

# Fast Gumbel-Max Sketch and its Applications

Yuanming Zhang, Pingui Wang, Yiyuan Qi, Kuankuan Cheng, Junzhou Zhao,  
Guangjian Tian and Xiaohong Guan

**Abstract**—The well-known Gumbel-Max Trick for sampling elements from a categorical distribution (or more generally a non-negative vector) and its variants have been widely used in areas such as machine learning and information retrieval. To sample a random element  $i$  in proportion to its positive weight  $v_i$ , the Gumbel-Max Trick first computes a Gumbel random variable  $g_i$  for each positive weight element  $i$ , and then samples the element  $i$  with the largest value of  $g_i + \ln v_i$ . Recently, applications including similarity estimation and weighted cardinality estimation require to generate  $k$  independent Gumbel-Max variables from high dimensional vectors. However, it is computationally expensive for a large  $k$  (e.g., hundreds or even thousands) when using the traditional Gumbel-Max Trick. To solve this problem, we propose a novel algorithm, *FastGM*, which reduces the time complexity from  $O(kn^+)$  to  $O(k \ln k + n^+)$ , where  $n^+$  is the number of positive elements in the vector of interest. FastGM stops the procedure of Gumbel random variables computing for many elements, especially for those with small weights. We perform experiments on a variety of real-world datasets and the experimental results demonstrate that FastGM is orders of magnitude faster than state-of-the-art methods without sacrificing accuracy or incurring additional expenses.

**Index Terms**—Gumbel-Max Trick, Sketching, Jaccard Similarity Estimation, Weighted Cardinality Estimation

## 1 INTRODUCTION

The Gumbel-Max Trick [2] is a popular technique for sampling elements from a categorical distribution (or more generally a non-negative vector), which has been widely used in many areas. Given a non-negative vector  $\vec{v} = (v_1, \dots, v_n)$ , let  $N_{\vec{v}}^+ \triangleq \{i: v_i > 0, i = 1, \dots, n\}$  be the set of indices of positive elements in  $\vec{v}$ . Then, the Gumbel-Max Trick computes a random variable  $s(\vec{v})$  as:

$$s(\vec{v}) \triangleq \operatorname{argmax}_{i \in N_{\vec{v}}^+} -\ln(-\ln a_i) + \ln v_i,$$

where  $a_i$  is a random variable drawn from the uniform distribution UNI(0, 1) independently and so  $g_i = -\ln(-\ln a_i)$  is a Gumbel random variable. Note that different vectors  $\vec{v}$  should use the same set of variables  $a_1, \dots, a_n$  to guarantee consistency. The distribution of random variable  $s(\vec{v})$  is  $P(s(\vec{v}) = i) = \frac{v_i}{\sum_{j=1}^n v_j}$ . Therefore, the Gumbel-Max Trick is popularly applied to sample an element from a

- An earlier conference version of this paper appeared at the Proceedings of The Web Conference 2020 [1]. In this extended version, we extend the Gumbel-Max sketch's definition, which can be applied to more applications. Compared with the conference version, we also propose a more efficient method *FastGM* to compute the Gumbel-Max sketch and include new experiments for the application of weighted cardinality estimation.
- Y. Zhang, P. Wang, Y. Qi, K. Cheng, and J. Zhao are with the MOE Key Laboratory for Intelligent Networks and Network Security, Xi'an Jiaotong University, P.O. Box 1088, No. 28, Xianning West Road, Xi'an, Shaanxi 710049, China. E-mail: {zhangyuanming, kuankuan.cheng}@stu.xjtu.edu.cn, phwang@mail.xjtu.edu.cn, qiyiyuan@idea.edu.cn, junzhou.zhao@xjtu.edu.cn.
- X. Guan is with the MOE Key Laboratory for Intelligent Networks and Network Security, Xi'an Jiaotong University, P.O. Box 1088, No. 28, Xianning West Road, Xi'an, Shaanxi 710049, China and also with the Center for Intelligent and Networked Systems, Tsinghua National Lab for Information Science and Technology, Tsinghua University, Beijing 100084, China. E-mail: xhguan@mail.xjtu.edu.cn.
- G. Tian is with Huawei Noah's Ark Lab, Hong Kong. E-mail: Tian.Guangjian@huawei.com.

Corresponding author: Pingui Wang  
Manuscript received April 19, 2005; revised August 26, 2015.

high-dimensional non-negative vector  $\vec{v}$  with probability proportional to the element's weight.

We call  $s(\vec{v})$  and  $x(\vec{v}) = \max_{i \in N_{\vec{v}}^+} -\ln(-\ln a_i) + \ln v_i$  as Gumbel-ArgMax and Gumbel-Max variables of vector  $\vec{v}$ , respectively. In this paper, we define a Gumbel-Max sketch of vector  $\vec{v}$  as a vector of  $k$  Gumbel-Max variables generated independently, i.e.,  $\vec{x}(\vec{v}) = (x_1(\vec{v}), \dots, x_k(\vec{v}))$ , where  $x_j(\vec{v}) = \max_{i \in N_{\vec{v}}^+} -\ln(-\ln a_{i,j}) + \ln v_i$ ,  $j = 1, \dots, k$  and  $a_{i,j} \sim \text{UNI}(0, 1)$ . Similarly, we define a Gumbel-ArgMax sketch of vector  $\vec{v}$  as a vector of  $k$  Gumbel-ArgMax variables generated independently, i.e.,  $\vec{s}(\vec{v}) = (s_1(\vec{v}), \dots, s_k(\vec{v}))$ . For simplicity, we also name the Gumbel-ArgMax sketch as the Gumbel-Max sketch when no confusion arises. We observe that the Gumbel-Max sketch has been actually exploited for applications including probability Jaccard similarity estimation [3], [4], [5], [6], [7] and weighted cardinality estimation [8], while the authors of these works might be unconscious of this.

**Probability Jaccard Similarity Estimation.** Similarity estimation lies at the core of many data mining and machine learning applications, such as web duplicate detection [9], [10], collaborate filtering [11] and association rule learning [12]. To efficiently estimate the similarity between two vectors, several algorithms [3], [4], [5], [6] compute  $k$  random variables  $-\frac{\ln a_{i,1}}{v_i}, \dots, -\frac{\ln a_{i,k}}{v_i}$  for each positive element  $v_i$  in  $\vec{v}$ , where  $a_{i,1}, \dots, a_{i,k}$  are independent random variables drawn from the uniform distribution UNI(0, 1). Then, these algorithms build a sketch of vector  $\vec{v}$  consisting of  $k$  registers, and each register records  $s_j(\vec{v})$  where

$$s_j(\vec{v}) = \operatorname{argmin}_{i \in N_{\vec{v}}^+} -\frac{\ln a_{i,j}}{v_i}, \quad 1 \leq j \leq k. \quad (1)$$

We find that  $s_j(\vec{v})$  is exactly a Gumbel-ArgMax variable of vector  $\vec{v}$  as  $\operatorname{argmin}_{i \in N_{\vec{v}}^+} -\frac{\ln a_{i,j}}{v_i} = \operatorname{argmax}_{i \in N_{\vec{v}}^+} \ln v_i - \ln(-\ln a_{i,j})$ . Let  $\mathbb{1}(x)$  be an indicator function. Yang et al. [3], [4], [5] use  $\frac{1}{k} \sum_{j=1}^k \mathbb{1}(s_j(\vec{u}) = s_j(\vec{v}))$  to estimate the weighted

*Jaccard similarity* of two non-negative vectors  $\vec{u}$  and  $\vec{v}$  which is defined by

$$\mathcal{J}_W(\vec{u}, \vec{v}) \triangleq \frac{\sum_{i=1}^n \min\{u_i, v_i\}}{\sum_{i=1}^n \max\{u_i, v_i\}}.$$

Recently, Moulton et al. [6] prove that the expectation of estimate  $\frac{1}{k} \sum_1^k \mathbb{1}(s_j(\vec{u}) = s_j(\vec{v}))$  actually equals the *probability Jaccard similarity*, which is defined by

$$\mathcal{J}_P(\vec{u}, \vec{v}) \triangleq \sum_{i \in N_{\vec{v}, \vec{u}}^+} \frac{1}{\sum_{l=1}^n \max\left(\frac{u_l}{v_i}, \frac{v_l}{v_i}\right)}.$$

Here,  $N_{\vec{v}, \vec{u}}^+ \triangleq \{i : v_i > 0 \wedge u_i > 0, i = 1, \dots, n\}$  is the set of indices of positive elements in both  $\vec{v}$  and  $\vec{u}$ . Compared with the weighted Jaccard similarity  $\mathcal{J}_W$ , Moulton et al. demonstrate that the probability Jaccard similarity  $\mathcal{J}_P$  is scale-invariant and more sensitive to changes in vectors. Moreover, each function  $s_j(\vec{v})$  maps similar vectors to the same value with a high probability. Therefore, similar to regular locality-sensitive hashing (LSH) schemes [13], [14], [15], one can use these Gumbel-Max sketches to build an LSH index for fast similarity search in a large dataset, which is capable to search similar vectors for any query vector in sub-linear time.

