



Article

Dichotomy Graph Sketch: Summarizing Graph Streams with High Accuracy Based on Deep Learning [†]

Ding Li ¹ , Wenzhong Li ^{2,*} , Guoqiang Zhang ³, Yizhou Chen ², Xu Zhong ², Mingkai Lin ² and Sanglu Lu ²

¹ School of Computer Science and Information Engineering, Hubei University, Wuhan 430062, China; liding@hubu.edu.cn

² State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China; mf20330010@smail.nju.edu.cn (Y.C.); xuzhong@smail.nju.edu.cn (X.Z.); mingkai@smail.nju.edu.cn (M.L.); sanglu@nju.edu.cn (S.L.)

³ School of Information Science and Technology, Hainan Normal University, Haikou 571158, China; zgqoop_cn@hotmail.com

* Correspondence: lwz@nju.edu.cn

[†] This paper is an extended version of the paper published in Ding Li; Wenzhong Li; Yizhou Chen; et al. Learning-Based Dichotomy Graph Sketch for Summarizing Graph Streams with High Accuracy. In Proceedings of the 16th International Conference on Knowledge Science, Engineering and Management (KSEM 2023), Guangzhou, China, 16–18 August 2023.

Abstract: In many applications, data streams are indispensable to describe the relationships between nodes in networks, such as social networks, computer networks, and hyperlink networks. Fundamentally, a graph stream is a dynamic representation of a graph, which is usually composed of a sequence of edges, where each edge is represented by two endpoints and a weight. As a result of its large volume and highly dynamic nature, several graph sketches were proposed for the purposes of summarizing large-scale graph streams and enabling fast query processing. By using a compact data structure with hash functions, the graph sketches sequentially store the edges. Nevertheless, the existing graph sketches suffer from low performance on graph query tasks as a result of unpredictable collisions between heavy edges and light edges. To store heavy edges and light edges, this paper introduces a novel learning-based Dichotomy Graph Sketch (DGS) mechanism that uses two separate graph sketches, a *heavy sketch* and a *light sketch*. During a graph stream session, DGS obtains heavy edges and light edges, and uses these edges as training samples for a deep neural network (DNN) based binary classifier. The DNN-based classifier is then used to determine whether the upcoming edges are heavy or not. We will store the edges that are classified as heavy edges in the heavy sketch, and those that are classified as light edges in the light sketch. By combining the learnable classifier and Dichotomy Graph Sketches, the proposed mechanism resolves the hashing collision problem in conventional graph sketches and significantly improves graph query accuracy. The DGS algorithm outperforms the state-of-the-art graph sketches in a variety of graph query tasks based on extensive experiments that were conducted on four real-world graph stream datasets.

Keywords: graph stream; graph stream summarization; sketch; graph sketch; deep learning



Citation: Li, D.; Li, W.; Zhang, G.; Chen, Y.; Zhong, X.; Lin, M.; Lu, S. Dichotomy Graph Sketch: Summarizing Graph Streams with High Accuracy Based on Deep Learning. *Appl. Sci.* **2023**, *13*, 13306. <https://doi.org/10.3390/app132413306>

Academic Editor: Tobias Meisen

Received: 26 October 2023

Revised: 7 December 2023

Accepted: 11 December 2023

Published: 16 December 2023



Copyright: © 2023 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

In many data stream applications, connections are essential in describing relationships within networks, including social networks, computer networks and communication networks. Unlike traditional data streams that are modeled as isolated items, the data in these applications are organized as *graph streams* [1,2]. A graph stream can form a dynamic graph that changes with every arrival of an edge. The network traffic, for instance, can be viewed as a graph stream in which each item represents a communication between two IP addresses. If a new packet is received, then the network traffic graph will be updated. Another example would be the interactions between users of a social network. These

interactions form a dynamic graph. When new interactions occur, the social network graph will be dynamically updated. A graph stream, in formal terms, comprises a sequence of items, where each item represents an edge of the graph, usually represented by a tuple consisting of two endpoints and a weight.

Due to the increasing size of graphs, it is necessary to build an accurate but small representation of the original graph so that analytics can be performed more efficiently. In order to accomplish this goal, researchers studied the *graph summarization* problem in order to maintain the overall structure of the graph, while reducing its size. For example, Merchant et al. proposed a novel algorithm called SpecSumm [3] for graph summarization via node aggregation. Additionally, Hajiabadi et al. have developed a utility-driven graph summarization method G-SCIS [4] that produces optimal compression with no loss of utility. They focused primarily on summarizing static graphs, which were incapable of handling dynamic graphs formed by graph streams. To this end, some researchers proposed *graph sketch* as a way to summarize graph streams. For example, Zhao et al. proposed gSketch which utilized CM-sketch [5] to support edge query and aggregated subgraph query. In addition, Tang et al. proposed a novel graph sketch called TCM [1]. Based on a hash function, TCM mapped each edge to a bucket in a matrix, and each bucket was assigned the edge weight. Since TCM kept topology information for the graph, it supported not only edge queries but also node queries. In a recent work, Gou et al. proposed GSS [2], a method that combines an adjacency list buffer with a matrix to improve edge query accuracy. If a hash collision occurred in the matrix, then the adjacency list buffer stored the edge. Their follow-up study has been published in [6]. In this study, the matrix used in GSS was partitioned into multiple blocks, and the query was accelerated using bitmaps and FPGA (field-programmable gate array).

Typical graph sketches use a random hash function to map each edge to a bucket of a matrix, and then record the edge weight within the bucket, as illustrated in Figure 1. As a result of the random nature of the hash function, hash collisions may occur when querying an edge or a node with the graph sketch. There will be a significant performance degradation on query tasks if a heavy edge (an edge with a large weight) collides with a light edge (an edge with a small weight). In the example presented in Figure 1, the graph stream contains one heavy edge (edge e_2) with a weight of 100 and three light edges (edges e_1 , e_3 and e_4). It is possible for a graph sketch to map a heavy edge e_1 and a light edge e_2 to the same bucket of the matrix due to a hash collision. In graph sketch, the bucket stores the sum of the edge weights (see Section 3 for details); therefore, the value recorded in this bucket is $1 + 100 = 101$. As a result, if we query the weight of edge e_1 , the graph sketch will return the value 101, and the relative error (see Section 5.2.1 for details) of this query is $(101 - 1)/1 = 100$, which is extraordinary high for a light edge. We further test the performance of TCM [1], a state-of-the-art graph sketch, on two real-world datasets, *subelj_jung* (http://konect.cc/networks/subelj_jung-j/, access date: 10 December 2023) and *subelj_jdk* (http://konect.cc/networks/subelj_jdk/, access date: 10 December 2023), and the results are shown in Figure 2. It is demonstrated that TCM performs poorly on edge query tasks, and the average relative error when querying light edges is extremely high. The average relative error dramatically increases with the increase in compression ratio (i.e., reducing the size of the graph sketch). Thus, a hash collision between a heavy edge and a light edge will result in an unacceptably high query error for the light edge, which is a major limitation of conventional graph sketches.

To address the above issue, it is desirable to design a new graph sketch mechanism to resolve the high query error caused by hash collisions. (This paper is a significant extension of the conference paper published in [7]. In this paper, significant modifications have been made to improve the practicability of the research problem, enhance the solution framework and algorithm with rationale, and provide additional experimental results with detailed analysis.) In this paper, we propose a novel Dichotomy Graph Sketch (DGS) approach, which is able to differentiate heavy edges and light edges during the edge recording process and, respectively, store them in two separate matrices to avoid hash collisions. DGS first

determines the type of edge by using the *edge classifier* when a new edge arrives. If the edge is classified as a heavy edge, it will be stored in the *heavy sketch*; otherwise, it will be stored in the *light sketch*. In order to train the edge classifier, we utilize a *sample generator* which is comprised of a temporal graph sketch and two min-heaps. At the end of each session, we obtain the heavy edges from the sample generator (see Section 4 for details) in order to train the edge classifier. In this way, the hash collision in Figure 1 can be avoided and the query performance can be improved.

