# Learning-Based Dichotomy Graph Sketch for Summarizing Graph Streams with High Accuracy

Ding Li, Wenzhong Li(✉), Yizhou Chen, Xu Zhong, Mingkai Lin, and Sanglu Lu

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
{liding,mf20330010,xuzhong,mingkai}@smail.nju.edu.cn, {lwz,sanglu}@nju.edu.cn

**Abstract.** Graph stream data is widely applied to describe the relationships in networks such as social networks, computer networks and hyperlink networks. Due to the large volume and high dynamicity of graph streams, several graph sketches were proposed to summarize them for fast queries. However, the existing graph sketches suffer from low performance on graph query tasks due to hash collisions between heavy and light edges. In this paper, we propose a novel learning-based Dichotomy Graph Sketch (DGS) mechanism, which adopts two separate graph sketches, a *heavy sketch* and a *light sketch*, to store heavy edges and light edges respectively. DGS periodically obtains heavy edges and light edges in a session of a graph stream, and use them as training samples to train a deep neural network (DNN) based binary classifier. The DNN-based classifier is then utilized to decide whether the upcoming edges are heavy or not, and store them in different graph sketches accordingly. With the learnable classifier and the dichotomy graph sketches, the proposed mechanism can resolve the hashing collision problem and significantly improve the accuracy for graph query tasks. We conducted extensive experiments on three real-world graph stream datasets, which show that DGS outperforms the state-of-the-art graph sketches in a variety of graph query tasks.

**Keywords:** Sketch · Graph sketch · Deep learning · Graph stream

## 1 Introduction

In many data stream applications, the connections are indispensable to describe the relationships in networks such as social networks, computer networks and communication networks. In these applications, data are organized as *graph*

This work was partially supported by the Natural Science Foundation of Jiangsu Province (Grant No. BK20222003), the National Natural Science Foundation of China (Grant Nos. 61972196, 61832008, 61832005), the Collaborative Innovation Center of Novel Software Technology and Industrialization, and the Sino-German Institutes of Social Computing.


Z. Jin et al. (Eds.): KSEM 2023, LNAI 14118, pp. 47–59, 2023.
https://doi.org/10.1007/978-3-031-40286-9_5

*streams* [15] [4] which are different from traditional data streams that are modeled as isolated items. A graph stream can form a dynamic graph that changes with every arrival of an edge. For example, network traffic can be seen as a graph stream, where each item in the stream represents the communication between two IP addresses. The network traffic graph will change if a new packet arrives. For another example, interactions between users in a social network can also be seen as a graph stream, and these interactions forms a dynamic graph. The social network graph will dynamically change when new interactions occur. Formally, a graph stream is a consecutive sequence of items, where each item represents a graph edge usually denoted by a tuple consisting of two endpoints and a weight.

With the increase of graph sizes, it is necessary to build an accurate but small representation of the original graph for more efficient analytics. To achieve this goal, researchers studied the *graph summarization* problem to concisely preserve overall graph structure while reducing its size. For example, Hajiabadi et al. designed a utility-driven graph summarization method G-SCIS [6] that produced optimal compression with zero loss of utility. However, graph summarization focused on summarizing static graphs, which was unable to handle dynamic graphs that are formed by graph streams. To summarize graph streams, some researchers proposed *graph sketch*. For example, Tang et al. proposed a novel graph sketch called TCM [15]. TCM summarized a graph stream by a matrix where each edge was mapped to a bucket of the matrix using a hash function, and the edge weight was recorded in the corresponding bucket. TCM supported not only edge query but also node query since it keeped the topology information of the graph. More recently, Gou et al. desinged GSS [4] which combined a matrix and an adjacency list buffer to improve the accuracy of edge query. The adjacency list buffer stored the edge when a hash collision happened in the matrix. In their follow-up work [5], they partitioned the matrix of GSS into multiple blocks, and accelerated the query with bitmaps and FPGA (Field Programmable Gate Array).