**Weighted Cardinality Estimation.** Given a sequence  $\Pi = o_1 o_2 \dots$ , where  $o_j \in \{1, \dots, n\}$  represents an object (e.g., a string) and each object  $i \in \{1, \dots, n\}$  may appear more than once. Each object  $i$  has a positive weight  $v_i$ . Let  $N_\Pi$  be the set of objects that occurred in  $\Pi$ . Then, the weighted cardinality of  $\Pi$  is defined as  $c_\Pi = \sum_{i \in N_\Pi} v_i$ . Take a SQL query "SELECT DISTINCT CompanyNames FROM Orders" as an instance. The size (in bytes) of the query result is a sum weighted by string length over the "CompanyNames". Besides the cardinality of a single sequence, sometimes, the data of interest consists of multiple sequences distributed over different locations and the target is to estimate the sum of all *unique* occurred objects' weights using as few resources (including memory space, computation time, and communication cost) as possible. The state-of-the-art method of weighted cardinality estimation is Lemiesz's sketch [8]. Let  $\vec{v}^\Pi = (v_1^\Pi, \dots, v_n^\Pi)$  be the underlying vector of sequence  $\Pi$ . That is, each element  $v_i^\Pi, i = 1, \dots, n$  equals  $v_i$  (i.e., the weight of object  $i$ ) when object  $i$  occurs in sequence  $\Pi$  (i.e.,  $i \in N_\Pi$ ) and 0 otherwise. Lemiesz computes a sketch  $\vec{y}(\vec{v}^\Pi) = (y_1(\vec{v}^\Pi), \dots, y_k(\vec{v}^\Pi))$ , and  $y_j(\vec{v}^\Pi)$  is defined as:

$$y_j(\vec{v}^\Pi) = \min_{i \in N_\Pi} -\frac{\ln a_{i,j}}{v_i}, \quad 1 \leq j \leq k, \quad (2)$$

where all variables  $a_{i,j} \sim \text{UNI}(0, 1)$  are independent with each other. Then, we easily find that the Gumbel-Max variable  $x_j(\vec{v}^\Pi) = \max_{i \in N_\Pi} -\ln(-\ln a_{i,j}) + \ln v_i = -\ln y_j(\vec{v}^\Pi)$ . Therefore, Lemiesz's sketch is a variant of the Gumbel-Max sketch. It is easy to find that each  $y_j(\vec{v}^\Pi)$  follows the exponential distribution  $\text{EXP}(c_\Pi)$  because  $P(y_j(\vec{v}^\Pi) \geq t) = \prod_{i \in N_\Pi} P\left(-\frac{\ln a_{i,j}}{v_i} \geq t\right) = e^{-c_\Pi t}$ . Therefore, the sum  $\sum_{j=1}^k y_j(\vec{v}^\Pi)$  follows the gamma distribution  $\Gamma(k, c_\Pi)$ . Based on the above observation, Lemiesz's algorithm estimates the weighted cardinality  $c_\Pi$  as  $\frac{k-1}{\sum_{j=1}^k y_j(\vec{v}^\Pi)}$ . The proposed sketch is mergeable, which facilitates efficiently estimating the weighted cardinality of a sequence represented as a

joint of different sequences  $\Pi_1, \dots, \Pi_d$ . Specifically, given the sketches of all  $\Pi_1, \dots, \Pi_d$ , the sketch of the joint sequence  $\Pi$  is computed as:

$$y_j(\vec{v}^\Pi) = \min_{l=1, \dots, d} y_j(\vec{v}^{\Pi_l}).$$

Therefore, we only need to compute and gather the sketches of all sequences  $\Pi_1, \dots, \Pi_d$  together, which significantly reduces the memory usage and communication cost.

To compute the Gumbel-Max sketches of a large collection of vectors (e.g., bag-of-words representations of documents), the straightforward method first instantiates variables  $a_{i,1}, \dots, a_{i,k}$  from  $\text{UNI}(0, 1)$  for each index  $i = 1, \dots, n$ . Then, for each non-negative vector  $\vec{v}$ , it enumerates each  $i \in N_{\vec{v}}^+$  and computes  $-\frac{\ln a_{i,1}}{v_i}, \dots, -\frac{\ln a_{i,k}}{v_i}$ . The above method requires memory space  $O(nk)$  to store all  $[a_{i,j}]_{1 \leq i \leq n, 1 \leq j \leq k}$ , and time complexity  $O(kn_{\vec{v}}^+)$  to obtain the Gumbel-Max sketch of each vector  $\vec{v}$ , where  $n_{\vec{v}}^+ = |N_{\vec{v}}^+|$  is the cardinality of set  $N_{\vec{v}}^+$ . We note that  $k$  is usually set to be hundreds or even thousands [6], [7], [16]. Therefore, the straightforward method costs a huge amount of memory space and time when the vector of interest has a large dimension, e.g.,  $n = 10^9$ . To reduce the memory cost, one can easily use hash techniques or random number generators with specific seeds (e.g., consistent random number generation methods in [17], [18], [19]) to generate each of  $a_{i,1}, \dots, a_{i,k}$  on the fly, which does not require to calculate and store variables  $[a_{i,j}]_{1 \leq i \leq n, 1 \leq j \leq k}$  in memory.

To address the computational challenge, in this paper, we propose a novel method *FastGM* to fast compute a Gumbel-Max sketch, which reduces the time complexity of computing the sketch from  $O(kn_{\vec{v}}^+)$  to  $O(k \ln k + n_{\vec{v}}^+)$ . From the example in Fig. 1, we find two interesting observations in computing the Gumbel-Max sketch in a straightforward way: 1) The  $k$  variables  $-\frac{\ln a_{i,1}}{v_i}, \dots, -\frac{\ln a_{i,k}}{v_i}$  of a relatively larger element  $v_i$  in the  $\vec{v}$  are more likely to be the Gumbel-Max variables, such as the  $v_1 = 0.3$  and  $v_5 = 0.2$ ; 2) Each Gumbel-Max variable occurs as one of an element  $v_i$ 's top minimal variables, e.g., all three Gumbel-Max variables (red ones in the first row) appeared in  $v_1$ 's Top-4 minimal variables. Therefore, we prioritize the generation order of all  $kn$  variables and reduce the number of generated variables for computing the Gumbel-Max sketch. The basic idea behind our *FastGM* can be summarized as follows. For each element  $v_i > 0$  in  $\vec{v}$ , we generate  $k$  random variables  $-\frac{\ln a_{i,1}}{v_i}, \dots, -\frac{\ln a_{i,k}}{v_i}$  in ascending order. As shown in Fig. 2, we can generate a sequence of  $k$  tuples  $(t_{i,1}, \pi_{i,1}), \dots, (t_{i,k}, \pi_{i,k})$ , where  $t_{i,j} = -\frac{\ln a_{i,\pi_{i,j}}}{v_i}$ ,  $t_{i,1} < \dots < t_{i,k}$  and  $\pi_{i,j} = i_j, (i_1, \dots, i_k)$  is a random permutation of integers  $1, \dots, k$ . When we are able to compute the Gumbel-Max sketch of  $\vec{v}$  by obtaining the  $k$  random variables in ascending order, it is easy to find once the current obtained  $t_{i,j}$  in tuple  $(t_{i,j}, \pi_{i,j})$  is larger than all elements in the  $\vec{y}(\vec{v})$ , there is no need to obtain the following tuples  $(t_{i,j+1}, \pi_{i,j+1}), \dots, (t_{i,k}, \pi_{i,k})$  because they have no chance to change the Gumbel-Max sketch of  $\vec{v}$ . Based on this property, we model the procedure of computing the Gumbel-Max sketch as a queuing model with  $k$ -servers and  $n$ -queues of different arrival rates. Specifically, each queue has  $k$  customers and each customer randomly selects a server. A server  $j = 1, \dots, k$  just serves the first arrived customer and ignores the other arrived customers. In addition, a server