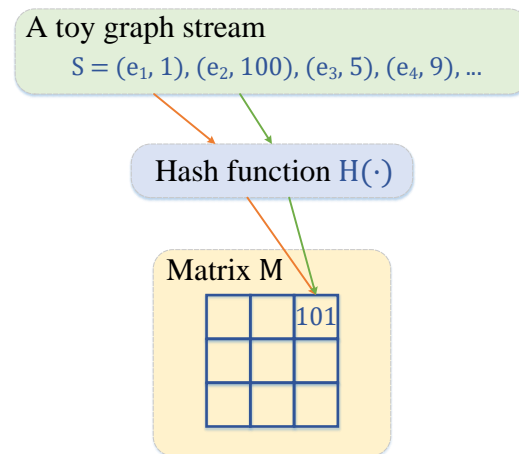


Figure 1. Existing graph sketch stores all edges in a single matrix.

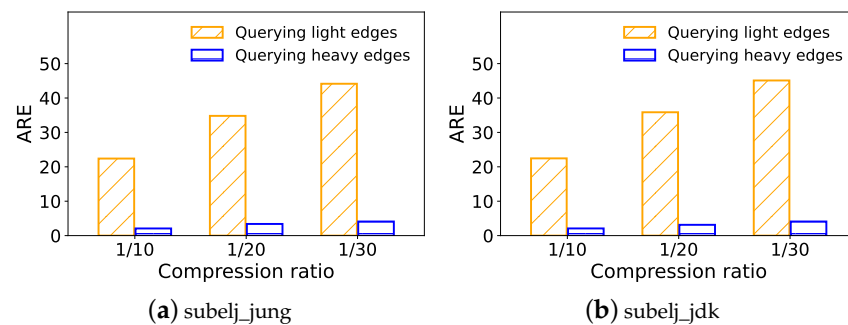


Figure 2. Average relative error (ARE) of querying light edges and querying heavy edges using TCM on dataset *subelj_jung* and *subelj_jdk*.

The main results and contributions of this paper are summarized as follows:

- As a novel method for summarizing graph streams, we propose Dichotomy Graph Sketch (DGS). It is capable of distinguishing heavy edges from light edges during edge updating.
- We adopt deep learning techniques in our graph sketch design. In particular, we design a novel neural network architecture to detect heavy edges during the edge updating process, which allows us to store heavy edges and light edges in two separate matrices.
- A comprehensive evaluation of the performance of DGS is conducted using three real-world graph streams. According to the experimental results, DGS outperforms state-of-the-art graph sketches in a variety of graph query tasks.

2. Related Work

We summarize the related work into two categories: data stream sketches and graph stream sketches.

2.1. Data Stream Sketches

Data stream sketches are designed for data stream summarization. Differently from graph streams, data streams are modeled as isolated items, and thus data stream sketches

cannot answer graph topology queries. C-sketch [8] utilized several hash tables to record data streams, but suffered from both overestimation and underestimation in the frequency estimation task. Cormode et al. then proposed CM-sketch [5], which only suffered from overestimation. Estan et al. designed CU-sketch [9] which improved CM-sketch's query accuracy at the cost of not supporting item deletions. The above sketches focused on solving the problem of frequency estimation. However, in various scenarios, we need to find lots of other important information in data streams. For instance, in financial markets, a burst of trading volume may indicate the happening of financial fraud or illegal market manipulation. For another example, persist items in the data streams can be used to detect malicious behaviors in the network security field [10]. Li et al. designed a generic algorithm, WavingSketch [11], and considered four query tasks: finding top-k frequent items, finding top-k heavy changes, finding top-k persistent items and finding top-k super-spreaders. Zhang et al. proposed On-Off sketch [12] which focused on the problem of persistence estimation and finding persistent items by a technique to separate persistent and non-persistent items. Similar to CM-Sketch, On-Off sketch adopted d hash tables to simultaneously summarize data stream. The difference is that a bucket of On-Off sketch can be updated at most once in one time window to avoid overestimation. More recently, Zhong et al. proposed BurstSketch [13] to detect bursts accurately in real time. BurstSketch used several hash tables to record a data stream, and find the potential bursts based on the records. Finally, BurstSketch utilized other two hash tables to record the frequency of potential bursts in two consecutive time windows to find real bursts. Liu et al. proposed the concept of periodic batch, namely α consecutive batches of the same item, where these batches arrive periodically, and they designed HyperCalm Sketch [14] to detect periodic batches in data streams. Different from most data streams sketches which found local top-k frequent items in a single data stream, Double-Anonymous Sketch [15] was designed to find global top-k frequent items in multiple disjoint data streams. Double-Anonymous Sketch consisted of two parts: a Randomized Admission Policy (RA) [16] as the top-k part and a CMM sketch [17] as the count part. RA was used to report top-k items, while CMM sketch was used to report an item's frequency. Wang et al. proposed JoinSketch [18] for inner-product estimation. JoinSketch considered the high skewness of real data, and recorded items with different frequencies in different components to improve accuracy. Li et al. designed SteadySketch [19] to find the steady items in data streams. SteadySketch was comprised of SteadyFilter and RollingSketch. SteadyFilter was responsible for continuity checking of small flows, while RollingSketch was responsible for continuity checking of large flows. Fan et al. proposed PISketch [20,21] to find persistent but infrequent item in data streams. Its key idea was to define a weight and its Reward and Penalty System for each flow to combine and balance the information of both persistency and infrequency, and to keep high-weighted flows in a limited space through a strategy.

Recently, researchers proposed a new research problem, i.e., summarizing data streams in a sliding window, since—in some scenarios—older data are less valuable. In [22], Zhao et al. proposed Stair Sketch which organized the memory used by different time periods in the shape of stairs in order to memorizing recent events with higher accuracy. In addition, Gou et al. designed a generic framework, Sliding sketches [23], which could be applied to many existing solutions for membership query, frequency query and heavy hitter query, and enabled them to support queries in sliding windows. Fan devised HoppingSketch [24] which improved the original persistent bloom filter [25], and focused on answering temporal membership query.

Besides designing novel data stream sketches, researchers also tried to devise some generic optimization method in order to improve the performance of the existing data stream sketches. For example, Liu et al. proposed SF-Sketch [26] which consisted of a Fat-subsketch and a Slim-subsketch. The Fat-subsketch was used for updating and periodically producing the Slim-subsketch, which was then transferred to the remote collector for answering queries quickly and accurately. Miao et al. proposed SketchConf [27], an automatic sketch configuration framework, which efficiently generated memory-optimal

configurations, and could be applied to order-independent sketches. Stingy Sketch [28] was a new sketch framework which utilized bit-pinching counter tree and prophet queue to optimize both the accuracy and speed for frequency estimation task. To enable the sketch fit into the on-chip memory, Yang et al. proposed a generic technique, self-adaptive counters [29,30] (SA Counter) which could be applied to typical sketches to improve their accuracy.

With the rapid development of the internet, researchers have also applied data stream sketch to *network measurement*. Liu et al. proposed UnivMon [31] which was able to simultaneously completed several measurement tasks including heavy hitter detection, DDos detection, heavy changer detection and entropy estimation. Then, Huang et al. proposed SketchVisor [32], which was a robust network measurement framework for software packet processing. It augmented sketch-based measurement in the data plane with a fast path, which was activated under high traffic load to provide high-performance local measurement with slight accuracy degradations. Tang et al. designed MV-sketch [33] which tracked candidate heavy items inside the sketch data structure via the idea of majority voting. In addition, Yang et al. proposed Elastic Sketch [34,35] which was adaptive to various traffic characteristics including available bandwidth, packet rate, and flow size distribution. Liu et al. claimed that sketches incurred significant computation overhead in software switches, and presented the design and implementation of NitroSketch [36], a sketching framework that systematically addresses the performance bottlenecks of sketches without sacrificing robustness and generality.

2.2. Graph Stream Sketch

In contrast to data stream sketches, graph sketches are specially designed for graph stream summarization, keeping the topology of a graph and thus simultaneously supporting several queries such as edge query and node query. Zhao et al. designed gSketch [37], which combined CM-sketch and with a sketch partitioning technique to support edge query and aggregate subgraph query. Tang et al. proposed TCM [1] which adopted several adjacency matrices with irreversible hash functions to store a graph stream. Different from TCM, gMatrix [38] used reversible hash functions to generate graph sketches. Gou et al. proposed GSS [2] which consisted of not only an adjacency matrix but also an adjacency list buffer. Adjacency list buffer was used to store the edge when an edge collision happened to improve the query accuracy. In their follow-up work [6], they proposed an improved version called blocked GSS, and designed two directions of accelerating query: GSS with node bitmaps and GSS implemented with FPGA. In [39], Li et al. proposed Dynamic Graph Sketch which was able to adaptively extend graph sketch size to mitigate the performance degradation caused by memory overload.

