Labeled Graph Sketches

Chunyao Song
College of Computer & Control Engineering, Nankai University
Email: chunyao.song@nankai.edu.cn

Tingjian Ge
Computer Science, University of Massachusetts, Lowell
Email: ge@cs.uml.edu

***Abstract*—Nowadays, a graph serves as a fundamental data structure for many applications. As graph edges stream in, users are often only interested in the recent data. In data exploration, how to store and process such massive amounts of graph stream data becomes a significant problem. As vertex and edge attributes are often referred to as labels, we propose a labeled graph sketch that stores real-time graph structural information in sub-linear space and supports queries of diverse types. This sketch also supports sliding window queries. We conduct experiments on three real-world datasets, comparing with a state-of-the-art method to show the superiority of our sketch.**

I. Introduction

A graph serves as a fundamental data structure 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 these applications, vertices and edges are often associated with a label/type. For data exploration, how to store and process this huge amount of data becomes a major problem. For example, in an email network, one may want to know the email frequency between two people within a recent period, or the number of emails a user sends on a certain topic. We call such queries **(label based) edge/vertex queries.** Now consider a road traffic network, where a user may be interested in whether there is a path from s to t only through roads with good traffic conditions. We call this a **path query**.

In all these applications, data are continuously produced at a fast rate. It is reported that there are about 500 million tweets per day and over 300 billion tweets in total as of 2013 [1]. Storing all needed 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 [2]. We propose a labeled graph sketch which stores graph stream information in sublinear space, while supporting many types of graph queries such as those mentioned above. Our sketch structure also supports the sliding window semantics, often needed in practice.

Related work. Some previous work on graph sketches maintains 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. [3] is an excellent survey for this line of work. However, each of such work is only for a specific problem, such as finding a spanning forest of a graph, testing k-connectivity, min-cut, and sparsification. Thus, each of them only needs to preserve few data features to solve a specific problem. Moreover, they do not deal with labeled graphs.

Some other sketch work also applies linear projections of data, but tries to preserve salient features of data using the CountMin sketch [4]. A CountMin sketch is originally used to summarize general data streams. The gSketch [5] extends it to support graph streams. It first treats each edge as an element with a unique identifier, and then uses CountMin. The gMatrix [6] and TCM [7] are proposed which both use a 3-dimensional sketch. Instead of treating the graph edge set as an element set and mapping each edge into a one-dimensional space, gMatrix and TCM apply hash functions which define a mapping of the graph vertex set to an integer in $1...w$. However, they do not handle vertex/edge labels. Also, they do not handle automatic edge expiration for a sliding window. TCM [7] is the state-of-the-art method most related to our work. Thus, we compare against TCM in experiments.

Other graph compression/summarization work includes [8], where Fan et al. propose a query preserving graph compression, especially for path reachability queries and bounded simulation pattern matching. However, it needs to know the whole data graph in advance. Shah et al. propose TimeCrunch [9] to summarize important temporal structures for dynamic graphs and try to find patterns that agree with intuition. Their target is different from ours. Some other work includes [10], [11], [12]. As we will show in Sec. IV, our sketch can serve as a black box for most of the above algorithms.

Our contributions. We formulate the problem (**Sec. II**) and propose a CountMin based labeled graph sketch (LGS) which stores the original data graph in only sublinear space while maintaining the structural information and supporting many types of graph queries (**Sec. III**). We then devise algorithms to answer various types of queries (**Sec. IV**). Finally, we conduct experiments on real-world datasets (**Sec. V**).

II. 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 $L_v(A)$, while 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. Each edge e also has an edge label $L_e(e)$. 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 as $\{v_1, v_2, ..., v_n\}$ and E is a set of edges as $\{e_1, e_2, ..., e_m\}$. Each vertex v_i has a vertex label $L_v(v_i)$ from vertex label set Σ_N and each edge e_i has an edge label $L_e(e_i)$ from edge label set Σ_E. A graph stream G is sorted by their edge incoming

time, in non-decreasing 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 will automatically discard edges with incoming time older than $t-W$. Our work applies to non-window settings too.

III. A 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 endpoints' identifiers, endpoints' labels and edge labels for each incoming edge when encoding edges into the sketch. We use hash functions from a pair-wise independent hash family to process these three factors. Specifically, we use two-level hashes for vertices' labels, and a product of prime numbers to encode edge labels. For sliding windows, we need to handle item expiration. As a very fine time granularity is typically unnecessary, we divide the whole window into k sub-windows based on the application, and maintain a counter for each sub-window. Moreover, we use v sketches to reduce errors. Algorithm 1 builds the sketch online.

Algorithm 1: LabeledGraphSketch

Input : A graph stream G, Window size W, Sub-window size W_s.
Output: Labeled Graph Sketch 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
```

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

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

IV. Answering Queries

Before we discuss in detail the queries that our sketch supports, let us first introduce an auxiliary process which is used to get the number of edges from a given vertex to another. All edges with the same two endpoints will be 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.

Algorithm 2: GetEdgeCount

Input : An edge label l (optional), A hash function $\boldsymbol{h}$ (optional), A prime number list P (optional), A 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
```

The input to GetEdgeCount includes an optional edge label l to match and 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, 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.