Typically, conventional graph sketches use a random hash function to map each edge to a bucket of a matrix, and then record the edge weight in the corresponding bucket, which works as illustrated in Fig. 1. However, due to the randomness of hash function, hash collisions may occur in querying an edge or a node with the graph sketch. Especially when a heavy edge (an edge with large

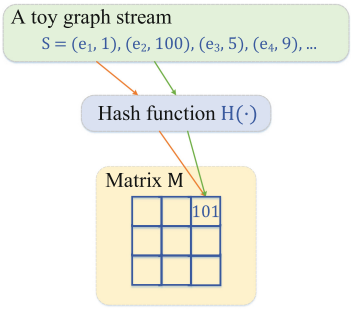



**Fig. 1.** Conventional graph sketch stores all edges in a single matrix.

weight) is collided with a light edge (an edge with small weight), it will cause severe performance degradation on query tasks. As shown in Fig. 1, a toy graph stream consists of a heavy edge (edge $e_2$) whose weight is 100 and several light edges (edge $e_1$, $e_3$, and $e_4$). The conventional graph sketches may map a heavy edge $e_2$ and a light edge $e_1$ 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 Sect. 3 for details), and thus the value recorded in this bucket is $1 + 100 = 101$. In this case, if we query the weight of edge $e_1$, the graph sketch will return the value 101, and the relative error of this query is $(101 - 1)/1 = 100$, which is exceptionally high for a light edge.

To address this issue, it is desirable to design a new graph sketch mechanism to resolve the high query error caused by hash collisions. In this paper, we propose a novel Dichotomy Graph Sketch (DGS) approach, which is able to differentiate heavy edges and light edges during the sketch updating process and respectively store them in two separate matrices to avoid hash collisions. When a new edge arrives, DGS first decides whether it is a heavy edge or not by an *edge classifier*. If the edge is classified as a heavy edge, it will be stored in a *heavy sketch*; otherwise, it will be stored in a *light sketch*. To train the edge classifier, we set a *sample generator* which consists of a temporal graph sketch and two min-heaps to generate training samples. The edges in each session are also fed into the sample generator to obtain the heavy edges (see Sect. 4 for details) to train the edge classifier. In this way, hash collision can be avoided and the query performance will be improved.

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

- We propose dichotomy graph sketch (DGS), a novel mechanism for graph stream summarization. It is able to differentiate heavy edges from light edges during the edge updating process.
- We introduce deep learning techniques into the design of graph sketch mechanism. Specifically, we design a novel deep neural network architecture to detect heavy edges during the edge updating process, which helps to store heavy edges and light edges in separate matrices to avoid hash collision.
- We conduct extensive experiments on three real-world graph streams to evaluate the performance of the proposed DGS. The experimental results show that DGS outperforms the state-of-the-art graph sketches in a variety of graph query tasks.

## 2 Related Work

### 2.1 Data Stream Sketches

Data stream sketches are designed for data stream summarization. C-sketch [1] utilized several hash tables to record data streams, but suffered from both overestimation and underestimation in frequency estimation task. Estan et al. designed CU-sketch [3] which improved query accuracy at the cost of not supporting item deletions. SF-sketch [14] used both a large sketch and a small sketch to upgrade the query accuracy. Li et al. designed WavingSketch [13] which was a generic

and unbiased algorithm for finding top-k items. Zhang et al. proposed On-Off sketch [16] which focused on the problem of persistence estimation and finding persistent items by a technique to separate persistent and non-persistent items. Stingy Sketch [12] was a sketch framework which designed bit-pinching counter tree and prophet queue to optimize both the accuracy and speed for frequency estimation task.

### 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. Tang et al. proposed TCM [15] which adopted several adjacency matrices with irreversible hash functions to store a graph stream. Different from TCM, gMatrix [8] used reversible hash functions to generate graph sketches. Gou et al. proposed GSS [4] 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 [5], 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 [11], 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 for light edge query task) due to hash collisions.

