different paths (of different lengths). Given that no path exists from A to B in the graph stream window, the null hypothesis is true for each of the v sketches. The overall error probability is the product of those from each sketch, because the v sketches are independent.

4.4. Subgraph queries

Subgraph Query: A subgraph query is to estimate the number of subgraph pattern matching, given a subgraph $G_S(Q)$, which is denoted as $\tilde{f}_{G_S}(Q)$.

A subgraph is represented as a group of constituent edges $G_S(Q) = (e_1, e_2, \ldots, e_n)$. Each e_i is represented as $< x_i, L_v(x_i), y_i, L_v(y_i), L_e(e_i) >$, where x_i and y_i represent the two endpoints of edge e_i; $L_v(x_i)$ and $L_v(y_i)$ are the node labels of the two endpoints, respectively; and $L_e(e_i)$ is an optional edge label. In particular, if for each e_i $(2 \leq i \leq n)$, y_{i-1} equals x_i, then the subgraph query becomes a **path search** query. Consider Fig. 6 as an example, where the distinct identifiers of each node are represented as lowercase letters, and the uppercase letters shown in parentheses denote node labels. The optional edge labels are shown on the edges. The subgraph shown in Fig. 6 can be represented as ($< a, A, b, A, E2 >$, $< d, M, a, A, E1 >$, $< d, M, c, M, * >$, $< b, A, c, M, * >$, $< d, M, e, A, E2 >$, $< e, A, b, A, E1 >$). A path search from node d to node b can be

Algorithm 7: Subgraph.

Input : A set of edges $(e_1, e_2, \ldots, e_n)$ constituting the subgraph, where each e_i is in the form of $< x_i, L_v(x_i), y_i, L_v(y_i), L_e(e_i) >$

Output: The number of matches of the subgraph or NO MATCH

1 $Sub_n \leftarrow$ MAX
2 **for** $i \leftarrow 1$ *to* n **do**
3 $m \leftarrow d * (\boldsymbol{h}(L_v(x_i)) \bmod w) + \boldsymbol{h}(x_i) \bmod d$
4 $n \leftarrow d * (\boldsymbol{h}(L_v(y_i)) \bmod w) + \boldsymbol{h}(y_i) \bmod d$
5 $e_c \leftarrow i$ returned by GetEdgeCount($L_e(e_i)$, $\boldsymbol{h}$, P, $S[m][n]$) **OR** j returned by GetEdgeCount($S[m][n]$) if $L_e(e_i)$ is empty
6 **if** e_c *is 0* **then**
7 **return NO MATCH**
8 **if** $e_c < Sub_n$ **then**
9 $Sub_n \leftarrow e_c$
10 **return** Sub_n

represented as ($< d, M, a, A, E1 >$, $< a, A, b, A, E2 >$). A subgraph query $\tilde{f}_{G_S}(Q)$ returns the number of matches of $G_S(Q)$ in the underlying graph stream, and there could be no match if the number is 0.

Essentially, a subgraph query is a set of aggregate edge queries. We execute an aggregate edge query for each of the constituent edges of the subgraph, and update the number of matches if we see fewer matching edges. The algorithm automatically terminates once we see a 0 match. The final result is determined by the minimum value of all v sketches.

Line 1 in Subgraph (Algorithm 7) initializes the matching result to a predefined MAX value. The for loop processes the edges of the subgraph, one after another. Lines 3–5 locate the current searching edge and get the matching number of the current edge with an optional edge label. Lines 6–7 early-terminate the algorithm if there is no match of the current edge. Lines 8–9 update the number of subgraph matches according to the edge matches seen most recently.

Example 7. Suppose at time 10, a subgraph query is shown in red edges in Fig. 5. For clarity, we use the timestamp to represent each edge. In v1, $\tilde{f}_e(4)$=3, $\tilde{f}_e(5)$=4, $\tilde{f}_e(6)$=4, and $\tilde{f}_e(7)$=3, and thus we get $\tilde{f}_{G_S}(Q)$=3. In v2, we get $\tilde{f}_e(4)$=1, $\tilde{f}_e(5)$=1, $\tilde{f}_e(6)$=2, and $\tilde{f}_e(7)$=1, and thus we get $\tilde{f}_{G_S}(Q)$=1. The final result is min{3, 1}=1.

The accuracy of **Subgraph** follows the error bound of aggregate edge queries, because subgraph queries are essentially combinations of aggregate edge queries. The time complexity of a subgraph query is linear with respect to the size of the querying subgraph.

4.5. Discussions

Our sketch encodes a broad range of information from graph streams, including not only edge and vertex connection information, but also edge and vertex label information. Thus, our sketch can be used as a subroutine or black box for many existing graph algorithms. For example, this sketch may be applied to subgraph pattern matching, where it is used as a prefilter. Only if all edges in the query pattern have their mapping edges in the sketch, do we go to the graph stream to verify and obtain an exact result.

In our scheme, subwindows are used to handle the sliding-window scenario. It is also possible to handle the nonsliding-window situation without the use of subwindows. In that case, an update includes both insertion and deletion, and deletion is just the reverse operation of insertion. Without subwindows, each matrix cell only maintains a single counter for the non-edge-label case and a prime number product list for the edge-label case. The prime number product list is only in use when the product is too large, and we need to break it into pieces. Otherwise, we can use a single number to hold the product. Overall, for the nonsliding-window case, and for the same raw data size, we could use less storage space to achieve the same accuracy bound compared to the sliding-window case. Additionally, when answering queries, the time complexity is also less because we do not need to multiply the subwindow number k. This is because we no longer need to sum up all the counters in the counter list for both the non-edge-label case and the edge-label case. The numbers we need to compute are also obviously fewer than in the sliding window case.