Vertex Queries. Our sketch supports all kinds of aggregate vertex queries, including edge label based or non-edge-label based aggregate single vertex queries (e.g., the in-degree of a certain vertex), 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 vertices of a certain type). To get a specific vertex's incoming degree, we apply $\boldsymbol{h}$ twice to map

this vertex to a particular column. Then we sum up all the cell values in this column. The cell value is computed from GETEDGECOUNT either with or without an edge label. To get a specific vertex's out degree, we just need to sum up the corresponding row values. Note that we use v groups of hash functions to distinguish collisions. The final value is the minimum one we find among all these v sketches. Since we group the vertices of the same type together, we also support aggregate vertex queries for different vertices with the same label. In that case, we only need to sum up the corresponding rows/columns together.

The accuracy guarantee follows the one for CMS [4]. For non-edge-label based aggregate vertex queries on a given vertex, the estimated aggreagate 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 $v = \lceil ln\frac{1}{\delta} \rceil$, 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. The time complexity of answering non-edge-label/edge-label-based aggregate vertex queries on a given vertex, and non-edge-label/edge-label-based aggregate vertex queries on a certain vertex label are $O(w \cdot d \cdot k)$, $O(w \cdot d \cdot k + i)$, $O(w^2 d)$ and $O(w^2 d + i)$, respectively, where i is the query result.

Edge Queries. Our sketch supports various kinds of aggregate edge queries, including the following: aggregate edge queries between two vertices, with or without a required edge label; aggregate edge queries between a vertex and vertices of a certain label, with or without a required edge label; and aggregated edge queries between vertices of two types, with or without a required edge label.

For all the query types above, we just need to locate the corresponding matrix cells from the sketch and retrieve the required information using GETEDGECOUNT. The final result is the minimum value among all v sketches.

The accuracy guarantee also follows the one for CMS [4]. In addition, 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 sub-windows in a matrix cell. The time complexity for answering edge-label-based aggregate edge queries between two vertices, non-edge-label/edge-label-based aggregate edge queries between a vertex and a vertex type, non-edge-label/edge-label-based aggregate edge queries between two vertex types are $O(k+i)$, $O(d \cdot k)$, $O(d \cdot k + i)$, $O(d^2 k)$ and $O(d^2 k + i)$ respectively, where i is the query result.

Path Queries. Since 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. We can use GETEDGECOUNT to check whether there are (required labeled) edges between two vertices, then run DFS to check if there is a path from vertex A to vertex B. Note there is no false negative and only false positive, i.e., if B is not reachable from A returned by the algorithm, then B is indeed not reachable from A. On the other hand, if B is reachable from A returned by the algorithm, it is possible that B is not indeed reachable from A.

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. If there indeed is a path from vertex A to vertex B, because of the existence of each connecting edge, the corresponding hash cell value would be greater than or equal to 1, so the sketch can also find it as a path. On the other hand, 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.

Suppose our algorithm finds a path (i.e., sequence) P of l positive edges in the sketch. 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 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 and vertex label hash value range size, respectively.

This gives the result for one sketch. When there are v sketches, they may return different paths (of different lengths) in their sketches. 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 since the v sketches are independent.

Discussions. Our sketch encodes a broad range of information from the graph stream, including not only edge and vertex connection information, but also edge and vertex label information. Thus our sketch can be used as a black box for many existing graph algorithms. For example, this sketch may be applied to subgraph pattern matching, where it is used as a pre-filter. Only if all edges in the query pattern have their mapping edges in the sketch, do we do the verification.

In our scheme, sub-windows are used to handle the sliding window scenario. It is also possible to handle non-sliding window situation without the use of sub-windows. In that case, an update includes both insertion and deletion, and deletion is just the reverse operation of insertion. Without sub-windows, 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 this non-sliding 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. Also, when answering queries, the time complexity is also less since we do not need to multiply the sub-window number k. This is because we no longer need to sum up all 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 less than in the sliding window case.

V. EXPERIMENTS

We use three real world datasets, **Phone data** [13], **Enron data** [14] and **Twitter data** in our experiments. For the last one, we use the Twitter Stream API to retrieve the messages with hashtags and resulting in a total size of about 36GB. We