In summary, all the existing graph sketches use a single matrix to store both heavy edges and light edges and thus suffer from low query accuracy (especially in light edge query task) due to hash collisions.

3. Preliminaries

Some formal definitions are provided in this section, as well as a preliminary introduction to the basic idea of summarizing a graph stream using graph sketches.

Definition 1 (Graph stream). *A graph stream is a consecutive sequence of items $S = \{e_1, e_2, \dots, e_n\}$, where each item $e_i = (s, d, t, w)$ denotes a directed edge from node s to node d arriving at timestamp t with weight w .*

It is important to note that an edge e_i may appear multiple times at different timestamps. Consequently, the final weight of e_i is calculated using an *aggregation function*. There are various types of aggregation functions, including $\min(\cdot)$, $\max(\cdot)$, $\text{average}(\cdot)$, $\text{sum}(\cdot)$, etc. The $\text{sum}(\cdot)$ function is the most widely accepted [1,2], so we also use it as the aggregation function in the remainder of this paper.

Due to the fact that edges arrive one by one in a graph stream $S = \{e_1, e_2, \dots, e_n\}$, the graph stream may form a dynamic graph which changes as edges arrive (both edge weight and graph architecture may change as edges arrive).

Definition 2 (Session). A session $C = \{e_i, e_{i+1}, \dots, e_j\} (1 \leq i < j \leq n)$ is defined as a continuous subsequent graph stream $S = \{e_1, e_2, \dots, e_n\}$.

Definition 3 (Heavy edge). In a graph stream S , an edge that ranks in the top k percentile out of all the unique edges in a graph stream is considered to be a heavy edge.

Definition 4 (Light edge). Each edge in a graph stream S except the heavy edges is regarded as a light edge.

Definition 5 (Graph sketch [1]). Supposing that a graph $G = (V, E)$ is formed by a given graph stream S , graph sketch \mathcal{K} is defined as a graph $\mathcal{K} = (V_{\mathcal{K}}, E_{\mathcal{K}})$ whose size is smaller than G , i.e., $|V_{\mathcal{K}}| \leq |V|$ and $|E_{\mathcal{K}}| \leq |E|$, where a hash function $H(\cdot)$ is associated with map each node in V to a node in $V_{\mathcal{K}}$. Correspondingly, an edge (s, d) in E will be mapped to the edge $(H(s), H(d))$ in $E_{\mathcal{K}}$.

In order to minimize the query error caused by hash collisions, it is common to simultaneously utilize a number of graph sketches $\{\mathcal{K}_1, \mathcal{K}_2, \dots, \mathcal{K}_m\}$ that use different hash functions $\{H_1(\cdot), H_2(\cdot), \dots, H_m(\cdot)\}$ to summarize a graph stream.

Definition 6 (Graph compression ratio [1]). Compression ratio r in graph summarization means that a graph sketch uses $|E| \times r$ space to store a graph $G = (V, E)$. For example, if a graph stream contains 500,000 edges, compression ratio $1/50$ indicates that the graph sketch takes $500,000 \times 1/50 = 10,000$ space units, which is a $\sqrt{10,000} \times \sqrt{10,000}$ (i.e., 100×100) matrix. In practice, adjacency matrix is usually adopted to implement a graph sketch. We call $\alpha = \sqrt{|E| \times r}$ the width of graph sketch.

In Figure 3, an example illustrates how a set of graph sketches can be used to summarize a graph stream and answer queries. Considering two graph sketches with different hash functions for summarizing a graph stream S , all the values in the two adjacency matrices are initialized to 0 at the beginning. In response to the arrival of an edge, both graph sketches conduct the following *edge update* operation.

Edge update: Each graph sketch \mathcal{K}_i calculates the hash values $(H_i(s), H_i(d))$ for each edge $e = (s, d, t, w)$ in graph stream S . Afterwards, it locates the corresponding position $M_i[H_i(s)][H_i(d)]$ in the adjacency matrix and adds the value there by w .

Graph sketches are capable of answering edge queries and node queries in linear time after processing all the edge updates. The method of answering queries are described as follows.

Edge query: The purpose of edge query is to return the estimated weight of an given edge $e = (s, d)$. The first step in answering the query is to determine the weight of e in each graph sketch. To be more specific, we locate the corresponding position $M_i[H_i(s)][H_i(d)]$ on each graph sketch \mathcal{K}_i , and return its value as an estimated weight. As a result, we can obtain a set of weights $\{w_1, w_2, \dots, w_m\}$. Based on the count-min sketch principle [5], we return $\min\{w_1, w_2, \dots, w_m\}$ for the edge query.

Node query: The purpose of node query is to return the aggregated edge weight for a given node n . The answer is obtained by finding the row corresponding to node n (i.e., the $H_i(n)$ th row) in the adjacency matrix M , and then summing the values in that row. In the same manner, we obtain a set of sums $\{sum_1, sum_2, \dots, sum_m\}$, and return the minimum value for the node query.

Top-k node query: The purpose of top-k node query is to return the list of top-k nodes whose aggregated weights are the highest in graph stream S . In order to respond to this

query, we maintain a min-heap with size k to hold the top- k nodes. Following the update of each edge $e = (s, d)$, we conduct a node query for node s and obtain its aggregated weight w_s . In the next step, we push the tuple (s, w_s) into the min-heap. The lowest-weight tuple will be popped out if the min-heap is full. We return the nodes in the min-heap as the answer to top- k node queries after all edge updates have been completed.

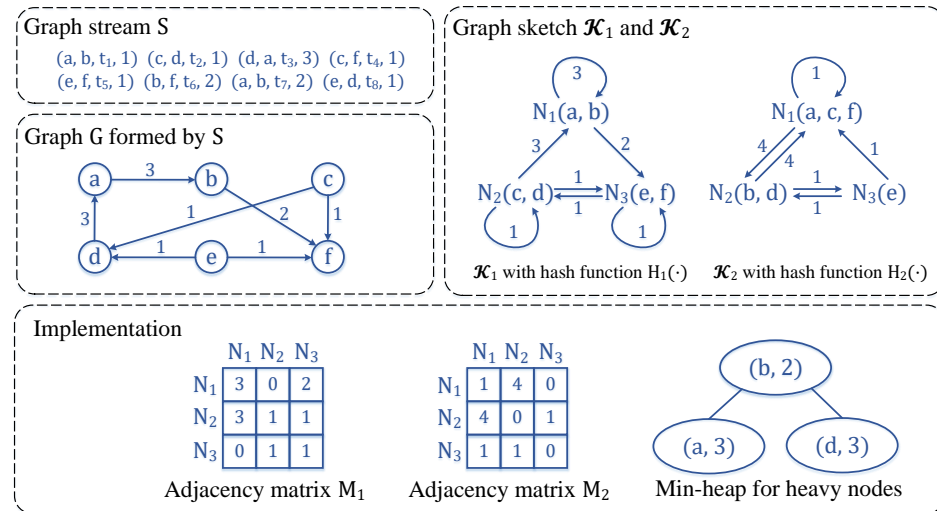


Figure 3. An example of using two graph sketches to summarize a graph stream.

4. Dichotomy Graph Sketch Mechanism

A detailed description of the Dichotomy Graph Sketch (DGS) mechanism is presented in this section. DGS is able to mitigate the performance problem caused by hash collisions between heavy edges and light edges. The proposed framework is illustrated in Figure 4.

To begin with, we represent the edges in the current session by a sub-graph stream $C = \{e_i, e_{i+1}, \dots, e_j\}$. Then, an edge classifier is fed sequentially with these edges. The edge will be stored in the *heavy sketch* if it is classified as a heavy edge; otherwise, it will be stored in the *light sketch*. Due to the lack of training of the edge classifier, all edges will be stored in the light sketch during the first session. To generate training samples for the edge classifier, a *sample generator* is applied. During the current session, it queries the min-heap to find all heavy edges, and then it randomly chooses the same number of light edges. Combined with the chosen light edges, the heavy edges serve as training samples for the edge classifier. Upon completion of each session, the sample generator is reset to null, and DGS proceeds to build the graph sketches incrementally in the next session.