$\vec{v}$	$j=1$	$\dots$			$j=8$				
$v_i$	$i=1$	7.33	0.51	5.48	0.17	0.37	6.46	7.86	2.16
	$i=2$	8.98	31.38	2.82	6.98	5.40	5.01	11.30	11.23
	$i=3$	2.46	7.37	21.21	5.62	10.80	27.37	5.24	3.68
	$i=4$	12.02	0.42	17.42	5.96	3.13	22.53	29.04	4.21
	$i=5$	4.98	6.31	9.30	0.92	16.78	4.56	1.50	2.93
	$i=6$	22.41	28.74	2.85	26.50	1.74	12.86	5.24	19.08
	$i=7$	8.19	3.78	4.70	7.82	8.97	9.51	1.93	3.85
	$i=8$	1.49	3.16	10.10	16.04	98.46	16.49	6.29	48.46
the matrix $\left[ -\frac{\ln a_{i,j}}{v_i} \right]_{1 \leq i \leq 8, 1 \leq j \leq 8}$									
$\vec{s}(\vec{v})$		8	4	2	1	1	5	5	1
$\vec{y}(\vec{v})$		1.49	0.42	2.82	0.17	0.37	4.56	1.50	2.16

Fig. 1: An example of computing the Gumbel-Max sketch  $\vec{y}(\vec{v})$  and  $\vec{s}(\vec{v})$  of length  $k = 8$  for a vector  $\vec{v}$ , of which elements  $y_j(\vec{v})$  and  $s_j(\vec{v})$  respectively record the smallest element value (i.e., the red one) and its index in the  $j$ -th column of the matrix.

$j$  only records the arrival time and the queue number (i.e., from which queue the customer comes) of its first arrived customer as  $y_j(\vec{v})$  and  $s_j(\vec{v})$ ,  $j = 1, \dots, k$  respectively, which have the same probability distributions as the variables  $y_j(\vec{v})$  and  $s_j(\vec{v})$  defined in Eq. (2) and Eq. (1). When each of the servers has processed its first arrived customer, we close all queues and obtain the Gumbel-Max sketch  $y_j(\vec{v})$  and  $s_j(\vec{v})$  of  $\vec{v}$ . Based on the above model, we propose *FastGM* to fast compute the Gumbel-Max sketch. We summarize our main contributions as:

- We introduce a simple queuing model to interpret the procedure of computing the Gumbel-Max sketch of vector  $\vec{v}$ . Using this stochastic process model, we propose a novel algorithm, called *FastGM*, to reduce the time complexity of computing the Gumbel-Max sketch  $(s_1(\vec{v}), \dots, s_k(\vec{v}))$  and  $(y_1(\vec{v}), \dots, y_k(\vec{v}))$  from  $O(n_{\vec{v}}^+ k)$  to  $O(k \ln k + n_{\vec{v}}^+)$ , which is achieved by avoiding calculating all  $k$  variables  $-\frac{\ln a_{i,1}}{v_i}, \dots, -\frac{\ln a_{i,k}}{v_i}$  for each  $i \in N_{\vec{v}}^+$ .
- We conduct experiments on a variety of real-world datasets for applications including probability Jaccard similarity estimation and weighted cardinality estimation. The experimental results demonstrate that our method *FastGM* is orders of magnitude faster than the state-of-the-art methods without incurring any additional cost.

The rest of this paper is organized as follows. Section 2 and Section 3 present our method *FastGM* and its extension *Stream-FastGM* for non-streaming and streaming settings respectively. The performance evaluation and testing results are presented in Section 4. Section 5 summarizes related work. Concluding remarks then follow.

## 2 OUR METHOD FASTGM

In this section, we first introduce the basic idea behind our method *FastGM* through a simple example. Then, we elaborate on *FastGM* in detail and discuss its space and time complexities.

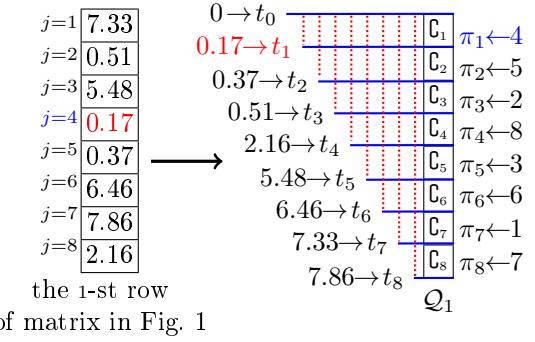


Fig. 2: An example of building a queue  $Q_1$  from 8 random variables in the 1-st row of matrix in Fig. 1. We use  $C$  to represent a customer in the queue.

### 2.1 Basic Idea

In Fig. 1, we provide an example of generating a Gumbel-Max sketch of a vector  $\vec{v} = (0.3, 0.1, 0.05, 0.05, 0.2, 0.07, 0.1, 0.03)$  to illustrate our basic idea, where we have  $n = 8$  and  $k = 8$ . Note that we aim to fast compute each  $y_j(\vec{v})$  and  $s_j(\vec{v})$ , where  $y_j(\vec{v}) = \min_{1 \leq i \leq 8} -\frac{\ln a_{i,j}}{v_i}$  and  $s_j(\vec{v}) = \arg\min_{1 \leq i \leq 8} -\frac{\ln a_{i,j}}{v_i}$ ,  $1 \leq j \leq 8$ , i.e., in each column  $j$  of matrix  $\left[ -\frac{\ln a_{i,j}}{v_i} \right]_{1 \leq i \leq 8, 1 \leq j \leq 8}$ ,  $y_j(\vec{v})$  records the minimum element and  $s_j(\vec{v})$  records the index of this element. We generate matrix  $\left[ -\frac{\ln a_{i,j}}{v_i} \right]_{1 \leq i \leq 8, 1 \leq j \leq 8}$  based on the traditional Gumbel-Max Trick and mark the minimum element (i.e., the red one indicating the Gumbel-Max variable) in each column  $j$ . We find that Gumbel-Max variables tend to equal index  $i$  with large weight  $v_i$ . For example, among the values of all Gumbel-Max variables  $s_1(\vec{v}), \dots, s_8(\vec{v})$ , index 1 with  $v_1 = 0.3$  appears 3 times, while index 3 with  $v_3 = 0.05$  never occurs. Based on the above observations, we prioritize the generation order of the 64 variables  $-\frac{\ln a_{i,1}}{v_i}, \dots, -\frac{\ln a_{i,k}}{v_i}$ ,  $1 \leq i \leq 8, 1 \leq j \leq 8$  according to their values. We first respectively select  $R_i$  smallest variables from each row  $i$  to compute the Gumbel-Max sketch, where  $R_i$  is proportional to the weight  $v_i$ . The total number  $R = R_1 + \dots + R_n$  of variables from all rows is computed as  $R = k \ln k$ . This is the expected number of generated variables before each column  $j$  has at least one variable. To some extent, it is similar to the *Coupon collector's problem* [20]. Specifically, in the example of Fig. 1, we have  $R = 17 = \lceil 8 \times \ln 8 \rceil$ , and each  $R_i$  is computed as  $R_i = \lceil Rv_i^* \rceil$ , where  $\vec{v}^* = (v_1^*, \dots, v_8^*)$  is the normalized vector of  $\vec{v}$ . We have  $R_1 = 6$ ,  $R_2 = 2$ ,  $R_3 = 1$ ,  $R_4 = 1$ ,  $R_5 = 4$ ,  $R_6 = 2$ ,  $R_7 = 2$ , and  $R_8 = 1$ . Meanwhile, we find that each Gumbel-Max variable occurs as one of a row  $i$ 's Top- $R_i$  minimal elements. For example, the two Gumbel-Max variables occurring in the 5-th row are all among the Top- $R_5$  (i.e., Top-4) minimal elements. Moreover, we easily observe that  $k$  random variables  $-\frac{\ln a_{i,1}}{v_i}, \dots, -\frac{\ln a_{i,k}}{v_i}$  in each row indeed are  $k$  independent random variables follow the exponential distribution  $\text{EXP}(v_i)$ . Therefore, we can generate these  $k$  variables in ascending order by exploiting the distribution of the order statistics of exponential random variables. Based on the above insights, we derive our method *FastGM*. As the example in Fig. 2, for each row, we construct such a queue  $Q_i$  with arrival rate  $v_i$  for the  $k$  variables drawn