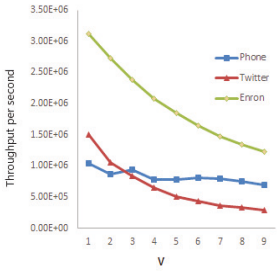

Fig.1. LGS Throughput

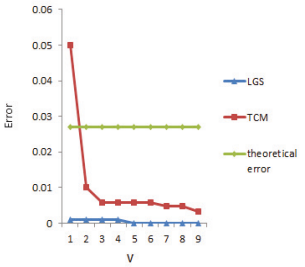

Fig.2. Vertex Queries(varying v) Enron dataset

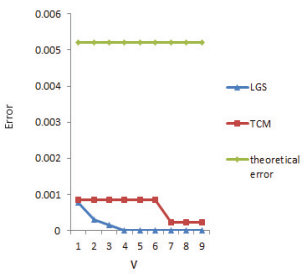

Fig.3. Edge Queries (varying v) Phone dataset

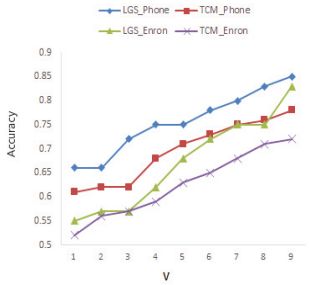

Fig.4. Path Queries (varying v) Phone & Enron datasets

compare with the most related state-of-the-art approach TCM [7]. All experiments are performed on a machine with an Intel Core i7 3.40 GHz processor and a 16GB memory.

We first examine the efficiency of LGS. The processing time of an edge insertion is only related to the number of sketches v, since v affects the number of times we need to insert the incoming edge into our sketches. Fig. 1 shows the throughput for LGS under the streaming scenario, varying v. The result is computed as the number of edges of the dataset divided by the time needed to process that dataset. The processing times for v from 1 to 9 on the Phone data are less than 0.1s. 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. 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 the throughputs are all greater than $3*10^5$ edges per second. LGS is able to handle very high rate graph streams.

We then examine the 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. 2 shows the result from Enron dataset. We compute the theoretical error and vary the number of sketches v from 1 to 9; thus the probabilities that query answer error is less than the theoretical error is increased from 0.632 to 0.9998. 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 is computed as $\frac{\hat{a_i}-a_i}{||\mathbf{a}||_1}$, where $||\mathbf{a}||_1$ is the number of total edges. We see from Fig. 2 that our method, LGS, is very accurate. The reason is that the vertex identifier of 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, all email addresses, all numbers, etc.), LGS tends to have a high accuracy.

Next, we evaluate the performance of aggregate edge queries between two vertices on LGS and TCM. Fig. 3 shows the result of the Phone dataset. We vary the number of sketches v in this set of experiments. The accuracy from LGS is better than that of TCM. Moreover, the error converges faster with LGS than TCM. We also perform edge queries between two vertices with edge label constraints on LGS under the stream scenario. As expected, as v grows, the accuracy increases and it shows a similar trend as in Fig. 3; we omit it from the figure for clarity.

Next, we perform evaluations on the 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 100 pairs of vertices. 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 comparison results between LGS and TCM on Phone and Enron datasets in Fig. 4. It shows that LGS obtains better accuracy than TCM as in previous experiments.

Acknowledgement. Chunyao Song was supported by the NSFC under the grants 61702285 and 61772289 and the NSF of Tianjin under the grants 17JCQNJC00200. Tingjian Ge was supported by NSF grants IIS-1149417 (CAREER award) and IIS-1633271.

REFERENCES

[1] http://www.internetlivestats.com/twitter-statistics/.
[2] S. Muthukrishnan, "Data streams: Algorithms and applications," *Foundations and Trends in Theoretical Computer Science*, vol. Vol. 1, No. 2, pp. 117–236, 2005.
[3] A. McGregor, "Graph stream algorithms: A survey," *SIGMOD Record*, vol. Vol. 43, No. 1, pp. 9–20, 2014.
[4] G. Cormode and S. Muthukrishnan, "An improved data stream summary: The count-min sketch and its applications," *J. Algorithms*, 2005.
[5] P. Zhao, C. C. Aggarwal, and M. Wang, "gsketch: On query estimation in graph streams," *VLDB*, 2011.
[6] A. Khan and C. Aggarwal, "Query-friendly compression of graph streams," *ASONAM*, 2016.
[7] N. Tang, Q. Chen, and P. Mitra, "Graph stream summarization: From big bang to big crunch," *SIGMOD*, 2016.
[8] W. Fan, J. Li, X. Wang, and Y. Wu, "Query preserving graph compression," *SIGMOD*, 2012.
[9] N. Shah, D. Koutra, T. Zou, B. Gallagher, and C. Faloutsos, "Timecrunch: Interpretable dynamic graph summarization," *KDD*, 2015.
[10] S. Cebiric, F. Goasdoue, and I. Manolescu, "Query-oriented summarization of rdf graphs," *VLDB*, 2015.
[11] L. Shi, S. Sun, Y. Xuan, Y. Su, H. Tong, S. Ma, and Y. Chen, "Topic: Toward perfect influence graph summarization," *ICDE*, 2016.
[12] S. Maneth and F. Peternek, "Compressing graphs by grammars," *ICDE*, 2016.
[13] N. Eagle and A. Pentland. Crawdad dataset mit/reality (v. 2005-07-01), downloaded from http://crawdad.org/mit/reality/20050701.
[14] A. H. Ruhe. http://www.ahschulz.de/enron-email-data/.