DGS consists of four main components, namely the heavy sketch, the light sketch, the sample generator, and the edge classifier.

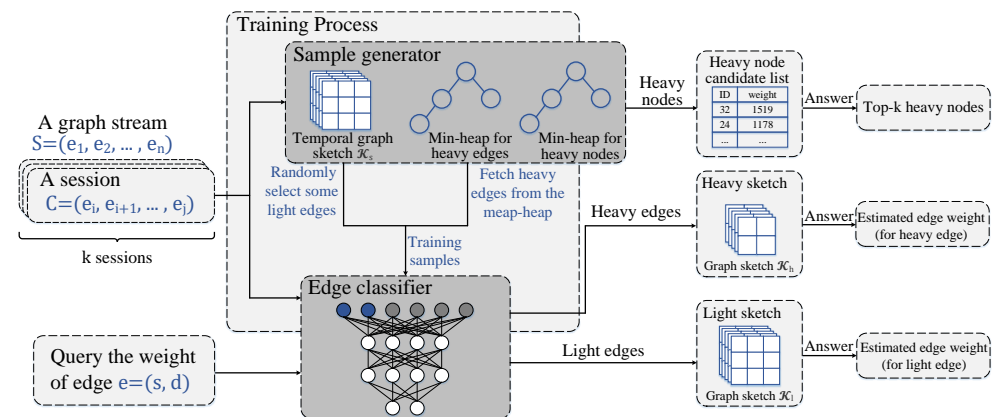


Figure 4. The framework of our proposed Dichotomy Graph Sketch.

4.1. Heavy Sketch and Light Sketch

Both heavy sketches and light sketches function as basic graph sketches (see Definition 5 for details). A graph edge will be fed first into the edge classifier when it arrives. The edge will be stored in the heavy sketch if it is classified as a heavy edge; otherwise, it will be stored in the light sketch. In practice, we set the size of the heavy sketch smaller than the size of the light sketch because the number of heavy edges is usually much smaller than the number of light edges.

4.2. Sample Generator

Using a temporal graph sketch and two min-heaps, we design a sample generator in order to obtain the heavy edges and nodes in each session. The temporal graph is constructed in exactly the same way as a basic graph sketch. One min-heap is used to obtain heavy edges, and the other one is used to obtain heavy nodes. After the sample generator finishes all edge updates of the current session, the heavy edges in the min-heap will be used as training samples to train the edge classifier, and heavy nodes together with their aggregated weights will be recorded in the *heavy node candidate list* (which is a hash table that maps the node ID to its aggregated weight as shown in Figure 4) as the candidates for top-k nodes query. The aggregate weight of a node in the heavy node candidate list will simply be updated by adding the new weight to the old weight. As a final step, sample generator resets all the values of the temporal graph sketch to null, and clears the min-heaps before processing the edges in the next session.

4.3. Edge Classifier

A binary probabilistic classification model is used during the edge updating process in order to distinguish heavy edges from light edges. Essentially, we try to learn a model f that predicts whether an edge $e_i = (n_a, n_b)$ is heavy or not. In other words, we can train a deep neural network classifier based on dataset $\mathcal{D} = \{(e_i = (n_a, n_b), y_i = 1) | e_i \in \mathcal{H}\} \cup \{(e_i = (n_a, n_b), y_i = 0) | e_i \in \mathcal{L}\}$ where \mathcal{H} denotes the set of heavy edges, and \mathcal{L} denotes the set of light edges (from the sample generator). To train the DNN, the node embeddings are also included in the input feature vector based on the discussion in Section 4.3.1.

4.3.1. Node Embeddings Using a Graph Auto-Encoder

Graph representation learning [40,41] typically involves obtaining node embeddings which are essentially feature representations, in order to help accomplish downstream tasks. Thus, using a graph auto-encoder (GAE) [40] model, we obtain node embeddings to assist the classifier in classifying heavy edges and light edges accurately.

Formally, given a graph $G = (V, E)$ with $|V| = N$, we denote its degree matrix as \mathbf{D} , and its adjacency matrix as \mathbf{A} . In addition, node features are summarized in an $N \times M$ feature matrix \mathbf{X} .

GAE utilizes a two-layer graph convolutional network (GCN) [42] as encoder to form the node embeddings of a graph. Formally, the node embeddings are denoted by $\mathbf{Z} = \text{GCN}(\mathbf{X}, \mathbf{A})$ where $\text{GCN}(\cdot)$ denotes the two-layer graph convolutional network. The two-layer GCN is defined as

$$\text{GCN}(\mathbf{X}, \mathbf{A}) = \tilde{\mathbf{A}} \text{ReLU}(\tilde{\mathbf{A}} \mathbf{X} \mathbf{W}_0) \mathbf{W}_1, \quad (1)$$

where $\text{ReLU}(\cdot) = \max(0, \cdot)$, and $\tilde{\mathbf{A}} = \mathbf{D}^{-\frac{1}{2}} \mathbf{A} \mathbf{D}^{-\frac{1}{2}}$ is the symmetrically normalized adjacency matrix. To reconstruct the graph, GAE adopts $\hat{\mathbf{A}} = \sigma(\mathbf{Z} \mathbf{Z}^T)$ as the decoder where $\hat{\mathbf{A}}$ denotes the adjacency matrix of the reconstructed graph G' .

To train a GAE for a graph G , we can feed the G 's adjacency matrix \mathbf{A} together with its node feature matrix \mathbf{X} into the GAE, and obtain the adjacency matrix $\hat{\mathbf{A}}$ of the reconstructed graph G' . Then, we minimize the following cross-entropy loss:

$$\mathcal{L} = \frac{1}{N} \sum y \log \hat{y} + (1 - y) \log(1 - \hat{y}), \quad (2)$$

where y denotes the element in \mathbf{A} , and \hat{y} denotes the element in $\hat{\mathbf{A}}$. After the GAE is well trained, we can obtain the current node embeddings $\mathbf{Z} = \text{GCN}(\mathbf{X}, \mathbf{A})$.

Note that constructing a lossless graph G for every graph stream session is both time-consuming and space-expensive. Since a graph sketch can be regarded as the compression of an original graph, we use the temporal graph sketch \mathcal{K}_s in the sample generator as input to train the GAE to reduce the complexity.

4.3.2. Architecture of the Edge Classifier

In Figure 5, we present the structure of the proposed deep neural network classifier. The input consists of two nodes (source and destination of an edge) as well as their corresponding node embeddings as we mentioned previously. The input first, respectively, goes through two fully connected layers. Afterward, the output of the fully connected layer is sent to a softmax layer, which generates a probability distribution representing the probability that the input edge is heavy.

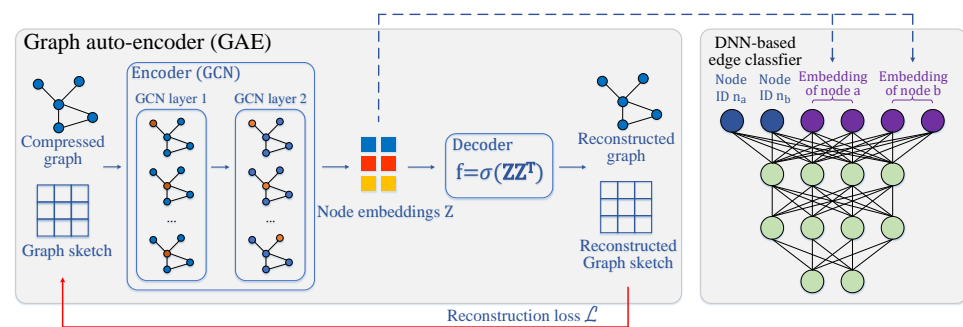


Figure 5. The details of the proposed edge classifier.

The details of the edge classifier is illustrated in Figure 5.

4.4. Implementation of Graph Query Tasks

We discuss how DGS answers different types of queries in this section.