from the distribution  $\text{EXP}(v_i)$  according to their values. Then, we first compute the variables that are in the front of the queues or in the queues with large arrival rates  $v_i$ , because they are smaller ones among all  $kn$  variables and are more likely to become the Gumbel-Max variables. Moreover, we early stop a queue when its remaining variables have no chance to be the Gumbel-Max variables. Also take Fig. 1 as an example. Compared with the straightforward method computing all  $nk = 64$  random variables, we compute  $s_1(\vec{v}), \dots, s_k(\vec{v})$  by only obtaining Top- $R_i$  minimal elements of each row  $i$ , which significantly reduces the computation cost to around  $\sum_{i=1}^8 R_i = 19$ . In summary, our method FastGM efficiently computes the Gumbel-Max sketch  $\vec{y}(\vec{v})$  and  $\vec{s}(\vec{v})$  of vector  $\vec{v}$  through managing the number and order of variables  $-\frac{\ln a_{i,j}}{v_i}$  belonging to different elements  $v_i$ . Specifically, we aim to *fast search* and compute those variables that have a high probability to become the elements of the Gumbel-Max sketch, and *fast prune* variables have no chance to be an element in the Gumbel-Max sketch. In the following, when no confusion arises, we simply write  $s_j(\vec{v})$ ,  $y_j(\vec{v})$  and  $n_{\vec{v}}^+$  as  $s_j$ ,  $y_j$  and  $n^+$  respectively.

## 2.2 Fast Gumbel-Max Sketch Generation

Our FastGM first constructs a queue  $\mathcal{Q}_i$  for variables in each row as shown in Fig. 2. Based on this, we propose two modules: **FastSearch** for efficiently searching small variables in each queue, and **FastPrune** for pruning overlarge queues that cannot contribute to the Gumbel-Max sketch. Before introducing our FastGM in detail, we first illustrate how to build a queue  $\mathcal{Q}_i$  and model the procedure of computing the Gumbel-Max sketch from another perspective via a *Queuing Model with k-servers and n-queues*.

**Queuing Model with  $k$ -servers and  $n$ -queues.** In Fig. 2, we show how to construct a queue  $\mathcal{Q}_i$  where  $k$  random variables  $-\frac{\ln a_{i,1}}{v_i}, \dots, -\frac{\ln a_{i,k}}{v_i}$  of  $v_i$  are sorted in ascending order. For simplicity, we define a variable  $b_{i,j}$  as:

$$b_{i,j} = \frac{-\ln a_{i,j}}{v_i}, \quad i = 1, \dots, n \quad j = 1, \dots, k. \quad (3)$$

We easily observe that  $b_{i,1}, \dots, b_{i,k}$  are equivalent to  $k$  independent random variables generated according to the exponential distribution  $\text{EXP}(v_i)$ . Let  $b_{i,(1)} < b_{i,(2)} < \dots < b_{i,(k)}$  be the order statistics corresponding to variables  $b_{i,1}, \dots, b_{i,k}$ . We construct each queue  $\mathcal{Q}_i$  with  $k$  customers  $\mathcal{C}_{i,j}$  whose arrival time  $t_{i,j} \triangleq b_{i,(j)}, j = 1, \dots, k$  and each customer randomly selects a server  $j$ . Specifically, we generate a sequence of  $k$  tuples  $(b_{i,(1)}, i_1), \dots, (b_{i,(k)}, i_k)$ , where  $(i_1, \dots, i_k)$  is a random permutation of integers  $1, \dots, k$  and denotes the server sequence randomly selected by customers in a queue  $\mathcal{Q}_i$ . It is easy to observe that values (resp. positions) of element  $v_i$ 's  $k$  variables are the customers' arrival time (resp. selected servers) in the queue  $\mathcal{Q}_i$ . Accordingly, we assign arrival time  $t_{i,j} \triangleq b_{i,(j)}$  and selected server  $\pi_{i,j} \triangleq i_j$  for each customer, i.e.,  $(t_{i,j}, \pi_{i,j}) \triangleq (b_{i,(j)}, i_j), j = 1, \dots, k$ . Note that,  $k$  variables  $b_{i,1}, \dots, b_{i,k}$  follow  $\text{EXP}(v_i)$  with a rate parameter  $v_i$ . Therefore, customers in queue  $\mathcal{Q}_i$  also arrive at this rate  $v_i$ . As shown in Fig. 3, based on the built queues, the procedure of computing the Gumbel-Max sketch can be modeled as a *Queuing Model with k-servers and n-queues*, where each server only serves the first arrived customer

(i.e., records this customer's arrival time and the index of queue  $i$ , same as Eq. (2) and Eq. (1)). Then, we naturally have the following two fundamental questions for the design of FastGM:

**Question 1.** How to fast search customers with the smallest arrival time to become candidates for the servers from these  $N_{\vec{v}}^+$  queues?

**Question 2.** How to early stop a queue  $\mathcal{Q}_i, i \in N_{\vec{v}}^+$ ?

We first discuss Question 1. We note that customers of different queues  $\mathcal{Q}_i$  arrive at different rates  $v_i$ . Recall the example in Fig. 1, the basic idea behind the following technique is that queue  $\mathcal{Q}_i$  with a high rate  $v_i$  is more likely to produce customers with the smallest arrival time (i.e. Gumbel-Max variables). Especially, when  $z$  customers have arrived, let  $t_{i,z}$  denote the arrival time of the  $z$ -th customer in queue  $\mathcal{Q}_i$ . We find that  $t_{i,z}$  can be represented as the sum of  $z$  identically distributed exponential random variables with mean  $\frac{1}{kv_i}$  (another perspective can be found in paper [1]). Therefore, the expectation and variance of variable  $t_{i,z}$  are computed as

$$\mathbb{E}(t_{i,z}) = \frac{z}{kv_i}, \quad \text{Var}(t_{i,z}) = \frac{z}{k^2 v_i^2}. \quad (4)$$

We easily find that  $\mathbb{E}(t_{i,z})$  is  $l$  times smaller than  $\mathbb{E}(t_{j,z})$  when  $v_i$  is  $l$  times larger than  $v_j$ .