## 3 Preliminaries

In this section, we provide formal definitions and introduce the preliminary of summarizing a graph stream with graph sketches.

**Definition 1 (Graph stream).** *A graph stream is a consecutive sequence of items* $S = \{e_1, e_2, \ldots, e_n\}$*, 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 worth noting that an edge $e_i$ may appear multiple times at different timestamps. Thus, the final weight of $e_i$ is computed by an *aggregation function* based on all edge weights of $e_i$. Common aggregation functions include $min(\cdot)$, $max(\cdot)$, $average(\cdot)$, $sum(\cdot)$, etc. Among them, $sum(\cdot)$ is mostly adpoted [15] [4], and thus we also use $sum(\cdot)$ as the aggregation function in the rest of this paper.

Since the edges in a graph stream arrive one by one, a graph stream $S = \{e_1, e_2, \ldots, e_n\}$ can form a dynamic graph which changes with every arrival of an edge (both edge weight and graph architecture may change).

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

**Definition 3 (Heavy edge).** *A heavy edge refers to the edge whose final weight is ranked in the top k percentage among all the unique edges in a graph stream S.*

**Definition 4 (Light edge).** *Except heavy edges, all the other edges are regarded as light edges in a graph stream S.*

**Definition 5 (Graph sketch [15]).** *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 to 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 $E_{\mathcal{K}}$.*

To reduce the query error caused by hash collisions, a common method is to simultaneously use a set of graph sketches $\{\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_m\}$ with different hash functions $\{H_1(\cdot), H_2(\cdot), \ldots, H_m(\cdot)\}$ to summarize a graph stream.

**Definition 6 (Graph compression ratio[15]).** *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. $100 \times 100$) 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.*

Assuming that two graph sketches with different hash functions are utilized to summarize a graph stream $S$, at the beginning, all values in the two adjacency matrices are initialized with 0. When an edge arrives, both graph sketches conduct an *edge update* operation as follows.

**Edge Update:** To record an edge $e = (s, d, t, w)$ from graph stream S, each graph sketch $\mathcal{K}_i$ calculates the hash values $(H_i(s), H_i(d))$. Then, it locates the corresponding position $M_i[H_i(s)][H_i(d)]$ in the adjacency matrix, and adds the value in that position by $w$.

After the graph sketches process all edge updates in the graph stream, they can be used to fastly answer edge query and node query in linear time, which can be described as follows.

**Edge Query:** Given an edge $e = (s, d)$, edge query is to return the weight of $e$ estimated by the graph sketch. To answer the query, we first query the weight of $e$ in all the graph sketches. Specifically, for each graph sketch $\mathcal{K}_i$, we locate the corresponding position $M_i[H_i(s)][H_i(d)]$ and return the value in the position as the estimated weight. In this way, we can obtain a set of weights $\{w_1, w_2, \ldots, w_m\}$. According to the principle of count-min sketch [2], the minimal value of the set of sketches is used to estimate the value of the accumulative count, therefore we return $min\{w_1, w_2, \ldots, w_m\}$ for the edge query.

**Node Query:** Given a node $n$, node query is to return the aggregated edge weight *from* node $n$. To answer the query, for each graph sketch $\mathcal{K}_i$, we can first locate the row corresponding to node $n$ (i.e. the $H_i(n)$th row) in the adjacency

matrix $M_i$, and then sum up the values in that row to obtain a set of sums $\{sum_1, sum_2, \ldots, sum_m\}$. Similarly, we return $min\{sum_1, sum_2, \ldots, sum_m\}$ for the node query.