The theoretical error bounds that we have shown in the theorems are conservative guarantees and work for any skewed data as well. For skewed data, we can collect the statistics for a short time at the beginning and allocate the storage matrix cells accordingly.

If queries on a required vertex type are frequent, we could use an extra matrix cell for each submatrix to hold the information of the aggregate edge queries between two vertex types, and thus improve the query efficiency. Overall, our sketch framework is flexible and could fit in many applications.

5. Experiments

5.1. Datasets and setup

We use six datasets in our experiments as follows:

Phone data: the MIT Reality Mining dataset [16]. The Reality Mining project was conducted from 2004 to 2005 at the MIT Media Lab. The study followed 94 subjects using mobile phones pre-installed with several pieces of software that recorded and sent the researchers data about call logs, Bluetooth devices in a proximity of approximately five meters, cell tower IDs, application usage, and phone status. Each item of the graph stream (i.e., edge) contains the caller ID, receiver ID, time of the call, call type, and duration of the call. The vertices are naturally divided into two groups: *core people* (94 subjects) involved in this research and *periphery people* (other people who contact them) as vertex labels. An edge label is determined by the call type and the duration of the call, such that the dataset contains nine edge labels. Edges are sorted according to their timestamps.

Road data: the real-time traffic speed map of the roads in Hong Kong [5]. This graph stream is in XML format, where each observation of a road segment (a piece of XML structure) corresponds to an incoming graph edge. This contains the observation time, vertex IDs of the two ends of the road, region, road type, road saturation level, and traffic speed. The edges are sorted according to their timestamps. We do not consider vertex labels in this scenario, because vertex labels are only maintained through an extra level of hashing. We are more interested in edge labels. We consider the road type and road saturation level as edge labels, and obtain six edge labels in this case.

DBLP data: We download the DBLP XML data from [2]; a description of the data can be found in [28]. The DBLP data in [2] is updated quite often, and we use the Nov. 17th, 2016 version. We extract 5,507,607 author–article pairs from the original XML data with known authors, article titles, and publication years. The vertex types include *author* and *article*. The publication year represents the edge label. This dataset contains 140 edge labels in total, including some noisy entries that we intentionally keep. To make the graph more complex, we also obtain DBLP coauthorship pairs from [3], where "author" is the only vertex label and different years are used as edge labels.

Twitter data: We use the Twitter Stream API [6] implemented by twitter4j [7] to retrieve real-time Twitter streams, using the twenty most common words as keywords (which results in a much higher rate than the random sample provided in [7]). The vertices have two labels, *user* and *message*. An edge from a user to a message is created when the user sends a message; edges from a user/message to a reply (retweet,resp.) user/message are also created when a reply (retweet,resp.) is performed. These edges have 20 different labels based on topics. We download the real-time stream for two weeks, resulting in a total size of approximately 36 GB.

Enron email data: We obtain the dataset from [8]. The dataset contains approximately 500,000 internal emails from approximately 150 employees at Enron. We retrieve sender email addresses and receiver email addresses as two endpoint identifiers. We also retrieve the corresponding sender status and receiver status as vertex labels. The status is the last position of the employee. We label the status of people who work outside of Enron as *Others*. We use email subjects as edge labels. The edges are ordered by date. The entire dataset contains 75,406 distinct vertices with 11 vertex labels, and 2,064,442 edges with 140,660 edge labels, from 17,527 distinct senders to 68,074 distinct receivers. The average in-degree is 30.33, and the average out-degree is 117.79. The largest out-degree is 65,675, and the largest in-degree is 19,198.

Friendster social network data: We adopt the largest dataset from SNAP [46], which contains 1,806,067,135 edges. We randomly assign a vertex label for each distinct vertex from a label set of size 100, and randomly assign an edge label from an edge label set of size 20 to make this dataset suit our needs, resulting in a size 74.4 GB on disk. We use this dataset to test the scalability of our methods.

Although TCM [43] is not able to handle queries on a labeled graph, it is the most recent approach that is most related to ours. Therefore, we use TCM [43] as a competing method. We run TCM[43] on non-label-based queries fro comparison and evaluate our method on label-based queries.

We implement all our algorithms and the competing method in Java. All the experiments are performed on a machine with an Intel Core i7 3.40 GHz processor and 16 GB of memory.

We try to answer the following questions:

- How are the time and storage efficiency of our LabeledGraphSketch?
- How accurate it is to use our sketch to answer vertex queries, edge queries, path queries, and subgraph queries? How does this compare with TCM [43]?
- How efficient is it to use our sketch to answer vertex queries, edge queries, path queries, and subgraph queries? How does this compare with TCM [43]?

5.2. Experimental results

Before we present the experimental results, let us first discuss the parameters we use. We informally define the ***compression ratio*** as the sketch matrix's cell count divided by the input dataset's edge count. The compression ratio is affected only by w and d, as w and d together determine the size of the sketch. When we determine the compression ratio, w and d are set according to the number of labels in different datasets to obtain a balanced distribution. If the dataset has only a

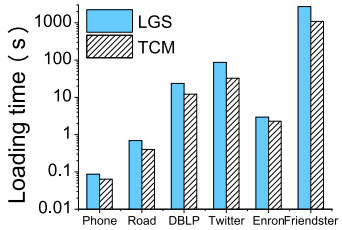

Fig. 7. Dataset loading time.

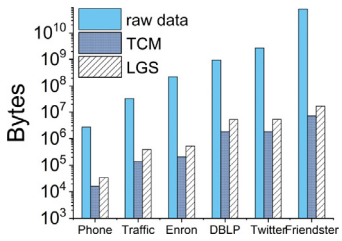

Fig. 8. Storage space needed for LGS, TCM, and raw data.

few labels, we will set a smaller w. If we set w to 1, LGS works equivalently to other algorithms that do not consider vertex labels, which indicates that LGS is an extended and more powerful scheme. The processing time of an edge insertion is only related to the number of sketches v, because v affects the number of times we need to insert the incoming edge into our sketches.