To obtain the first  $R$  customers of the joint of all queues  $\mathcal{Q}_i, i \in N_{\vec{v}}^+$ , we let each queue  $\mathcal{Q}_i$  release  $R_i = \lceil Rv_i^* \rceil$  customers, where  $\vec{v}^*$  is the normalized vector of  $\vec{v}$ . Then, we have  $R \approx \sum_{i=1}^n R_i$ . For all  $i \in N_{\vec{v}}^+$ , their  $t_{i,R_i}$  approximately have the same expectation.

$$\mathbb{E}(t_{i,R_i} | R) \approx \frac{R}{k \sum_{j=1}^n v_j}, \quad i \in N_{\vec{v}}^+. \quad (5)$$

Therefore, the  $R$  customers with the smallest arrival time are expected to be released.

Next, we discuss Question 2, which is inspired by the generation of ascending-order random variables. For an element with index  $j$  in the Gumbel-Max Sketch, we use two registers  $y_j$  and  $s_j$  to keep track of information on the customer with the smallest arrival time among all the released customers selected by server  $j$ , where  $y_j$  records the customer's arrival time and  $s_j$  records the index of the queue this customer comes from, i.e.,  $\mathcal{Q}_{s_j}$ . When all servers  $1, \dots, k$  have been selected by at least one customer, we let  $y^*$  keep track of the maximum value of  $y_1, \dots, y_k$ , i.e.,

$$y^* = \max_{j=1, \dots, k} y_j.$$

Then, we can stop queues  $\mathcal{Q}_i$  when a customer coming from  $\mathcal{Q}_i$  has an arrival time larger than  $y^*$  because the arrival time of the subsequent customers from  $\mathcal{Q}_i$  is also larger than  $y^*$ , which will not change any  $y_1, \dots, y_k$  and  $s_1, \dots, s_k$ .

Based on the above two discussions, we develop our method FastGM to fast generate a  $k$ -length Gumbel-Max sketch with  $\vec{s}_{\vec{v}} = (s_1, \dots, s_k)$  and  $\vec{y}_{\vec{v}} = (y_1, \dots, y_k)$  of any non-negative vector  $\vec{v}$ . As shown in Fig. 3, FastGM consists of two modules: FastSearch and FastPrune. FastSearch is designed to quickly search customers with the smallest arrival time coming from all queues  $\mathcal{Q}_1, \dots, \mathcal{Q}_n$  and check whether all servers  $1, \dots, k$  have received at least one appointment from customers (i.e., selected by at least one customer). When

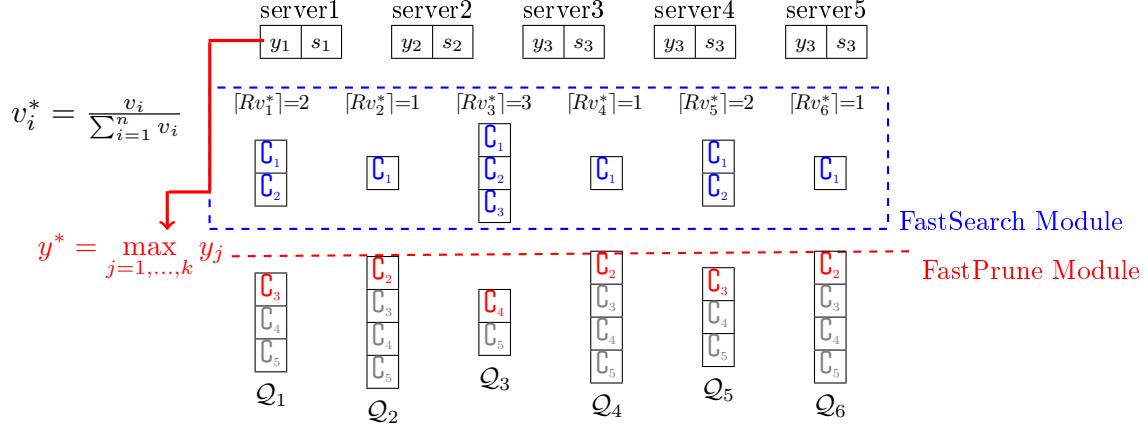


Fig. 3: Illustration of our FastGM. The FastSearch module first selects a number of customers  $\mathbb{C}$  from each queue as candidates (i.e., the blue ones) and uses their arrival time to compute the  $y^* = \max_{j=1, \dots, k} y_j$ . Then, the FastPrune module closes a queue according to the arrival time of the next customer (i.e., the red one) in each queue compared with the value of  $y^*$ , if  $y^*$  is smaller, we close the queue for the rest of customers (i.e., the gray ones), which have no chance to be the first customers for any servers.

no servers are unreserved, we start the FastPrune module to close each queue  $\mathcal{Q}_i, i \in N_v^+$ . We perform the procedure of FastPrune because following customers coming from  $\mathcal{Q}_i$  may also have an arrival time smaller than  $y^*$  and the customers may become the first arrived customers for some servers  $j$  and change the values of  $y_j$  and  $s_j$  after the procedure of FastSearch. Before we introduce these two modules in detail, we first elaborate on the method of generating exponential random variables in ascending order, which is a building block for both modules.

• **Generating Ascending Exponential Random Variables:** Next we detail how to sequentially generate  $k$  random variables  $b_{i,1}, \dots, b_{i,k}$  in ascending order for each positive element  $v_i$  of vector  $\vec{v}$  (Lines 9-14 and Lines 24-29 in Algorithm 1). As we mentioned, these  $b_{i,1}, \dots, b_{i,k}$  are random variables according to the exponential distribution  $\text{EXP}(v_i)$ , i.e.,

$$b_{i,j} \sim \text{EXP}(v_i), \quad j = 1, \dots, k. \quad (6)$$

Let  $b_{i,(1)} < b_{i,(2)} < \dots < b_{i,(k)}$  be the order statistics corresponding to variables  $b_{i,1}, \dots, b_{i,k}$ . Alfré Rényi [21] observes that each  $b_{i,(z)}, z = 1, \dots, k$  satisfies

$$b_{i,(z)} \sim \frac{1}{v_i} \left( \sum_{n=1}^z \frac{-\ln u_{i,n}}{k-n+1} \right), \quad (7)$$

where all variables  $u_{i,1}, \dots, u_{i,z} \sim \text{UNI}(0, 1)$  are independent random variables. Note that  $-\ln u_{i,n}$  is an EXP(1) distributed random variable. Therefore, one easily obtains the following equation:

$$b_{i,(z)} - b_{i,(z-1)} \sim \frac{1}{v_i} \left( \frac{-\ln u_{i,z}}{k-z+1} \right), \quad 2 \leq z \leq k. \quad (8)$$

Based on the above observation, we generate the order statistics  $b_{i,(1)}, \dots, b_{i,(k)}$  for each element  $v_i$  of vector  $\vec{v}$  in an iterative way as:

$$b_{i,(z)} \leftarrow b_{i,(z-1)} + \frac{1}{v_i} \left( \frac{-\ln u_{i,z}}{k-z+1} \right), \quad 1 \leq z \leq k,$$

where  $b_{i,(0)} = 0$ . In addition, we use the Fisher-Yates shuffle [22] (Lines 11-12 and lines 26-27 in Algorithm 1)

to iteratively produce a random permutation  $i_1, \dots, i_k$  for integers  $1, \dots, k$ . For an array  $(\pi_{i,1}, \dots, \pi_{i,k})$  with elements  $(1, \dots, k)$ , in each step  $z$ ,  $1 \leq z \leq k$ , this method randomly selects a number  $i_z$  from  $z, z+1, \dots, k$  and swaps the two elements in the array with indices  $z$  and  $i_z$ . To build the queue  $\mathcal{Q}_i$ , we assign  $b_{i,(z)}$  and the element with index  $z$  in the array to the arrival time and selected server of  $z$ -th customer, respectively, i.e.,

$$t_{i,z} \leftarrow b_{i,(z)}, \quad i_z \leftarrow \pi_{i,z}.$$

We easily find that  $k$  variables  $b_{i,(1)}, \dots, b_{i,(k)}$  shuffled by the random permutation  $i_1, \dots, i_k$  have the same distribution as the variables  $b_{i,1}, \dots, b_{i,k}$  generated in a direct manner.