**Top-K Node Query:** Top-k node query is to return the list of top-k nodes with the highest aggregated weights in graph stream $S$. To answer this query, we additionally maintain a min-heap with size $k$ to store the top-k nodes. Specifically, after updating each edge $e = (s, d)$, we conduct a node query for node $s$, and obtain its aggregated weight $w_s$. Then, we push the tuple $(s, w_s)$ into the min-heap. If the min-heap is full, it will pop out the tuple with the lowest weight. After all edge updates, we return the set of nodes in the min-heap as the answer of top-k node query.

## 4 Dichotomy Graph Sketch Mechanism

In this section, we detailly introduce the proposed Dichotomy Graph Sketch (DGS) mechanism which is able to mitigate the performance problem caused by hash collisions between heavy edges and light edges. The proposed framework is illustrated in Fig. 2. Firstly, all edges in the current session are represented by a sub-graph stream $C = \{e_i, e_{i+1}, \ldots, e_j\}$. These edges are sequentially fed into an *edge classifier*. If an edge is classified as a heavy edge, it will be stored in the *heavy sketch* ($\mathcal{K}_h$); otherwise it will be stored in the *light sketch* ($\mathcal{K}_l$). In the first session, since the edge classifier has not been trained, all edges will be stored in the light sketch. In the subsequent sessions, a *sample generator* is applied to generate the training samples for the edge classifier. It obtains heavy edges in the current session by querying all heavy edges from the min-heap, and chooses the same amount of light edges in the current session with random sampling. The heavy edges together with the chosen light edges are used as training samples to train the edge classifier, which is used to construct dichotomy sketches. After each session, all values in the sample generator are reset to null, and DGS continues to process the next session to construct the graph sketches incrementally.

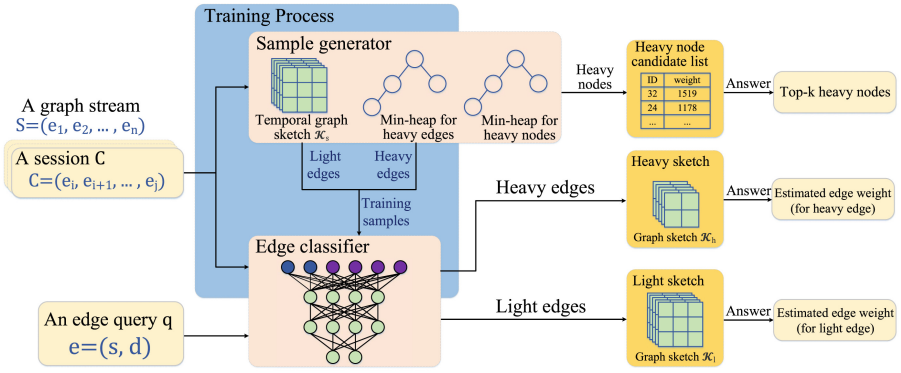



**Fig. 2.** The framework of our proposed dichotomy graph sketch.

The major components of DGS, i.e., the heavy sketch, the light sketch, the sample generator and the edge classifier, are introduced as follows.

### 4.1 Heavy Sketch and Light Sketch

Heavy sketch and light sketch both work exactly as a basic graph sketch (see definition 5 for details). When an edge in the graph stream arrives, it will first be fed into the edge classifier. 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.

Since the number of heavy edges is usually much smaller than the number of light edges in a graph stream, we set the size of heavy sketch smaller than that of the light sketch in practice.

### 4.2 Sample Generator

To obtain the heavy edges and the heavy nodes in each session, we design a sample generator which consists of a temporal graph sketch and two min-heaps. The temporal graph works exactly the same as a basic graph sketch, and the two min-heaps are maintained for obtaining heavy edges and heavy nodes, respectively. 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 Fig. 2) as the candidates for top-k nodes query. If a node is already in the heavy node candidate list, we simply update its aggregated weight by adding up the old weight and the new weight. Finally, before processing the edges in the next session, sample generator resets all the values of the temporal graph sketch to null, and clear the min-heaps.