Time and Storage Efficiency:

Although our sketch can automatically handle sliding windows under the stream scenario, TCM is not capable of this. Methods that work for stream scenarios can be naturally adapted for non-stream scenarios, but the converse does not hold. Therefore, in this set of experiments, we first test the loading time needed to process all the datasets and compare the storage requirements for LGS, TCM, and the raw data. The loading time is defined as the time necessary to load the entire dataset into two sketches, LGS and TCM. We use $v = 5$ in this set of experiments. The loading times for each dataset are shown in Fig. 7. We can see from Fig. 7 that the loading times for the Phone dataset and Road dataset are both less than 1 s, whereas the loading times for the DBLP, Twitter, and Friendster dataset of LGS are approximately twice the loading times of TCM. The reason for this is that for LGS, we need to perform five hashes for each incoming edge (two hashes for the identifiers of two endpoints, two hashes for the vertex labels of two endpoints, and one for the edge label of the incoming edge). For TCM, however, only two hashes are needed (for two identifiers of the two endpoints). The need for extra processing time is reasonable, because we need to deal with more information.

In this set of experiments, we also report the storage space needed for LGS and TCM. We use the same matrix size for LGS and TCM. The parameters determining the matrix sizes are reported in Table 2. Fig. 8 shows the comparison of the storage requirements of LGS, TCM, and raw data. We can see from Fig. 8 that the storage space of LGS is approximately two to three times that of TCM. The reason for this is, for LGS, in each matrix cell, we need to store not only the number of edges, but also the prime number product result to encode the edge label information. When the product is too large, we need additional storage units. However, the storage space needed for LGS is still only two orders of magnitude less than the original data.

Fig. 9 shows the throughput for LGS under the streaming scenario, with varying v. The result is computed as the number of edges of the dataset divided by the time needed to process the dataset. Because the DBLP dataset is not ordered by timestamp and the coauthor graph does not have a timestamp, neither does the Friendster dataset. These are not under the streaming scenario. Therefore, we exclude the DBLP and Friendster data from this experiment. We show the number of edges for the other four datasets, as well as the window size and subwindow size we choose for those four datasets in Tables 3

Table 2
Matrix Size for Each Dataset.

Dataset	w^*d
Phone	25
Traffic	80
Enron	100
Twitter	300
DBLP	300
Friendster	600

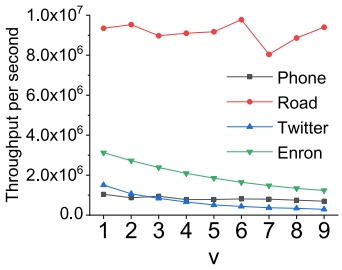

Fig. 9. LGS Throughput.

Table 3
Edge Count for Each Dataset.

	Phone	Road	Twitter	Enron
Number of edges	52,050	680,551	63,136,932	2,064,151

Table 4
Window Size and Subwindow Size for Each Dataset.

Dataset	Window size	Subwindow size
Phone	1 week	1 h
Road	1 day	5 min
Twitter	1 day	10 min
Enron	1 week	1 h

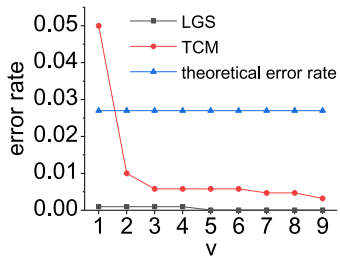

Fig. 10. Vertex Queries (varying v) Enron dataset.

and 4, respectively. The processing times for v from 1 to 9 on the Phone and Road data are all less than 0.1 s. The dataset processing time includes not only the insertion time for each edge, but also some initialization work. The initialization of the Phone dataset spans most of the data processing time, which is why the throughput of this dataset has little variation. This is similar to the Road data, except that the edges in the Road data are much more numerous (the fluctuation around $v = 7$ is due to Java's garbage collection). The Enron and Twitter datasets are sufficiently large to overcome the influence of the initialization cost. We can see that the throughput decreases slightly as v increases, but that the throughputs are all greater than 3^*10^5 edges per second. Thus, LGS is able to handle very high graph streaming rates.

Error measures: We introduce two error measures for vertex, edge, and subgraph queries. The first is an error rate. Suppose the query answer returned by the sketch is $\hat{a}_i$ (either by LGS or TCM), and the true answer is a_i; then, the error rate is computed as $\frac{\hat{a}_i - a_i}{||\mathbf{a}||_1}$, where $||\mathbf{a}||_1$ is the total number of edges. This error rate is used to compare with the theoretical error rate computed from Theorems 1, 3, and 5. The second measure is a relative error. Observe that even though the error rate ϵ is quite small (typically less than 0.01), because the dataset is large (i.e., the number of edges is large), $\epsilon||\mathbf{a}||_1$ could result in a large number, which implies a great distance between the estimated answer $\hat{a}_i$ and the true answer a_i. Thus, the relative error is computed as $|\hat{a}_i - a_i|/a_i$. We compare our method with TCM in terms of the error rate and evaluate the accuracy of LGS itself in terms of the relative error. These two error measures show identical error trends from the two aspects.

Vertex Queries:

For aggregate vertex queries on a given vertex, we first evaluate the query accuracy for both LGS and TCM without an edge label constraint. Fig. 10 shows the result from the Enron dataset. In this case, we set the compression ratio to 1/200; then, the corresponding w^*d equals 100, which gives us a theoretical error rate of $\epsilon = \frac{e}{100}$, as computed from Theorem 1. We vary the number of sketches v from 1 to 9, and the probability that the query answer error rate is less than the theoretical error rate is thus increased from 0.632 to 0.9998.