• **FastSearch Module:** This module fast searches customers with the smallest arrival time, and consists of the following steps:

**Step 1:** Iterate on each  $i \in N_v^+$  and repeat to generate  $[Rv_i^*]$  exponential variables (i.e., the arrival time of customers) in ascending order (Lines 9-14 in Algorithm 1). Meanwhile, each server  $j$  uses registers  $y_j$  and  $s_j$  to keep track of information of the first arrived customer, where  $y_j$  records the customer's arrival time and  $s_j$  records the index of the queue where the customer comes from (Lines 15-18 in Algorithm 1);

**Step 2:** If there remain any unreserved servers, we increase  $R$  by  $\Delta$  and then repeat Step 1. Otherwise, we stop the FastSearch procedure.

For simplicity, we set the parameter  $\Delta = k$ . In our experiments, we find that the value of  $\Delta$  has a small effect on the performance of FastGM.

• **FastPrune Module:** When all servers  $1, \dots, k$  have been selected by at least one customer among all the released customers. We start the FastPrune module, which mainly consists of the following two steps:

**Step 1.** Compute  $y^* = \max_{j=1, \dots, k} y_j$ .

**Step 2.** For each  $\mathcal{Q}_i, i \in N_v^+$ , we repeat to compute the next customer's arrival time (Lines 24-29 in Algorithm 1). Once a customer's arrival time is larger than  $y^*$ , we

stop releasing customers from queue  $\mathcal{Q}_i$  (Lines 30-32 in Algorithm 1). As we mentioned, variables  $y_j$  and  $s_j$  keep track of information of the first arrived customer. Therefore,  $y_1, \dots, y_k$  and  $s_1, \dots, s_k$  may also be updated by receiving new appointments from newly released customers with arrival times smaller than  $y^*$  at this step (Lines 33-36 in Algorithm 1). Therefore,  $y^*$  may also decrease with the number of released customers, which accelerates the termination of all queues  $\mathcal{Q}_i, i \in N_{\vec{v}}^+$ .

### 2.3 Mergeability

For some applications, the dataset of interest  $\Pi$  is distributed over multiple sites. Suppose that there are  $r$  sites, each site  $i = 1, \dots, r$  holds a sub-dataset  $\Pi_i$ . Each site  $i$  can compute the Gumbel-Max sketch  $(\vec{s}^{(i)}, \vec{y}^{(i)})$  of its set  $N^{(i)}$ , which is the set of objects appearing in  $\Pi_i$ . Here set  $N^{(i)}$  can be easily represented as a weighted vector following the weighted cardinality estimation discussed in Section 1 and its Gumbel-Max sketch  $(\vec{s}^{(i)}, \vec{y}^{(i)})$  can be computed based on our method FastGM. A central site can collect all sites' sketches  $(\vec{s}^{(1)}, \vec{y}^{(1)}), \dots, (\vec{s}^{(r)}, \vec{y}^{(r)})$  and then use them to compute the Gumbel-Max sketch  $(\vec{s}^{\cup}, \vec{y}^{\cup})$  of the union set  $N^{(1)} \cup \dots \cup N^{(r)}$ . For the sketch of union set, each element  $y_i^{\cup}, j = 1, \dots, k$  of  $\vec{y}^{\cup}$  is computed as  $y_j^{\cup} = \min_{i=1, \dots, r} y_j^{(i)}$ , where  $y_j^{(i)}$  is the  $j$ -th element of vector  $\vec{y}^{(i)}$ . Each element  $s_i^{\cup}$  of  $\vec{s}^{\cup}$  is computed as  $s_j^{\cup} = s_j^{(i^*)}$ , where  $i^* = \operatorname{argmin}_{i=1, \dots, r} y_j^{(i)}$  and  $s_j^{(i^*)}$  is the  $j$ -th element of vector  $\vec{s}^{(i^*)}$ . At last, the weighted cardinality of dataset  $\Pi$  can be estimated from the above Gumbel-Max sketch  $(\vec{s}, \vec{y})$ .

### 2.4 Error Analysis

As aforementioned, the parts  $\vec{s}(\vec{v})$  and  $\vec{y}(\vec{v})$  of Gumbel-Max sketches produced by FastGM are equivalent to the sketches proposed in [6] and [8], respectively. Therefore, we have the following error analysis results.

**Theorem 1.** [6] When using the part  $\vec{s}(\vec{v})$  of Gumbel-Max sketch to estimate the probability Jaccard similarity  $\mathcal{J}_{\mathcal{P}}(\vec{u}, \vec{v})$  between  $\vec{u}$  and  $\vec{v}$ , the expectation and variance of estimation  $\hat{\mathcal{J}}_{\mathcal{P}}(\vec{u}, \vec{v})$  are

$$\begin{aligned}\mathbb{E}(\hat{\mathcal{J}}_{\mathcal{P}}(\vec{u}, \vec{v})) &= \mathcal{J}_{\mathcal{P}}(\vec{u}, \vec{v}), \\ \text{Var}(\hat{\mathcal{J}}_{\mathcal{P}}(\vec{u}, \vec{v})) &= \frac{1}{k} \mathcal{J}_{\mathcal{P}}(\vec{u}, \vec{v}) (1 - \mathcal{J}_{\mathcal{P}}(\vec{u}, \vec{v})).\end{aligned}$$

**Theorem 2.** [8] When using the part  $\vec{y}(\vec{v})$  of Gumbel-Max sketch to estimate the weighted cardinality  $c_{\Pi}$  of a sequence  $\Pi$ , the expectation and variance of estimation  $\hat{c}_{\Pi}$  are

$$\begin{aligned}\mathbb{E}(\hat{c}_{\Pi}) &= c_{\Pi}, \\ \text{Var}(\hat{c}_{\Pi}/c_{\Pi}) &= 2/k + \mathcal{O}(1/k^2) \approx 2/k.\end{aligned}$$

### 2.5 Space and Time Complexities

**Space Complexity.** For a non-negative vector  $\vec{v}$  with  $n_{\vec{v}}^+$  positive elements, our method FastGM requires  $k \log k$  bits to store the  $(\pi_{i,1}, \dots, \pi_{i,k})$  of each  $i \in N_{\vec{v}}^+$ , and in summary,  $n_{\vec{v}}^+ k \log k$  bits are desired. In addition, 64k bits are desired for storing  $y_1, \dots, y_k$  (we use 64-bit floating-point registers to record  $y_1, \dots, y_k$ ), and  $k \log n$  bits are required for storing