4.4.1. Dealing with Edge Query and Node Query

Given an edge $e_i = (s, d)$, we first feed node s , node d , and their node embeddings into the trained edge classifier to predict whether e_i is a heavy edge. If e_i is classified as a heavy edge, the heavy sketch will answer the weight of e_i (the method to answer the weight is exactly the same as that introduced in Section 3); otherwise the light sketch will answer the weight of e_i . As for node query for node n , we first locate the row corresponding to node n in the graph sketch, sum up the values in that row, and then return the minimum value among all the sums.

4.4.2. Dealing with Top-k Node Query

Having stored each session's heavy nodes in the heavy node candidate list, we can simply return the top-k heavy nodes with the largest weight in the heavy node candidate list as the answer.

4.5. Time and Space Complexity Analysis

In this section, we analyze the space cost and time complexity of DGS. The memory cost of DGS is $O(\alpha^2)$, where α is the width of the light sketch. Specifically, the memory cost of DGS includes the light sketch, the heavy sketch, the heavy node candidate list and the edge classifier. Since the sample generator is temporal and its memory can be freed when all edges have been processed, we do not calculate its memory cost here. Supposing that α_h denotes the width of heavy sketch, the memory cost of heavy sketch is $O(\alpha_h^2)$. Since $\alpha_h < \alpha$ (see Section 4.1 for detail), $O(\alpha_h^2)$ is below $O(\alpha^2)$. The memory cost of heavy node candidate list is a constant c_1 and in practice c_1 is small ($c_1 < 200$). Thus, it is below $O(\alpha^2)$. The

memory cost of edge classifier is a constant c_2 since the size of DNN is fixed. In addition, we set $c_2 < \alpha^2$ in practice, and thus c_2 is also below $O(\alpha^2)$.

The edge update of DGS includes three steps: inferring with the edge classifier, locating the right position in the light sketch/heavy sketch, and updating the value in that position. Supposing that the number of neurons in the i th fully connected layer is β_i , and the input feature vector contains β_{input} elements, the time cost of DNN inference is $O(\beta_{input}\beta_1 + \beta_1\beta_2 + \beta_2\beta_3)$. Locating the right position in the sketch is $O(1)$, since we only need to calculate two hash values. Obviously, updating the value is also $O(1)$. Thus, the total time cost of edge update is $O(\beta_{input}\beta_1 + \beta_1\beta_2 + \beta_2\beta_3)$. In practice, since β_1 , β_2 , and β_3 is fixed in the DNN inference process, the time cost can be regarded as $O(\beta_{input})$. The process of edge query is similar to that of edge update, and thus the time cost of edge query is $O(\beta_{input})$ as well. For top-k heavy nodes query, since we can directly obtain top-k nodes from the heavy node candidate list, the time cost is $O(k)$.

5. Performance Evaluation

A comprehensive set of experiments was conducted on three real-world graph stream datasets in order to validate the effectiveness of the Dichotomy Graph Sketch (DGS). Our method was compared to two state-of-the-art graph sketches: TCM [1] and GSS [2]. A laptop equipped with Intel Core i5-9300H processors (4 cores, 8 threads), 8GB of RAM and NVIDIA GeForce RTX 2060 GPU was used for all experiments. Except for GSS, all sketches were implemented in Python. For GSS, we used the C++ source code provided on the Github (<https://github.com/Puppy95/Graph-Stream-Sketch>). Because GSS does not limit the memory usage of its adjacency list buffer, we disabled it for fair comparison. Our proposed edge classifier was implemented using the PyTorch library [43].

5.1. Datasets

We use three real-world graph stream datasets, which are described as follows.

- **wiki_talk_cy:** The first dataset consists of the Welsh Wikipedia's communication network (http://konect.cc/networks/wiki_talk_cy/). Users are represented by nodes, and an edge from user A to user B indicates that user A posted a message on the talk page of user B at a specific time. It contains 2233 users (nodes) and 10,740 messages (edges).
- **subelj_jung:** The second dataset is the software class dependency network of the JUNG 2.0.1 and javax 1.6.0.7 libraries, namespaces edu.uci.ics.jung and java/javax (http://konect.cc/networks/subelj_jung-j/). A node represents a class, and an edge between two nodes indicates that these classes are dependent on one another. It contains 6210 classes (nodes) and 138,706 dependencies (edges).
- **facebook-wosn-wall:** The third dataset is the directed network of a small subset of posts to other user's wall on Facebook (<http://konect.cc/networks/facebook-wosn-wall/>). A node in the network represents a Facebook user, and a directed edge represents a post, connecting the user writing a post with the user whose wall the post appears on. It contains 46,952 users (nodes) and 876,993 posts (edges).
- **slashdot:** The fourth dataset is the reply network of technology website Slashdot (<http://konect.cc/networks/slashdot-threads/>). Nodes are users and edges are replies. The edges are directed and start from the responding user. Edges are annotated with the timestamp of the reply. It contains 51,083 users (nodes) and 140,778 replies (edges).

5.2. Performance Metrics

We adopt the following metrics for performance evaluation in our experiments.

5.2.1. Average Relative Error (ARE)

Average relative error measures the accuracy of the weights that are estimated by a graph sketch in the edge query task. Given a query q , the *relative error* $RE(q)$ is formally defined as follows:

$$RE(q) = \frac{|\hat{f}(q) - f(q)|}{f(q)} \quad (3)$$

where $\hat{f}(q)$ denotes the estimated answer of query q , and $f(q)$ denotes the ground truth answer of query q .

Given a set of queries $\mathcal{Q} = \{q_1, q_2, \dots, q_n\}$, the *average relative error (ARE)* is calculated by averaging the relative errors over all the queries in \mathcal{Q} , and it is formally defined as follows:

$$ARE(\mathcal{Q}) = \frac{\sum_{i=1}^n RE(q_i)}{n} \quad (4)$$

5.2.2. Intersection Accuracy (IA)

Intersection accuracy [1] measures the accuracy of the top-k heavy nodes reported by a graph sketch. Supposing that X denotes the set of reported top-k heavy nodes, and Y denotes the set of ground truth top-k heavy nodes. The *intersection accuracy (IA)* is formulated as follows:

$$IA = \frac{|X \cap Y|}{k} \quad (5)$$

It is obvious that $IA \in [0, 1]$, and a larger IA means more top-k heavy nodes are found.

5.2.3. Normalized Discounted Cumulative Gain (NDCG)

Normalized discounted cumulative gain [44] is a measure of ranking quality. Given a ranking list of heavy nodes reported by a graph sketch, *discounted cumulative gain (DCG@k)* measures the usefulness (also called gain) of a node based on its position in the ranking list, and it is formally defined as follows:

$$DCG@k = \sum_{i=1}^k \frac{rel_i}{\log_2(i+1)} \quad (6)$$

where rel_i is the relevance value of the result at position i , and $rel_i \in \{0, 1\}$. If the i th node in the ranking list is indeed a heavy node, then rel_i will be 1; otherwise, it will be 0. Using the definition above, $NDCG@k$ is formulated as follows:

$$NDCG@k = \frac{DCG@k}{IDCG@k} \quad (7)$$

where $IDCG@k$ represents the $DCG@k$ of an *ideal ranking list*. The ideal ranking list can be obtained by sorting the nodes in the ranking list in descending order with respect to their relative scores. It is obvious that $NDCG \in [0, 1]$, and the higher NDCG indicates a stronger ability to find top-k heavy nodes.

5.3. Numerical Results

A numerical analysis of edge query and node query results is conducted for various sketches. In order to make a fair comparison, TCM, GSS and our proposed DGS all use the same amount of memory for both edge query tasks and node query tasks. Note that the DGS framework refers to two hyperparameters: the compression ratio and the size of the edge classifier (i.e., the number of neurons in each fully connected layer). In our experiments, we set these hyperparameters as shown in Table 1. Furthermore, we examine the effect of hyperparameters in Section 5.3.4.

Table 1. The setting of hyperparameters.