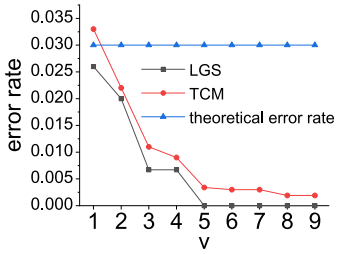

Fig. 11. Vertex Queries (varying v) Phone dataset.

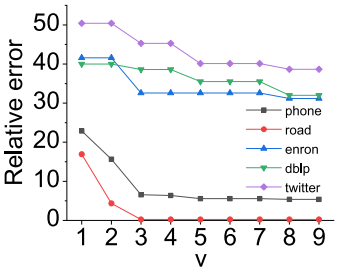

Fig. 12. Vertex Queries with edge constraints (varying v).

We see from Fig. 10 that our method, LGS, is very accurate. The reason for this is that the vertex identifiers of the Enron dataset are all email addresses and there are 11 vertex types. When there are various vertex labels, and vertex identifiers all occupy similar characteristics (all names, email addresses, numbers, etc.), LGS tends to have a high accuracy.

Fig. 11 shows the result from the Phone dataset. There are only two vertex types in this dataset; thus, LGS does not demonstrate a significant advantage over TCM, although LGS is still more accurate. We see from Figs. 10 and 11 that when v is 1, the error rate of TCM is greater than the theoretical error rate. This is because when v is 1, the probability that the error rate returned by the sketch falls below the theoretical error rate is only 0.632. The experiments on the other datasets show similar results to either the Phone or Enron datasets, and thus we omit them from the figure for clarity. All the queries are answered in less than 1 ms using our LGS. This is much quicker compared to using the original graph streams, which typically needs a few seconds in data graphs such as the Enron dataset and may require several minutes in a large data graph such as the Twitter dataset. The average response times for this type of query is given in Table 5, for compression ratio $=1/100$, and v=5. LGS runs slightly slower than TCM, because it deals with a more complicated data structure to support label-based queries, whereas TCM is not able to support queries on labeled graphs or sliding windows under the stream scenario. However, the difference is negligible compared to the time taken on raw data. For the Friendster dataset, the queries run on raw data took too long, and thus we terminated the query after one hour. We did not use the Friendster dataset for our accuracy evaluation, because it took too long to run on raw data to get a ground truth result.

Next, we evaluate the accuracy of vertex queries with edge label constraints. An example of such a query is “select the number of calls that core-person 50 made, where the call type is short voice calls” for the Phone dataset. The accuracy, in this case, is affected by the compression ratio, number of sketches v, and number of prime numbers c we use for the edge labels. We first set the compression ratio to 1/100, and c is set to no more than 20. We only vary v for the different settings.

We can see from Fig. 12 that the relative error either remains unchanged or decreases as v increases. The reason for this is that the query result is computed as the minimum number of all v sketches. It is possible that although we use more sketches, the minimum result stays unchanged. However, the trend is that as the number of sketches increases, the relative error decreases. Additionally, we see from Fig. 12 that the relative errors of the Twitter, DBLP, and Enron datasets are higher than those of the Phone and Road datasets. This is because the DBLP, Twitter, and Enron datasets have higher vertex/edge ratios, which result in more edges with different endpoints being hashed to the same cell, and hence greater relative errors.

Table 5
Response times for vertex queries.

Method	Dataset					
	Phone	Traffic	Enron	DBLP	Twitter	Friendster
LGS	63μs	86μs	115μs	135μs	155μs	320μs
TCM	49μs	76μs	88μs	118μs	138μs	298μs
Raw data	46 ms	312 ms	1909 ms	2793 ms	22936 ms	–

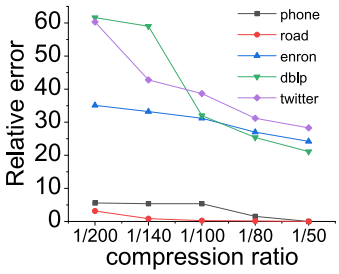

Fig. 13. Vertex Queries with edge constraints (varying compression ratio).

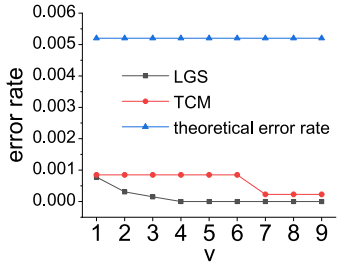

Fig. 14. Edge Queries (varying v) Phone dataset.

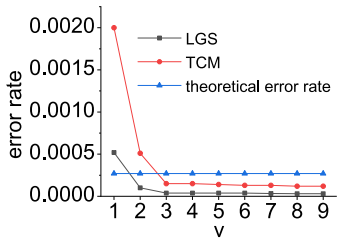

Fig. 15. Edge Queries (varying v) Enron dataset.

For the Enron dataset, we use the email subject as edge labels. This dataset has numerous edge labels, whereas we only set c as 20 to save computation time. If we set c to 100, the relative error would decrease sharply.

Next, we set v to 9 and vary the compression ratio. The results are shown in Fig. 13. We can see from the figure that the relative error decreases as the compression ratio increases. The reason for this is that when the compression ratio is high, the hash value range is larger, resulting in fewer hash collisions. In this set of experiments, all the queries could be answered in less than 0.1 s, even for a large dataset such as Friendster, as shown in Table 5. However, answering queries using original graph streams may take up to several minutes. The performance of LGS under the streaming scenario is similar to this, and hence we omit it from the figures for clarity. However, it is worth noting that, in the streaming case, for a reasonably high accuracy, the required memory space using our sketch is much lower than storing the original graph stream window. Therefore, our technique provides instant query answering and requires significantly less space consumption while maintaining high query result accuracy.