### 4.3 Edge Classifier

To differentiate heavy edges from light edges during the edge updating process, we build a binary probabilistic classification model. The basic idea is to learn a model $f$ that can predict if an edge $e_i = (n_a, n_b)$ is a heavy edge 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). According to the discussion in Sect. 4.3, we find that *node embeddings* are very recognizable features to differentiate between heavy edges and light edges. Therefore the node embeddings are included in the input feature vector to train the DNN.

**Node embeddings using a graph auto-encoder** In graph representation learning, it is common to obtain node embeddings which are essentially feature representations to help accomplish downstream tasks. Thus, to help the classifier accurately classify heavy edges and light edges, we obtain node embeddings using a graph auto-encoder (GAE) [9] model.

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

$$GCN(\mathbf{X}, \mathbf{A}) = \tilde{\mathbf{A}} ReLU(\tilde{\mathbf{A}}\mathbf{X}\mathbf{W_0})\mathbf{W_1}, \tag{1}$$

where $ReLU(\cdot) = max(0, \cdot)$, 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 $\hat{\mathbf{A}} = \sigma(\mathbf{Z}\mathbf{Z}^{\mathbf{T}})$ 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:

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

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 $\mathbf{Z} = GCN(\mathbf{X}, \mathbf{A})$.

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

**Structure of the Edge Classifier.** The structure of the proposed deep neural network classifier is presented in Fig. 3. As mentioned previously, the input consists of two nodes (source node and destination node of an edge) and their corresponding node embeddings. The input first respectively goes through two fully-connected layers. Then, the output of the fully-connected layer is fed into a softmax layer, which outputs a probability distribution representing the probability that the input edge is a heavy edge. The details of the edge classifier is illustrated in Fig. 3.

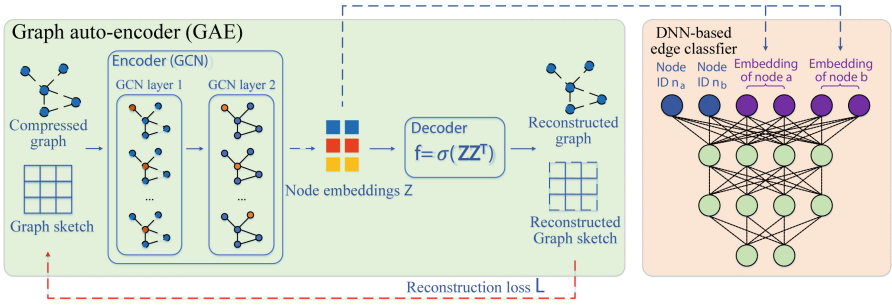



**Fig. 3.** The details of the proposed edge classifier.

### 4.4 Implementation of Graph Query Tasks

In this section, we introduce how DGS answers different kinds of queries.

**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 Sect. 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.

**Dealing with Top-K Node Query.** Since we store the heavy nodes of each session in the heavy node candidate list, to answer the top-k heavy node query, we can simply return the top-k heavy nodes with the highest weight in the heavy node candidate list.

## 5 Performance Evaluation

To validate the effectiveness of the proposed Dichotomy Graph Sketch (DGS), we conducted extensive experiments on three real-world graph stream datasets. We compare our method with two state-of-the-art graph sketches: **TCM** [15] and **GSS** [4]. All experiments were performed on a laptop with Intel Core i5-9300H processors (4 cores, 8 threads), 8 GB of memory, and NVIDIA GeForce RTX 2060 GPU. All sketches except GSS were implemented in Python. For GSS, we used the C++ source code provided on the Github[1].

### 5.1 Datasets

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

**Wiki_talk_cy** [2]**:** The first dataset is the communication network of the Welsh Wikipedia. Nodes represent users, and an edge from user A to user B denotes that user A wrote a message on the talk page of user B at a certain timestamp. It contains 2,233 users (nodes) and 10,740 messages (edges).