Number of Neurons in fully connected Layer	Compression Ratio
32	1/10

5.3.1. Edge Query

To evaluate the ability to answer edge query on the baseline methods and our proposed DGS, for each dataset, we query all the edges and calculating the average relative error (ARE). Tables 2–5 show the ARE of edge query task achieved by TCM, GSS and our proposed DGS. Besides calculating the total ARE by querying all edges, we also separately calculate the ARE of querying heavy edges and that of querying light edges. As can be seen, DGS achieves the lowest ARE among all three methods. Specifically, in TCM and GSS, the ARE of edge queries on dataset *subelj_jung* is 22.359 and 39.386, respectively. In contrast, our proposed DGS outperforms the other algorithms significantly, and its ARE is only 17.264. Moreover, both the ARE of querying light edges (17.295) and that of query heavy edges (1.433) achieved by DGS are the lowest compared with the baseline methods. Similarly, DGS also achieves the lowest ARE on the other three datasets. It verifies the effectiveness of DGS for edge query.

Table 2. The ARE of edge query (*wiki_talk_cy*).

Method	Total ARE	ARE of Light Edges	ARE of Heavy Edges
TCM	10.388	10.436	0.274
GSS	10.038	10.083	0.622
DGS	7.875	7.909	0.202

Table 3. The ARE of edge query (*subelj_jung*).

Method	Total ARE	ARE of Light Edges	ARE of Heavy Edges
TCM	22.359	22.399	2.079
GSS	39.386	39.458	3.320
DGS	17.264	17.295	1.433

Table 4. The ARE of edge query (*facebook-wosn-wall*).

Method	Total ARE	ARE of Light Edges	ARE of Heavy Edges
TCM	2.261	2.262	0.015
GSS	5.256	5.258	0.340
DGS	2.056	2.057	0.212

Table 5. The ARE of edge query (*slashdot*).

Method	Total ARE	ARE of Light Edges	ARE of Heavy Edges
TCM	5.345	5.349	0.923
GSS	8.909	8.915	1.499
DGS	5.188	5.192	0.643

5.3.2. Top-k Heavy Node Query

We evaluate the ability to find top-k heavy nodes of DGS as well as TCM. We do not conduct this experiment with GSS since GSS does not support heavy node query. The results are shown in Figures 6 and 7. As shown in Figure 6, DGS outperforms TCM on all four datasets. Specifically, in the task of finding top-20 heavy nodes, DGS achieves an IA of 95%, 85%, 90% and 90% on datasets *wiki_talk_cy*, *subelj_jung*, *facebook-wosn-wall* and *slashdot*, respectively. In contrast, TCM only achieves an IA of 80%, 25%, 45% and 45% for each of these, respectively. In the task of finding top-50 and top-100 heavy nodes, DGS also

outperforms TCM significantly. This illustrates that DGS has strong ability to find heavy nodes accurately.

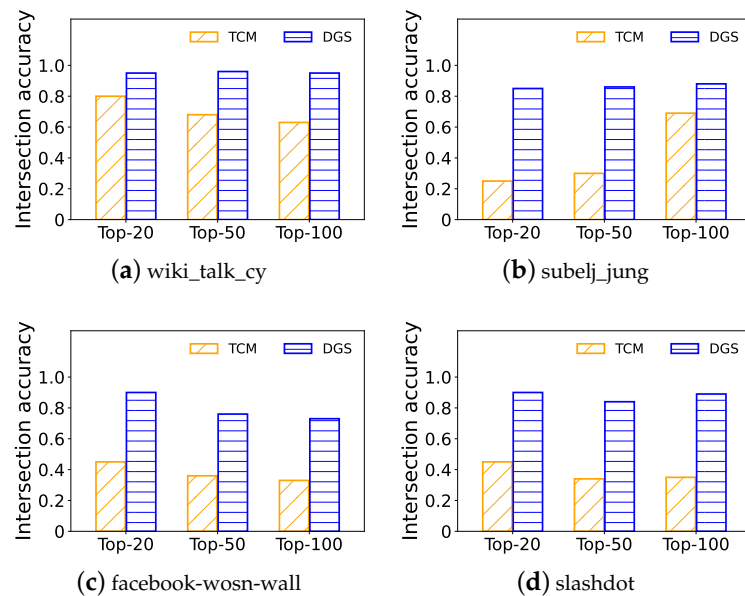


Figure 6. Heavy node query (intersection accuracy).

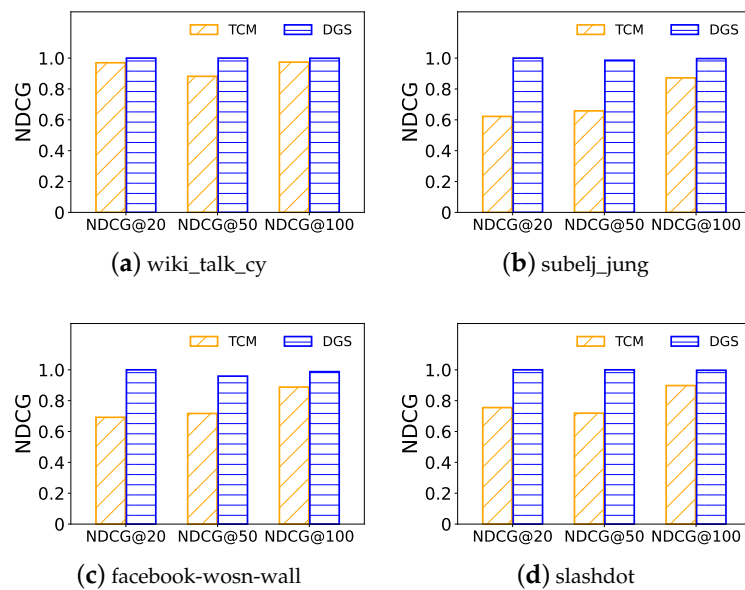


Figure 7. Heavy node query (NDCG).

We also calculate the NDCG based on the result list of top-k heavy node query. The results are shown in Figure 7. Similarly, the NDCG achieved by DGS is much higher than that of TCM.

5.3.3. Memory Usage Comparison on Different Datasets

In this section, we make a memory usage comparison with respect to the datasets used in the experiments. As shown in Table 1, we set the default compression ratio to 1/10. Taking the dataset *wiki_talk_cy* as example, the space units used by heavy sketch and light sketch is $|E| \times r = 10,740 \times 1/10 = 1074$. Supposing that the size of an integer is 4 Bytes, the memory usage of dataset *wiki_talk_cy* is 4 Byte \times 1074 = 4296 Byte \approx 4.19 KB. In the same manner, the memory that are needed on dataset *subelj_jung*, *facebook-wosn-wall*,

slashdot can also be calculated, and we present the memory usage comparison respect to the four datasets in Table 6.

Table 6. Memory usage comparison with respect to the datasets.

Dataset	Wiki_talk_cy	Subelj_jung	Facebook-Wosn-Wall	Slashdot
Memory usage	4.19 KB	54.18 KB	342.57 KB	54.99 KB

5.3.4. Hyperparameter Analysis

The DGS framework refers to two hyperparameters: the size of the edge classifier, and compression ratio. We study the influence of these hyperparameters on edge query task using the dataset *subelj_jung* and *slashdot*, and the results are shown in Figures 8 and 9.

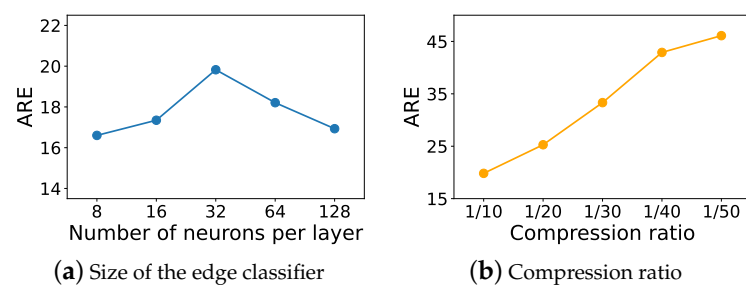


Figure 8. Hyperparameter analysis on edge query task using dataset *subelj_jung*.

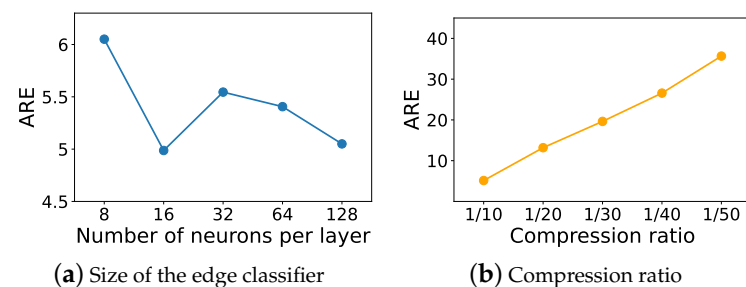


Figure 9. Hyperparameter analysis on edge query task using dataset *slashdot*.