Edge Queries:

Next, we evaluate the performance of aggregate edge queries between two vertices on LGS and TCM. Queries of this type include as "select the number of emails between Mark Taylor and Mark Elliott from the Enron dataset." Fig. 14 shows the results of the Phone dataset. We set the compression ratio to 1/100 and compute the corresponding $w*d*w*d$ as well as the corresponding ϵ. We vary the number of sketches v in this set of experiments. The accuracy results from LGS are better than the accuracy of TCM. Addtionally, the error rate converges faster with LGS than TCM. Fig. 15 shows the result of the Enron dataset. The error rate is high when v is very small, exceeding the theoretical error rate. This is because when v is small, the probability that the actual error rate is less than the theoretical error rate is small. This probability increases sharply as v increases. When v is greater than 3, the error rates from both LGS and TCM are less than the theoretical error rate. Again, LGS converges much more quickly than TCM. The experiments on the other datasets show similar results, and we omit them from the figures. In this set of experiments, all the queries could be answered in less than 1 ms, compared with several seconds to a few minutes when querying on raw data graphs. The average response times for edge queries are given in Table 6, for compression ratio $=1/100$ and $v = 5$. We can see that the response times for edge queries are typically

Table 6
Response times for edge queries.

Method	Dataset					
	Phone	Traffic	Enron	DBLP	Twitter	Friendster
LGS	9μs	12μs	11μs	13μs	15μs	16μs
TCM	11μs	8μs	9μs	12μs	13μs	14μs
Raw data	51 ms	343 ms	2056 ms	3000 ms	20992 ms	–

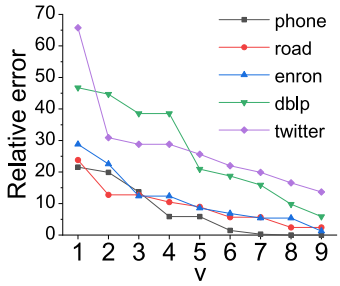

Fig. 16. Edge Queries with edge constraints (varying v).

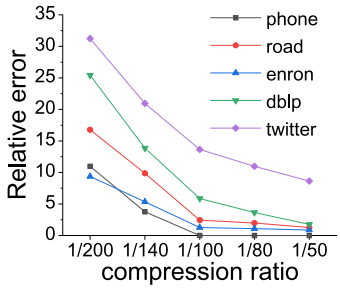

Fig. 17. Edge Queries with edge constraints (varying compression ratio).

shorter than those for vertex queries. This is because for edge queries, the number of matrix cells needed to support the queries is normally less than that for vertex queries.

Next, we perform edge queries between two vertices with edge label constraints on LGS. A sample query of this type is "select the number of emails between Mark Taylor and Mark Elliott and where the email subject is "salary" from the Enron dataset." The results are shown in Figs. 16 and 17. In 16, we set the compression ratio to 1/100 and vary v from 1 to 9. We can see that some datasets result in a high relative error, and that is because those datasets have higher vertex/edge ratios, whereas the real result values are relatively small. Thus, a few collisions could lead to large relative errors. The relative error reduces as v increases, as expected. Next, we set v to 9 and vary the compression ratio. The result is shown in Fig. 17. The relative error decreases as the compression ratio increases, as expected, and the relative error converges quickly. The accuracy results under the stream and nonstream scenarios are similar. All the queries could be answered in less than 1 ms, comparison with tens of seconds to several minutes when querying on raw data graphs.

We also evaluate other types of edge queries on the Enron dataset. Vertex type to vertex type queries with an edge constraint (referred to as VT to VT with EC) include such queries as "select the number of emails between all Employees with the subject 'P&G update' from the Enron dataset." Vertex type to vertex type queries without an edge constraint (referred to as VT to VT without EC) include such queries as "select the number of emails all managers send to all managing directors." Vertex to vertex type queries with an edge constraint (referred to as V to VT with EC) include such queries as "select the number of emails Mark Taylor sends to all Employees with subject *salary*." Vertex to vertex type queries without an edge constraint (referred to as V to VT without EC) include such queries as "select the number of emails Mark Elliott sends to all Directors." The results are shown in Figs. 18 and 19. In Fig. 18, we set the compression ratio to 1/100 and vary v from 1 to 9. In Fig. 19, we set v to 9 and vary the compression ratio. The evaluation results show similar trends to the other types of edge queries.

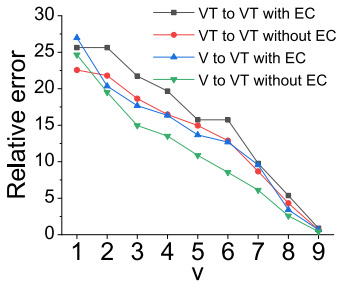

Fig. 18. Other Edge Queries (varying v) Enron Dataset.

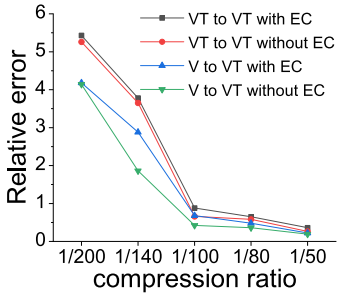

Fig. 19. Other Edge Queries (varying compression ratio) Enron Dataset.

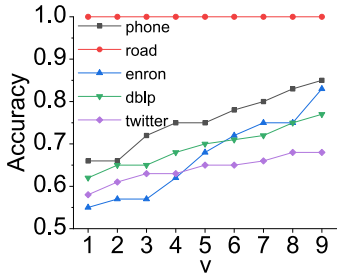

Fig. 20. Path Queries (varying v).

Table 7
Response times for path queries.