**Subelj_jung** [3]**:** 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. Nodes represent classes, and an edge between them indicates that there exists a dependency between two classes. It contains 6,210 classes (nodes) and 138,706 dependencies (edges).

[1] https://github.com/Puppy95/Graph-Stream-Sketch.
[2] http://konect.cc/networks/wiki_talk_cy/.
[3] http://konect.cc/networks/subelj_jung-j/.

**Facebook-wosn-wall** [4]**:** The third dataset is the directed network of a small subset of posts to other user's wall on Facebook. The nodes of the network are Facebook users, and each directed edge represents one post, linking the users writing a post to the users whose wall the post is written on. It contains 46,952 users (nodes) and 876,993 posts (edges).

### 5.2 Performance Metrics

We adopt the following metrics for performance evaluation in our experiments.
**Average relative error (ARE)** [15]**:** it measures the accuracy of the weights that are estimated by a graph sketch in the edge query task.
**Intersection accuracy (IA)** [15]**:** it measures the accuracy of the top-k heavy nodes reported by a graph sketch.
**Normalized discounted cumulative gain (NDCG)** [7]**:** it is a measure of ranking quality representing the usefulness (also called gain) of a node based on its position in the ranking list. Normally $NDCG \in [0, 1]$, and the higher NDCG means the stronger ability to find the top-k nodes.

### 5.3 Numerical Results

We analyze the numerical results for edge query and node query of different sketches. *For a fair comparison, the memory usages of TCM, GSS, and our proposed DGS are equal in both edge query task and node query task.* Note that the DGS framework refers to two hyperparameters: compression ratio (set to 1/10 by default), and the input size of the edge classifier (set to 32 by default).

**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 calculate the average relative error (ARE). Table 1, 2 and 3 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 querying 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 two datasets.

[4] http://konect.cc/networks/facebook-wosn-wall/.

**Table 1.** 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 2.** The ARE of edge query (subelj_jung)

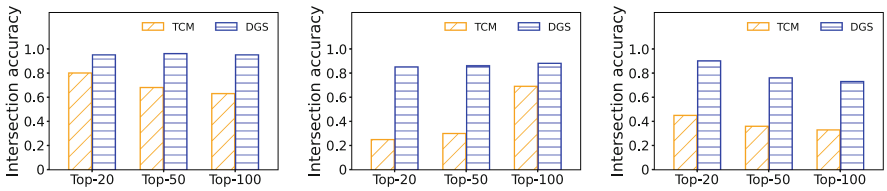
| Method | Total ARE | ARE of light edges | ARE of heavy edges |
|---|---|---|---|
| TCM | 22.359 | 22.399 | 2.079 |
| GSS | 39.386 | 39.458 | 3.320 |
| DGS | **17.264** | **17.295** | **1.433** |

**Table 3.** 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 |

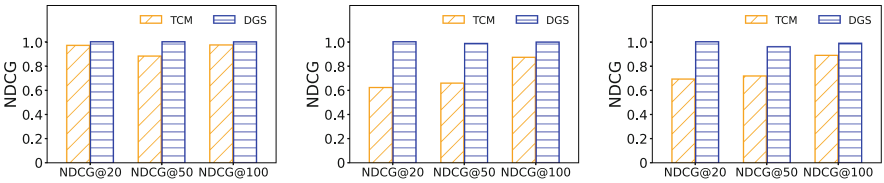
**Top-K Heavy Node Query.** We evaluate the ability to find top-k heavy nodes of DGS as well as TCM. We do not conduct this experiment with GSS since GSS does not support heavy node query. The results are shown in Fig. 4 and Fig. 5. As shown in Fig. 4, DGS outperforms TCM on all three datasets. Specifically, in the task of finding top-20 heavy nodes, DGS achieves an IA of 95%, 85% and 90% on dataset *wiki_talk_cy*, *subelj_jung*, and *facebook-wosn-wall*, respectively. In contrast, TCM only achieves an IA of 80%, 25%, and 45%. 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.