Figures 8a and 9a show the change of ARE when tuning the number of neurons in each fully connected layer of the edge classifier from 8 to 128. The results show that a small-size DNN classifier is sufficient to differentiate heavy edges from light edges, helping DGS achieving a low ARE on edge query task. Enlarging the size of DNN does not significantly lower the ARE. Thus, we set the number of neurons to 16.

Figures 8b and 9b show the change of ARE when tuning the compression ratio from 1/10 to 1/50. The results show that the ARE is very sensitive to compression ratio. With the increase in compression ratio, the ARE increases drastically. Thus, to avoid an unacceptable ARE, we should use a relatively low compression ratio.

6. Conclusions

The DGS framework is proposed in this paper as a novel method for summarizing large graph streams. To avoid the serious performance drops caused by collisions between hashes, DGS uses two separate matrices, the heavy sketch and the light sketch, to store heavy edges and light edges, respectively. In order to determine whether an edge is heavy or not, DGS first obtains node embeddings using a GAE, and adopts a DNN-based edge classifier which utilizes the node embeddings as features to make the classification. Edges classified as heavy edges will be stored in the heavy sketch, while edges classified as light edges will be stored in the light sketch. Furthermore, DGS sets a sample generator to

periodically generate training samples for the edge classifier, and record the heavy edges of each session in the heavy node candidate list for fast top-k nodes query. In extensive experiments using four real-world graph streams, the proposed method was able to achieve high accuracy for graph queries including edge query and top-k nodes query compared to the state of the art. The ARE of edge query achieved by DGS was reduced by up to 24.19% compared to TCM, and was reduced by up to 60.88% compared to GSS. Meanwhile, the IA of node query achieved by DGS is significantly higher than that achieved by TCM. Finally, we made a hyperparameter analysis and found the best hyperparameters for the DGS framework.

Author Contributions: Conceptualization, D.L. and W.L.; methodology, D.L. and W.L.; software, D.L., Y.C. and X.Z.; validation, D.L. and M.L.; formal analysis, D.L.; investigation, D.L. and W.L.; resources, W.L. and S.L.; data curation, D.L. and G.Z.; writing—original draft preparation, D.L.; writing—review and editing, D.L. and W.L.; visualization, D.L., G.Z. and M.L.; supervision, W.L.; project administration, W.L.; funding acquisition, W.L. and S.L. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the National Natural Science Foundation of China Grant Numbers 62262018, 61972196.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The datasets used in this article can be found at http://konect.cc/networks/wiki_talk_cy/, http://konect.cc/networks/subelj_jung-j/, <http://konect.cc/networks/fac ebook-wosn-wall/>, and <http://konect.cc/networks/slashdot-threads/> (accessed on 10 December 2023).

Acknowledgments: This work was partially supported by the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Sino-German Institutes of Social Computing.

Conflicts of Interest: The authors declare no conflicts of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript; or in the decision to publish the results.

References

1. Tang, N.; Chen, Q.; Mitra, P. Graph Stream Summarization: From Big Bang to Big Crunch. In Proceedings of the 2016 International Conference on Management of Data (SIGMOD'16), San Francisco, CA, USA, 26 June–1 July 2016; pp. 1481–1496.
2. Gou, X.; Zou, L.; Zhao, C.; Yang, T. Fast and Accurate Graph Stream Summarization. In Proceedings of the 35th IEEE International Conference on Data Engineering (ICDE'19), Macao, China, 8–11 April 2019; pp. 1118–1129.
3. Merchant, A.; Mathioudakis, M.; Wang, Y. Graph Summarization via Node Grouping: A Spectral Algorithm. In Proceedings of the WSDM'23: Proceedings of the Sixteenth ACM International Conference on Web Search and Data Mining, Singapore, 27 February–3 March 2023; pp. 742–750.
4. Hajiabadi, M.; Singh, J.; Srinivasan, V.; Thomo, A. Graph Summarization with Controlled Utility Loss. In Proceedings of the KDD'21: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual, Singapore, 14–18 August 2021; pp. 536–546.
5. Cormode, G.; Muthukrishnan, S. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms* **2005**, *55*, 58–75.
6. Gou, X.; Zou, L.; Zhao, C.; Yang, T. Graph Stream Sketch: Summarizing Graph Streams with High Speed and Accuracy. *IEEE Trans. Knowl. Data Eng.* **2022**, *35*, 5901–5914.
7. Li, D.; Li, W.; Chen, Y.; Zhong, X.; Lin, M.; Lu, S. Learning-Based Dichotomy Graph Sketch for Summarizing Graph Streams with High Accuracy. In Proceedings of the KSEM'23: The 16th International Conference on Knowledge Science, Engineering and Management, Guangzhou, China, 16–18 August 2023; pp. 47–59.
8. Charikar, M.; Chen, K.C.; Farach-Colton, M. Finding Frequent Items in Data Streams. In Proceedings of the 29th International Colloquium on Automata, Languages and Programming, Málaga, Spain, 8–13 July 2002; pp. 693–703.
9. Eitan, C.; Varghese, G. New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice. *ACM Trans. Comput. Syst.* **2003**, *21*, 270–313.
10. Giroire, F.; Chandrashekar, J.; Taft, N.; Schooler, E.M.; Papagiannaki, D. Exploiting Temporal Persistence to Detect Covert Botnet Channels. In Proceedings of the Recent Advances in Intrusion Detection, 12th International Symposium, RAID 2009, Saint-Malo, France, 23–25 September 2009; Volume 5758, pp. 326–345.