---

#### Algorithm 1: Pseudo code of our FastGM.

---

```

Input :  $\vec{v} = (v_1, \dots, v_n)$ 
Output:  $\vec{s} = (s_1, \dots, s_k), \vec{y} = (y_1, \dots, y_k)$ 

1  $R \leftarrow 0; k^* \leftarrow k; (y_1, \dots, y_k) \leftarrow (-1, \dots, -1);$ 
2 foreach  $i \in N_{\vec{v}}^+$  do
3    $\lfloor (b_i, z_i) \leftarrow (0, 0); (\pi_{i,1}, \dots, \pi_{i,k}) \leftarrow (1, \dots, k);$ 
   /* The following part is FastSearch */
4 while  $k^* \neq 0$  do
5    $R \leftarrow R + \Delta;$ 
6   foreach  $i \in N_{\vec{v}}^+$  do
7      $R_i \leftarrow \lceil Rv_i^* \rceil;$ 
8     while  $z_i < R_i$  do
9        $z_i \leftarrow z_i + 1;$ 
10      /* Variable  $u \sim \text{UNI}(0, 1)$ . */
11       $u \leftarrow \text{RandUNI}(0, 1, \text{seed} \leftarrow i || z_i);$ 
12       $b_i \leftarrow b_i - \frac{1}{v_i} \left( \frac{\ln u}{k-z_i+1} \right);$ 
      /* RandInt( $z_i, k$ ) returns a number from
      { $z_i, z_i+1, \dots, k$ } at random. */
13       $j \leftarrow \text{RandInt}(z_i, k);$ 
      /* Swap( $\pi_{i,z_i}, \pi_{i,j}$ ) exchanges the values
      of two variables  $\pi_{i,z_i}$  and  $\pi_{i,j}$ . */
14      Swap( $\pi_{i,z_i}, \pi_{i,j});$ 
15       $c \leftarrow \pi_{i,z_i};$ 
16      if  $y_c < 0$  then
17         $\lfloor (y_c, s_c) \leftarrow (b_i, i); k^* \leftarrow k^* - 1;$ 
18      else if  $b_i < y_c$  then
         $\lfloor (y_c, s_c) \leftarrow (b_i, i);$ 

      /* The following part is FastPrune */
19  $j^* \leftarrow \operatorname{argmax}_{j=1, \dots, k} y_j; N \leftarrow N_{\vec{v}}^+;$ 
20 while  $N$  is not empty do
21    $R \leftarrow R + \Delta;$ 
22   foreach  $i \in N$  do
23     while  $z_i < R_i$  do
24        $z_i \leftarrow z_i + 1;$ 
25        $u \leftarrow \text{RandUNI}(0, 1, \text{seed} \leftarrow i || z_i);$ 
26        $b_i \leftarrow b_i - \frac{1}{v_i} \left( \frac{\ln u}{k-z_i+1} \right);$ 
27        $j \leftarrow \text{RandInt}(z_i, k);$ 
28       Swap( $\pi_{i,z_i}, \pi_{i,j});$ 
29        $c \leftarrow \pi_{i,z_i};$ 
30       if  $b_i > y_{j^*}$  then
31          $N \leftarrow N \setminus \{i\};$ 
32         break;
33       if  $b_i < y_{j^*}$  then
34          $\lfloor (y_c, s_c) \leftarrow (b_i, i);$ 
35         if  $c == j^*$  then
             $\lfloor j^* \leftarrow \operatorname{argmax}_{j=1, \dots, k} y_j;$ 

```

---

$s_1, \dots, s_k$ , where  $n$  is the size of the vector. However, the additional memory is released immediately after computing the sketch and is far smaller than the memory for storing the generated sketches of massive vectors (e.g. documents). Therefore, FastGM requires  $n_{\vec{v}}^+ k \log k + 64k + k \log n$  bits when generating a  $k$ -length Gumbel-Max sketch  $\vec{s}(\vec{v}) =$

$(s_1, \dots, s_k)$  and  $\vec{y}(\vec{v}) = (y_1, \dots, y_k)$  of  $\vec{v}$ .

**Time Complexity.** We easily find that a non-negative vector and its normalized vector have the same Gumbel-Max sketch. For simplicity, therefore we analyze the time complexity of our method only for normalized vectors. Let  $\vec{v}^* = (v_1^*, \dots, v_n^*)$  be a normalized and non-negative vector. Define a variable  $\tilde{y}^*$  as:

$$\tilde{y}^* = \max_{j=1, \dots, k} \tilde{y}_j,$$

where  $\tilde{y}_j = \min_{i \in N_{\vec{v}}^+} -\frac{\ln a_{i,j}}{v_i^*}$ ,  $j = 1, \dots, k$ . At the end of our FastPrune procedure, we easily find that each register  $y_j$  used in the procedure equals  $\tilde{y}_j$  and register  $y^*$  equals  $\tilde{y}^*$ . Because  $-\frac{\ln a_{i,j}}{v_i^*} \sim \text{EXP}(v_i^*)$ , we easily find that each  $y_j$  follows the exponential distribution  $\text{EXP}(\sum_{i=1}^n v_i^*)$ , i.e.  $\text{EXP}(1)$ . From [23], we have

$$\begin{aligned}\mathbb{E}(\tilde{y}^*) &= \sum_{m=1}^k \frac{1}{m} \leq \ln k + \gamma, \\ \text{Var}(\tilde{y}^*) &= \sum_{m=1}^k \frac{1}{m^2} < \sum_{m=1}^{\infty} \frac{1}{m^2} = \frac{\pi^2}{6},\end{aligned}$$

where  $\gamma = 1$ . From Chebyshev's inequality, we have

$$P\left(|\tilde{y}^* - \mathbb{E}(\tilde{y}^*)| \geq \alpha \sqrt{\text{Var}(\tilde{y}^*)}\right) \leq \frac{1}{\alpha^2}.$$

Therefore,  $\tilde{y}^* \leq \mathbb{E}(\tilde{y}^*) + \alpha \sqrt{\text{Var}(\tilde{y}^*)}$  happens with a high probability when  $\alpha$  is large. In other words, the random variable  $\tilde{y}^*$  can be upper bounded by  $\mathbb{E}(\tilde{y}^*) + \alpha \sqrt{\text{Var}(\tilde{y}^*)}$  with a high probability. Next, we derive the expectation of  $t_{i,R}$  after the first  $R$  customers have been released. For each queue  $Q_i$ ,  $i \in N_{\vec{v}}^+$ , from Eqs. (4) and (5), we find that the last customer among these first  $R$  customers has a timestamp  $t_{i,R_i}$  with the expectation  $\mathbb{E}(t_{i,R_i} \mid R) \approx \frac{R}{k}$ . When  $R = k(\mathbb{E}(\tilde{y}^*) + \alpha \sqrt{\text{Var}(\tilde{y}^*)}) < k(\ln k + \gamma + \frac{\alpha\pi}{\sqrt{6}})$ , the probability of  $\mathbb{E}(t_{i,R_i}) > \tilde{y}^*$  is almost 1 for large  $\alpha$ , e.g.,  $\alpha > 10$ . Therefore, we find that after the first  $O(k \ln k)$  customers, each queue  $Q_i$  is expected to be early terminated and so we are likely to acquire all the Gumbel-Max variables. We also note that each positive element has to be enumerated once in the FastPrune model. Therefore, the total time complexity of our method FastGM is  $O(k \ln k + n_{\vec{v}}^+)$ .

### 3 OUR METHOD STREAM-FASTGM

We extend our method FastGM to handle data streams. Given a stream  $\Pi$  represented as a sequence of elements  $i \in \{1, \dots, n\}$ . An element  $i$  may occur multiple times in  $\Pi$  and it has a fixed weight  $v_i$ . Our method Stream-FastGM is a fast one-pass algorithm for computing the Gumbel-Max sketch of  $\Pi$ , which reads and processes each element arriving at the stream exactly once.

The pseudo-code of Stream-FastGM is shown in Algorithm 2. Similar to FastGM, for each server  $j = 1, \dots, k$ , we use two registers  $y_j$  and  $s_j$  to record its first customer's arrival time and queue number. In addition, we use  $y^* = \max_{j=1, \dots, k} y_j$  to record the maximum of all  $y_1, \dots, y_k$ . As we mentioned, the FastPrune procedure can be used only after each of the servers has been selected by at least one customer. We use a flag  $FlagFastPrune$  to indicate

---

### Algorithm 2: Pseudo code of our Stream-FastGM.

---

```

Input :data stream  $\Pi$ 
Output:  $\vec{s} = (s_1, \dots, s_k)$ ,  $\vec{y} = (y_1, \dots, y_k)$ 

1  $k^* \leftarrow k$ ;  $j^* \leftarrow 1$ ;  $(y_1, \dots, y_k) \leftarrow (-1, \dots, -1)$ ;
2 foreach element  $i$  in stream  $\Pi$  do
3    $b \leftarrow 0$ ;  $(\pi_1, \dots, \pi_k) \leftarrow (1, \dots, k)$ ;
4   for  $l = 1, \dots, k$  do
5      $u \leftarrow \text{RandUNI}(0, 1, \text{seed} \leftarrow i || l)$ ;
      /*  $v_i$  is the weight of element  $i$ . */
6      $b \leftarrow b - \frac{1}{v_i} \left( \frac{\ln u}{k-l+1} \right)$ ;
7      $j \leftarrow \text{RandInt}(l, k)$ ;
8     Swap( $\pi_l, \pi_j$ );
9      $c \leftarrow \pi_l$ ;
10    if  $FlagFastPrune == \text{False}$  then
11      if  $y_c < 0$  then
12         $(y_c, s_c) \leftarrow (b, i)$ ;
13         $k^* \leftarrow k^* - 1$ ;
14        if  $k == 0$  then
15           $FlagFastPrune \leftarrow \text{True}$ ;
16           $j^* \leftarrow \text{argmax}_{j=1, \dots, k} y_j$ ;
17        else if  $b < y_c$  then
18           $(y_c, s_c) \leftarrow (b, i)$ ;
19    if  $FlagFastPrune == \text{True}$  then
20      if  $b > y_{j^*}$  then
21        break;
22      if  $b < y_c$  then
23         $(y_c, s_c) \leftarrow (b, i)$ ;
24        if  $c == j^*$  then
25           $j^* \leftarrow \text{argmax}_{j=1, \dots, k} y_j$ ;
```