Method	Dataset					
	Phone	Traffic	Enron	DBLP	Twitter	Friendster
LGS	3μs	8μs	12μs	55μs	86μs	256μs
TCM	3μs	7μs	10μs	48μs	76μs	198μs
Raw data	23 ms	268 ms	878 ms	1356 ms	8978 ms	–

Path Queries:

Next, we perform evaluations of path (reachability) queries. In this set of experiments, for each dataset, we randomly pick 100 pairs of vertices, and the accuracy is measured by the results from these. A "right answer" from a sketch is when it returns the same answer ("reachable" or "not reachable") as the ground truth for a pair of vertices. The accuracy is the number of right answers divided by 100 and falls in [0,1]. We show the accuracy result for path queries for LGS. We first set the compression ratio to 1/100 and vary v, and the result is shown in Fig. 20. We can see that the accuracy for the Road dataset is always 1. This is because the Road dataset is a connected graph. There are no false negatives but only false positives from LGS. That is, if two vertices have a reachable path in the raw data graph, they will always be returned as being reachable with LGS. For other datasets, the accuracy is better on data graphs with lower vertex/edge ratios. As v increases, the accuracy also increases. Next, we set v to 9 and vary the compression ratio. The result is shown in Fig. 21. We can see that the accuracy increases because the collisions are reduced owing to a larger space.

We also report the response times for path queries in Table 7, for compression ratio $=$ 1/100 and $v = 5$. We can see that some queries can be answered extremely quickly, and even faster than edge queries. This is because path queries may

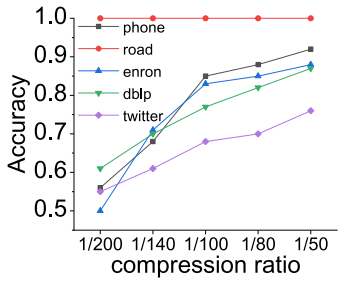

Fig. 21. Path Queries (varying compression ratio).

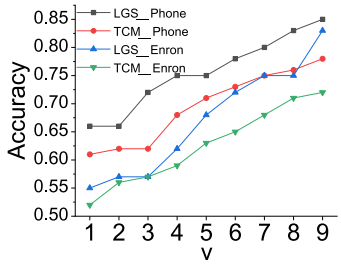

Fig. 22. Path Queries (varying v) Phone & Enron datasets.

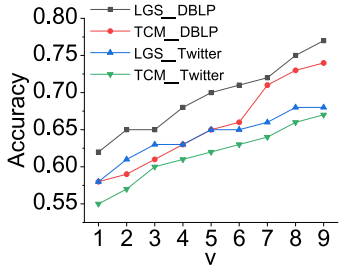

Fig. 23. Path Queries (varying v) DBLP & Twitter datasets.

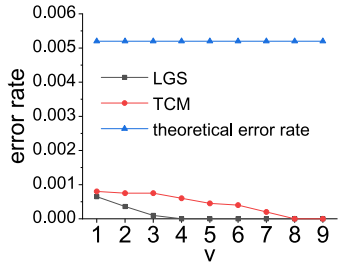

Fig. 24. Subgraph Queries (varying v) Phone dataset.

terminate early if there is no path, and they do not need to run on all v sketches, because there are no false negatives. For the queries on raw data, we store the raw data in an adjacency list to support path queries.

We show the comparison results between LGS and TCM on the four datasets in Figs. 22 and 23. We set the compression ratio to 1/100 and vary v from 1 to 9. The figures show that LGS yields better accuracy than TCM, as in the previous experiments.

Subgraph Queries:

In this set of experiments, we evaluate the performance of LGS on subgraph queries. A sample query of this type is "return the number of subgraph pattern matchings given a subgraph of size3." Fig. 24 shows the results of the Phone dataset. Fig. 25 shows the results of the Enron dataset. We adopt the same parameter settings as for edge queries. The results are similar to those of edge queries as expected, because subgraph queries can be considered as combinations of several edge queries. The theoretical error rate is computed in the same manner as for edge queries. However, the actual error rate for subgraph queries is slightly lower than that for edge queries. This is because for subgraph queries, the results

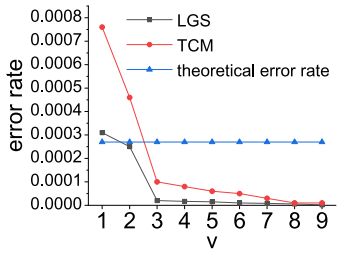

Fig. 25. Subgraph Queries (varying v) Enron dataset.

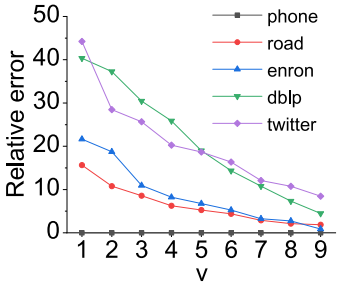

Fig. 26. Subgraph Queries with edge constraints (varying v).

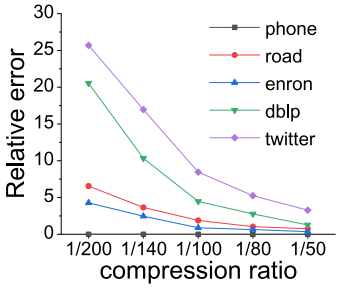

Fig. 27. Subgraph Queries with edge constraints (varying compression ratio).

are a minimum (from all v sketches) of the minimum (from all the edges in one sketch). The results of the other datasets are similar, and thus we omit them from the figures.

We report the response times needed for subgraph queries in Table 8. All the queries could be answered in less than 1 ms, and the query response times are linear with respect to the subgraph size.

Next, we perform subgraph queries with edge constraints. That is, the edges constituting the subgraph have edge label constraints. In Fig. 26, we set the compression ratio to 1/100 and vary v. The error decreases as v increases. It is worth noting that the relative error of a subgraph query on the Phone dataset is 0. This is because there is no match for the given subgraph pattern, and our sketch catches this even when there is only one sketch. In Fig. 27, we set $v = 9$ and vary the compression ratio. The error decreases as the compression ratio increases, as expected.