We also calculate the NDCG based on the result list of top-k heavy node query. The results are shown in Fig. 5. Similarly, the NDCG achieved by DGS is much higher than that of TCM.




(a) wiki_talk_cy (b) subelj_jung (c) facebook-wosn-wall

**Fig. 4.** Heavy node query (intersection accuracy)




(a) wiki_talk_cy (b) subelj_jung (c) facebook-wosn-wall

**Fig. 5.** Heavy node query (NDCG)

## 6 Conclusion

This paper proposed a novel framework for large-scale graph stream summarization called DGS. Different from conventional graph sketches that store all edges in a single matrix, DGS adopts two separate matrices, the heavy sketch and the light sketch, to respectively store heavy edges and light edges to avoid the serious performance drop caused by hash collisions. DGS designs a DNN-based binary classifier to decide whether an edge is heavy or not before storing it. 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. Extensive experiments based on three real-world graph streams showed that the proposed method is able to achieve high accuracy for graph queries compared to the state-of-the-arts.

## References

1. Charikar, M., Chen, K.C., Farach-Colton, M.: Finding frequent items in data streams. In: 29th International Colloquium on Automata, Languages and Programming, pp. 693–703 (2002)
2. Cormode, G., Muthukrishnan, S.: An improved data stream summary: the count-min sketch and its applications. J. Algorithm. **55**(1), 58–75 (2005)
3. Estan, C., Varghese, G.: New directions in traffic measurement and accounting: focusing on the elephants, ignoring the mice. ACM Trans. Comput. Syst. **21**(3), 270–313 (2003)
4. Gou, X., Zou, L., Zhao, C., Yang, T.: Fast and accurate graph stream summarization. In: The 35th IEEE International Conference on Data Engineering (ICDE 2019), pp. 1118–1129 (2019)
5. 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)
6. Hajiabadi, M., Singh, J., Srinivasan, V., Thomo, A.: Graph summarization with controlled utility loss. In: The 27th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Singapore, pp. 536–546 (2021)
7. Järvelin, K., Kekäläinen, J.: Cumulated gain-based evaluation of IR techniques. ACM Trans. Inform. Syst. **20**(4), 422–446 (2002)
8. Khan, A., Aggarwal, C.C.: Query-friendly compression of graph streams. In: 2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2016), pp. 130–137 (2016)

9. Kipf, T.N., Welling, M.: Variational graph auto-encoders (2016). arxiv.org/abs/1611.07308
10. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: 5th International Conference on Learning Representations (ICLR 2017), Toulon, France (2017)
11. Li, D., Li, W., Chen, Y., Lin, M., Lu, S.: Learning-based dynamic graph stream sketch. In: Advances in Knowledge Discovery and Data Mining - 25th Pacific-Asia Conference, Virtual Event (PAKDD 2021), vol. 12712, pp. 383–394 (2021)
12. Li, H., Chen, Q., Zhang, Y., Yang, T., Cui, B.: Stingy sketch: a sketch framework for accurate and fast frequency estimation. Proc. VLDB Endowm. **15**(7), 1426–1438 (2022)
13. Li, J., et al.: Wavingsketch: An unbiased and generic sketch for finding top-k items in data streams. In: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, Virtual Event (KDD 2020), pp. 1574–1584 (2020)
14. Liu, L., et al.: SF-sketch: A two-stage sketch for data streams. IEEE Trans. Parallel Distrib. Syst. **31**(10), 2263–2276 (2020)
15. Tang, N., Chen, Q., Mitra, P.: Graph stream summarization: From big bang to big crunch. In: The 2016 International Conference on Management of Data (SIGMOD 2016), pp. 1481–1496 (2016)
16. Zhang, Y., et al.: On-off sketch: a fast and accurate sketch on persistence. Proc. VLDB Endowm. **14**(2), 128–140 (2020)