11. Li, J.; Li, Z.; Xu, Y.; Jiang, S.; Yang, T.; Cui, B.; Dai, Y.; Zhang, G. WavingSketch: An Unbiased and Generic Sketch for Finding Top-k Items in Data Streams. In Proceedings of the KDD'20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event, 6–10 July 2020; pp. 1574–1584.
12. Zhang, Y.; Li, J.; Lei, Y.; Yang, T.; Li, Z.; Zhang, G.; Cui, B. On-Off Sketch: A Fast and Accurate Sketch on Persistence. *Proc. VLDB Endow.* **2020**, *14*, 128–140.
13. Zhong, Z.; Yan, S.; Li, Z.; Tan, D.; Yang, T.; Cui, B. BurstSketch: Finding Bursts in Data Streams. In Proceedings of the SIGMOD'21: International Conference on Management of Data, Virtual, 20–25 June 2021; pp. 2375–2383.
14. Liu, Z.; Kong, C.; Yang, K.; Yang, T.; Miao, R. HyperCalm Sketch: One-Pass Mining Periodic Batches in Data Streams. In Proceedings of the ICDE'23: 39th IEEE International Conference on Data Engineering, Anaheim, CA, USA, 3–7 April 2023.
15. Zhao, Y.; Han, W.; Zhong, Z.; Zhang, Y.; Yang, T.; Cui, B. Double-Anonymous Sketch: Achieving Fairness for Finding Global Top-K Frequent Items. In Proceedings of the SIGMOD'23: International Conference on Management of Data, Seattle, WA, USA, 18–23 June 2023.
16. Ben-Basat, R.; Einziger, G.; Friedman, R.; Kassner, Y. Randomized admission policy for efficient top-k and frequency estimation. In Proceedings of the INFOCOM'17: IEEE Conference on Computer Communications, Atlanta, GA, USA, 1–4 May 2017; pp. 1–9.
17. Fan, D.; Rafiei, D. New estimation algorithms for streaming data: Count-min can do more. 2014. Available online: https://www.researchgate.net/publication/228531010_New_estimation_algorithms_for_streaming_data_Count-min_can_do_more (accessed on 10 December 2023).
18. Wang, F.; Chen, Q.; Li, Y.; Yang, T.; Tu, Y.; Yu, L.; Cui, B. JoinSketch: A Sketch Algorithm for Accurate and Unbiased Inner-Product Estimation. In Proceedings of the SIGMOD'23: International Conference on Management of Data, Washington, DC, USA, 18–23 June 2023.
19. Li, X.; Fan, Z.; Li, H.; Zhong, Z.; Guo, J.; Long, S. SteadySketch: Finding Steady Flows in Data Streams. In Proceedings of the IWQoS'23: IEEE/ACM International Symposium on Quality of Service, Anaheim, CA, USA, 3–7 April 2023.
20. Fan, Z.; Hu, Z.; Wu, Y.; Guo, J.; Liu, W.; Yang, T.; Wang, H.; Xu, Y.; Uhlig, S.; Tu, Y. PISketch: finding persistent and infrequent flows. In Proceedings of the FFSPIN '22: The ACM SIGCOMM Workshop on Formal Foundations and Security of Programmable Network Infrastructures, Amsterdam, The Netherlands, 11–25 May 2022; pp. 8–14.
21. Fan, Z.; Hu, Z.; Wu, Y.; Guo, J.; Wang, S.; Liu, W.; Yang, T.; Tu, Y.; Uhlig, S. PISketch: Finding Persistent and Infrequent Flows. *IEEE/ACM Trans. Netw.* **2023**. <https://doi.org/10.1109/TNET.2023.3272287>.
22. Zhao, Y.; Zhang, Y.; Yi, P.; Yang, T.; Cui, B.; Uhlig, S. The Stair Sketch: Bringing more Clarity to Memorize Recent Events. In Proceedings of the ICDE'22: 38th IEEE International Conference on Data Engineering, Kuala Lumpur, Malaysia, 9–12 May 2022; pp. 164–177.
23. Gou, X.; Zhang, Y.; Hu, Z.; He, L.; Wang, K.; Liu, X.; Yang, T.; Wang, Y.; Cui, B. A Sketch Framework for Approximate Data Stream Processing in Sliding Windows. *IEEE Trans. Knowl. Data Eng.* **2022**, *35*, 4411–4424.
24. Fan, Z.; Zhang, Y.; Dong, S.; Zhou, Y.; Liu, F.; Yang, T.; Uhlig, S.; Cui, B. HoppingSketch: More Accurate Temporal Membership Query and Frequency Query. *IEEE Trans. Knowl. Data Eng.* **2022**, *35*, 9067–9072.
25. Peng, Y.; Guo, J.; Li, F.; Qian, W.; Zhou, A. Persistent Bloom Filter: Membership Testing for the Entire History. In Proceedings of the SIGMOD'18: Proceedings of the 2018 International Conference on Management of Data, Houston, TX, USA, 10–15 June 2018; pp. 1037–1052.
26. Liu, L.; Shen, Y.; Yan, Y.; Yang, T.; Shahzad, M.; Cui, B.; Xie, G. SF-Sketch: A Two-Stage Sketch for Data Streams. *IEEE Trans. Parallel Distrib. Syst.* **2020**, *31*, 2263–2276.
27. Miao, R.; Dong, F.; Zhao, Y.; Zhao, Y.; Wu, Y.; Yang, K.; Yang, T.; Cui, B. SketchConf: A Framework for Automatic Sketch Configuration. In Proceedings of the ICDE'23: 39th IEEE International Conference on Data Engineering, Anaheim, CA, USA, 3–7 April 2023.
28. Li, H.; Chen, Q.; Zhang, Y.; Yang, T.; Cui, B. Stingy Sketch: A Sketch Framework for Accurate and Fast Frequency Estimation. *Proc. VLDB Endow.* **2022**, *15*, 1426–1438.
29. Yang, T.; Xu, J.; Liu, X.; Liu, P.; Wang, L.; Bi, J.; Li, X. A generic technique for sketches to adapt to different counting ranges. In Proceedings of the INFOCOM'19: IEEE Conference on Computer Communications, Paris, France, 29 April–2 May 2019; pp. 2017–2025.
30. Liu, X.; Xu, Y.; Liu, P.; Yang, T.; Xu, J.; Wang, L.; Xie, G.; Li, X.; Uhlig, S. SEAD counter: Self-adaptive counters with different counting ranges. *IEEE/ACM Trans. Netw.* **2021**, *30*, 90–106.
31. Liu, Z.; Manousis, A.; Vorsanger, G.; Sekar, V.; Braverman, V. One sketch to rule them all: Rethinking network flow monitoring with univmon. In Proceedings of the SIGCOMM'16: The Annual Conference of the ACM Special Interest Group on Data Communication, Florianopolis, Brazil, 22–26 August, 2016; pp. 101–114.
32. Huang, Q.; Jin, X.; Lee, P.P.; Li, R.; Tang, L.; Chen, Y.C.; Zhang, G. Sketchvisor: Robust network measurement for software packet processing. In Proceedings of the SIGCOMM'17: The Annual Conference of the ACM Special Interest Group on Data Communication, Los Angeles, CA, USA, 21–25 August 2017; pp. 113–126.
33. Tang, L.; Huang, Q.; Lee, P.P.C. MV-Sketch: A Fast and Compact Invertible Sketch for Heavy Flow Detection in Network Data Streams. In Proceedings of the IEEE Conference on Computer Communications (INFOCOM'19), Paris, France, 29 April–2 May 2019; pp. 2026–2034.

34. Yang, T.; Jiang, J.; Liu, P.; Huang, Q.; Gong, J.; Zhou, Y.; Miao, R.; Li, X.; Uhlig, S. Elastic sketch: Adaptive and fast network-wide measurements. In Proceedings of the SIGCOMM'18: The Annual Conference of the ACM Special Interest Group on Data Communication, Budapest, Hungary, 20–25 August, 2018; pp. 561–575.
35. Yang, T.; Jiang, J.; Liu, P.; Huang, Q.; Gong, J.; Zhou, Y.; Miao, R.; Li, X.; Uhlig, S. Adaptive measurements using one elastic sketch. *IEEE/ACM Trans. Netw.* **2019**, *27*, 2236–2251.
36. Liu, Z.; Ben-Basat, R.; Einziger, G.; Kassner, Y.; Braverman, V.; Friedman, R.; Sekar, V. Nitrosketch: Robust and general sketch-based monitoring in software switches. In Proceedings of the SIGCOMM'19: The annual ACM Special Interest Group on Data Communication, Beijing, China, 19–23 August 2019; pp. 334–350.
37. Zhao, P.; Aggarwal, C.C.; Wang, M. gSketch: On Query Estimation in Graph Streams. *Proc. VLDB Endow.* **2011**, *5*, 193–204.
38. Khan, A.; Aggarwal, C.C. Query-friendly compression of graph streams. In Proceedings of the 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM'16), Davis, CA, USA, 18–21 August 2016; Kumar, R., Caverlee, J., Tong, H., Eds.; pp. 130–137.
39. Li, D.; Li, W.; Chen, Y.; Lin, M.; Lu, S. Learning-Based Dynamic Graph Stream Sketch. In Proceedings of the PAKDD'21: Advances in Knowledge Discovery and Data Mining—25th Pacific-Asia Conference, Virtual, 11–14 May 2021; Volume 12712, pp. 383–394.
40. Kipf, T.N.; Welling, M. Variational Graph Auto-Encoders. *arXiv* **2016**, arXiv:1611.07308.
41. Hamilton, W.L.; Ying, Z.; Leskovec, J. Inductive Representation Learning on Large Graphs. In Proceedings of the NeurIPS'17: Advances in Neural Information Processing Systems 30, Long Beach, CA, USA, 4–9 December 2017; pp. 1024–1034.
42. Kipf, T.N.; Welling, M. Semi-Supervised Classification with Graph Convolutional Networks. In Proceedings of the ICLR'17: 5th International Conference on Learning Representations, Toulon, France, 24–26 April 2017.
43. Paszke, A.; Gross, S.; Massa, F.; Lerer, A.; Bradbury, J.; Chanan, G.; Killeen, T.; Lin, Z.; Gimelshein, N.; Antiga, L.; et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Proceedings of the Annual Conference on Neural Information Processing Systems (NeurIPS'19), Vancouver, BC, Canada, 8–14 December, 2019; pp. 8024–8035.
44. Järvelin, K.; Kekäläinen, J. Cumulated gain-based evaluation of IR techniques. *ACM Trans. Inf. Syst.* **2002**, *20*, 422–446.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.