5.3. Summary

We have presented a comprehensive evaluation in this section. From all these experiments, we can see that LGS is capable of supporting different types of queries, including aggregate vertex queries, aggregate edge queries (either on two

Table 8
Response times for subgraph queries.

Method	Dataset					
	Phone	Traffic	Enron	DBLP	Twitter	Friendster
LGS	35μs	41μs	46μs	48μs	52μs	58μs
TCM	30μs	36μs	40μs	43μs	46μs	52μs
Raw data	168 ms	1058 ms	6875 ms	10693 ms	68986 ms	–

vertices or vertices of two types), path queries, and subgraph queries. The accuracy of LGS is better than that of the competing method TCM. All the queries supported by TCM are supported by LGS, and LGS also works under the streaming scenario and works well for highly dynamic graphs. Queries are answered instantly using our sketches, much faster than querying the original graph streams. To achieve all these benefits, LGS only needs a small amount of extra storage space to store the encoded label information in a compressed format. To the best of our knowledge, we are the first to store a labeled graph stream in a sublinear form while supporting various query types.

6. Conclusion and future work

In this paper, we first discuss the usage scenarios for aggregate vertex queries, aggregate edge queries, label based aggregate vertex/edge queries, path reachability queries, and subgraph queries on labeled graph streams. We then point out the significance of the automatic handling of item expirations, and why it is needed under the streaming scenario. Specifically, we use the sliding-window model. As data are very large and fast in the big data era, using sublinear memory space to support a diverse set of general graph queries is crucial, and requires exploring the delicate tradeoff between accuracy and efficiency. We propose a novel sketch using only sublinear storage space to support queries on labeled graphs. We provide algorithms for aggregate vertex/edge queries and labeled aggregate vertex/edge queries. We also prove that our proposed sketch could serve as a black box for other graph algorithms. In addition, we discuss how our sketch works for path reachability queries and subgraph queries. We then perform comprehensive experiments for a comparison with competing methods and show the superiority of our proposed method.

Some of our preliminary studies show that our sketch could serve as a black box for many existing graph algorithms. We would like to explore this research direction in detail in the future, and show how our sketch technique could work together with those algorithms. For some particular queries, if the true query result is small, our sketch may result in a relatively high relative error. We would like to further study nonuniform matrix distributions to reduce such errors.

Declaration of Conflict Interest

The authors of the submitted manuscript “Labeled Graph Skethces: Keeping Up with Real-time Graph Streams” hereby state that there is no financial or personal interest or belief that could affect our objectivity. There is no potential conflicts exist, either.

Acknowledgements

Chunyao Song was supported in part by the NSFC under the grants 61702285, 61772289, and U1836109, the NSF of Tianjin under the grant 17JCQNJC00200, and Jiangsu Key Laboratory of Big Data Security & Intelligent Processing, NJUPT under the grant BDSIP1902. Tingjian Ge was supported in part by the National Science Foundation under the grants IIS-1149417 (CAREER award) and IIS-1633271.

References

[1] http://www.internetlivestats.com/twitter-statistics/.
[2] http://dblp.dagstuhl.de/xml/.
[3] http://projects.csail.mit.edu/dnd/DBLP/.
[4] http://www.partow.net/programming/hashfunctions/.
[5] https://data.gov.hk/sc-data/dataset/hk-td-sm_1-traffic-speed-map.
[6] https://dev.twitter.com/docs/streaming-apis/streams/public.
[7] http://twitter4j.org/en/index.html.
[8] http://www.ahschulz.de/enron-email-data/.
[9] N.K. Ahmed, N. Duffield, J. Neville, R. Kompella, Graph Sample and Hold: A Framework for Big-Graph Analytics, in: Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2014, pp. 1446–1455.
[10] N.K. Ahmed, N. Duffield, T.L. Willke, R.A. Rossi, On sampling from massive graph streams, VLDB 10 (2017) 1430–1441.
[11] K.J. Ahn, S. Guha, A. McGregor, Spectral Sparsification in Dynamic Graph Streams, in: Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques, Springer, 2013, pp. 1–10.
[12] v. Čebirić, F. Goasdoué, I. Manolescu, Query-oriented summarization of rdf graphs, VLDB 8 (12) (2015) 2012–2015.
[13] J. Cheng, S. Huang, H. Wu, A.W.-C. Fu, Tf-Label: A Topological-Folding Labeling Scheme for Reachability Querying in a Large Graph, in: Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, ACM, 2013, pp. 193–204.
[14] J. Cheng, X. Zeng, J.X. Yu, Top-k Graph Pattern Matching Over Large Graphs, in: 2013 IEEE 29th International Conference on Data Engineering, IEEE, 2013, pp. 1033–1044.
[15] G. Cormode, S. Muthukrishnan, An improved data stream summary: the count-min sketch and its applications, J. Algorithms 55 (1) (2005) 58–75.
[16] N. Eagle, A. Pentland, Crawdad dataset mit/reality (v. 2005-07-01), Downloaded from http://crawdad.org/mit/reality/20050701.
[17] W. Fan, C. Hu, Big graph analyses: from queries to dependencies and association rules, Data Sci. Eng. 2 (1) (2017) 36–55.
[18] W. Fan, J. Li, X. Wang, Y. Wu, Query Preserving Graph Compression, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, ACM, 2012, pp. 157–168.
[19] W. Fan, X. Wang, Y. Wu, Diversified top-k graph pattern matching, VLDB 6 (13) (2013) 1510–1521.
[20] O. Frank, Sampling and estimation in large social networks, Soc. Netw. 1 (1978) 91–101.
[21] M. Gupta, J. Gao, X. Yan, H. Cam, J. Han, Top-k Interesting Subgraph Discovery in Information Networks, in: 2014 IEEE 30th International Conference on Data Engineering, IEEE, 2014, pp. 820–831.
[22] M. Jha, C. Seshadhri, A. Pinar, A Space Efficient Streaming Algorithm for Triangle Counting using the Birthday Paradox, in: Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2013, pp. 589–597.

[23] R. Jin, Y. Xiang, N. Ruan, H. Wang, Efficiently Answering Reachability Queries on Very Large Directed Graphs, in: Proceedings of the 2008 ACM SIGMOD International Conference on Management of data, ACM, 2008, pp. 595–608.
[24] R. Jin, H. Hong, H. Wang, N. Ruan, Y. Xiang, Computing Label-Constraint Reachability in Graph Databases, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, ACM, 2010, pp. 123–134.
[25] B.M. Kapron, V. King, B. Mountjoy, Dynamic Graph Connectivity in Polylogarithmic Worst Case Time, in: Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, 2013, pp. 1131–1142.
[26] A. Khan, C. Aggarwal, Query-Friendly Compression of Graph Streams, in: Proceedings of the 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining, IEEE Press, 2016, pp. 130–137.
[27] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, I. Stoica, Zipg: A Memory-Efficient Graph Store for Interactive Queries, in: Proceedings of the 2017 ACM International Conference on Management of Data, 2017, pp. 1149–1164.
[28] M. Lay, Dblp - some lessons learned, VLDB 2 (2) (2009) 1493–1500.
[29] Y. Lim, U. Kang, Mascot: Memory-Efficient and Accurate Sampling for Counting Local Triangles in Graph Streams, in: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2015, pp. 685–694.
[30] Y. Lim, M. Jung, U. Kang, Memory-efficient and accurate sampling for counting local triangles in graph streams: from simple to multigraphs, TKDD 12 (4) (2018).
[31] S. Maneth, F. Peternek, Compressing Graphs by Grammars, in: 2016 IEEE 32nd International Conference on Data Engineering, IEEE, 2016, pp. 109–120.
[32] A. McGregor, Graph stream algorithms: a survey, SIGMOD Record 43 (1) (2014) 9–20.
[33] R. Motwani, P. Raghavan, Randomized Algorithms, Cambridge University Press, 1995.
[34] S. Muthukrishnan, Data streams: algorithms and applications, Found. Trend. Theor. Comput. Sci. 1 (2) (2005) 117–236.
[35] P. Nayek, P. Nayek, A. Bhattacharya, Neighbor-Aware Search for Approximate Labeled Graph Matching using the Chi-Square Statistics, in: Proceedings of the 26th International Conference on World Wide Web, 2017, pp. 1281–1290.
[36] H. Reittu, I. Norros, T. Räty, M. Bolla, F. Bazsó, Regular decomposition of large graphs: foundation of a sampling approach to stochastic block model fitting, Data Sci. Eng. (2019) 1–17.
[37] X. Ren, J. Wang, Multi-query optimization for subgraph isomorphism search, VLDB 10 (3) (2016) 121–132.
[38] R. Rubinfeld, Pairwise independent hash functions, https://people.csail.mit.edu/ronitt/COURSE/S12/handouts/lec5.pdf.Lecture Note.
[39] N. Shah, D. Koutra, T. Zou, B. Gallagher, C. Faloutsos, Timecrunch: Interpretable Dynamic Graph Summarization, in: Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, 2015, pp. 1055–1064.
[40] L. Shi, S. Sun, Y. Xuan, Y. Su, H. Tong, S. Ma, Y. Chen, Topic: Toward Perfect Influence Graph Summarization, in: 2016 IEEE 32nd International Conference on Data Engineering, IEEE, 2016, pp. 1074–1085.
[41] C. Song, T. Ge, Labeled graph sketches, ICDE Poster (2018).
[42] C. Song, T. Ge, C. Chen, J. Wang, Event pattern matching over graph streams, VLDB 8 (4) (2014) 413–424.
[43] N. Tang, Q. Chen, P. Mitra, Graph Stream Summarization: From Big Bang to Big Crunch, in: Proceedings of the 2016 International Conference on Management of Data, ACM, 2016, pp. 1481–1496.
[44] R.R. Veloso, L. Cerf, W. Meira Jr, M.J. Zaki, Reachability queries in very large graphs: a fast refined online search approach, EDBT (2014) 511–522.
[45] J. Wang, N. Ntarmos, P. Triantafillou, Indexing query graphs to speedup graph query processing, in: Proceedings of the 19th International Conference on Extending Database Technology, 2016, pp. 41–52.
[46] J. Yang, J. Leskovec, Defining and Evaluating Network Communities Based on Ground-Truth, in: IEEE 12th International Conference on Data Mining, IEEE, 2012, pp. 745–754. URL http://snap.stanford.edu/data/com-Friendster.html.
[47] Z. Yang, A.W.-C. Fu, R. Liu, Diversified Top-k Subgraph Querying in a Large Graph, in: Proceedings of the 2016 International Conference on Management of Data, ACM, 2016, pp. 1167–1182.
[48] X. Zhang, L. Chen, Distance-aware selective online query processing over large distributed graphs, Data Sci. Eng. 2 (1) (2017) 2–21.
[49] P. Zhao, C.C. Aggarwal, M. Wang, Gsketch: on query estimation in graph streams, VLDB 5 (3) (2011) 193–204.
[50] A.D. Zhu, W. Lin, S. Wang, X. Xiao, Reachability Queries on Large Dynamic Graphs: A Total Order Approach, in: Proceedings of the 2014 ACM SIGMOD International Conference on Management of data, ACM, 2014, pp. 1323–1334.