---

whether the FastPrune procedure can be used. For each element  $i$  arriving at stream  $\Pi$ , we repeat to generate random exponential variables in ascending order. When the flag  $FlagFastPrune$  is true and the generated variable has a value larger than  $y^*$ , we stop processing the current element.

## 4 EVALUATION

We evaluate our method FastGM with the state-of-the-art on two tasks: **(Task 1) probability Jaccard similarity estimation** and **(Task 2) weighted cardinality estimation**. All algorithms run on a computer with a Quad-Core Intel(R) Xeon(R) CPU E3-1226 v3 CPU 3.30GHz processor. To demonstrate the reproducibility of the experimental results, we make our source code publicly available<sup>1</sup>.

### 4.1 Datasets

For the task of probability Jaccard similarity estimation, we verify the efficiency of our FastGM in generating Gumbel-Max sketch with different lengths  $k \in \{2^6, 2^7, \dots, 2^{12}\}$  for vectors of length in the range  $n \in \{10^2, 10^3, 10^4\}$ . We generate the weights of synthetic vectors according to the uniform

1. <https://github.com/YuanmingZhang05/FastGM>

TABLE 1: Statistics of used real-world datasets.

Dataset	#Vectors	#Features
Real-sim [24]	72,309	20,958
Rcv1 [25]	20,242	47,236
News20 [26]	19,996	1,355,191
Libimseti [27]	220,970	220,970
Wiki10 [28]	14,146	104,374
MovieLens [29]	69,878	80,555

distribution UNI(0, 1) and the exponential distribution with rate 1 EXP(1). In addition, we also run experiments on six real-world datasets: Real-sim [24], Rcv1 [25], News20 [26], Libimseti [27], Wiki10 [28], and MovieLens [29]. In detail, Real-sim [24], Rcv1 [25], and News20 [26] are datasets of web documents from different sources where each vector represents a document and each entry in the vector refers to the TF-IDF score of a specific word for the document. Libimseti [27] is a dataset of ratings between users on the Czech dating site, where each vector refers to a user and each entry records the user’s rating to another one. Wiki10 [28] is a dataset of tagged Wikipedia articles, where each vector and element represent an article and a tag, respectively. Moreover, the weight of an element indicates how relevant the tag is for the article. MovieLens [29] is a dataset of movie ratings, where each vector is a user and each entry in the vector is that user’s rating for a specific movie. The statistics of all the above datasets are summarized in Table 1.

As for the task of weighted cardinality estimation, we follow the experimental settings in [8]. We conduct experiments on both synthetic datasets and a simulated scenario obtained from real-world problems. In the later Section 4.5, we detail them. In addition, we also design a data steaming setting to demonstrate the mergeability of our Gumbel-Max sketch and the performance of our Stream-FastGM. Specifically, we generate a set of elements arriving in a streaming fashion.

## 4.2 Baseline

To demonstrate the improvement of our FastGM over the conference version of FastGM (in short, **FastGM-c**), we also apply FastGM-c as a baseline in efficiency experiments. For task 1, **probability Jaccard similarity estimation**, we compare our method FastGM with  $\mathcal{P}$ -MinHash [6]. To highlight the efficiency of FastGM, we further compare FastGM with the state-of-the-art weighted Jaccard similarity estimation method, BagMinHash [30], which is used for estimating weighted Jaccard similarity  $\mathcal{J}_W$ . The weighted Jaccard similarity  $\mathcal{J}_W$  that BagMinHash aims to estimate is an alternative similarity metric to the probability Jaccard similarity  $\mathcal{J}_P$  we focused on in this paper. Experiments and theoretical analysis in [6] have shown that weighted Jaccard similarity  $\mathcal{J}_W$  and probability Jaccard similarity  $\mathcal{J}_P$  usually have similar performance on many applications such as fast searching similar set. Notice that BagMinHash estimates a different similarity metric and thus we only show its results on efficiency. For task 2, **weighted cardinality estimation**, we compare our method with Lemiesz’s sketch [8].

## 4.3 Metric

For both tasks of probability Jaccard similarity estimation and weighted cardinality estimation, we use the running

time and root mean square error (RMSE) to measure our method’s efficiency and effectiveness, respectively. In detail, we measure the RMSEs of probability Jaccard similarity estimation  $\hat{J}$  and weighted cardinality estimation  $\hat{c}$  with respect to their true values  $J$  and  $c$  as:

$$\text{RMSE}(\hat{J}) = \sqrt{\mathbb{E}((\hat{J} - J)^2)}, \quad \text{RMSE}(\hat{c}) = \sqrt{\mathbb{E}((\hat{c} - c)^2)}.$$

All experimental results are empirically computed from 1,000 independent runs by default.

## 4.4 Probability Jaccard Similarity Estimation

We conduct experiments on both synthetic and real-world datasets for the task of probability Jaccard similarity estimation. Specially, we first use synthetic weighted vectors to evaluate the performance of FastGM for vectors with different dimensions. Then, we show results on 6 real-world datasets.

**Results on synthetic vectors.** We first conduct experiments on weighted vectors with uniform-distribution weights. Without loss of generality, we let  $n_v^+ = n$  for each vector, i.e., all elements of each vector are positive. As shown in Fig. 4 (a), (b) and (c), when  $n = 10^3$ , FastGM is 13 and 22 times faster than BagMinHash and  $\mathcal{P}$ -MinHash respectively. As  $n$  increases to  $10^4$ , the improvement becomes 8 and 125 times respectively. Especially, the sketching time of our method is around 0.02 seconds when  $n = 10^4$  and  $k = 2^{12}$ , while BagMinHash and  $\mathcal{P}$ -MinHash take over 0.15 and 2.5 seconds for sketching respectively. In Fig. 4 (d), (e), and (f), we show the running time of all competitors for different  $n$ . Our method FastGM is 13 to 100 times faster than  $\mathcal{P}$ -MinHash for different  $n$ . Compared with BagMinHash, FastGM is about 60 times faster when  $n = 1,000$ , and is comparable as  $n$  increases to 100,000. It indicates that our method FastGM significantly outperforms BagMinHash for vectors having less than 100,000 positive elements, which are prevalent in real-world datasets. As shown in Fig. 4, our FastGM is consistently faster than FastGM-c, when  $n = 100$  and  $n = 1,000$  FastGM is around 1.2 and 1.5 times faster than FastGM-c, respectively. Results are similar when the weights of synthetic vectors follow the exponential distribution EXP(1), thus we omit them here.

**Results on real-world datasets.** Next, we show results on the real-world datasets in Table 1. We report the sketching time of all algorithms in Fig. 5. We see that our method outperforms  $\mathcal{P}$ -MinHash and BagMinHash on all the datasets. FastGM is consistently faster than FastGM-c, especially on datasets Rcv1, Libimseti, and MovieLens, FastGM is 4 times faster than FastGM-c on average. On sparse datasets such as Real-sim, Rcv1, Wiki10, and MovieLens, FastGM is about 8 and 12 times faster than  $\mathcal{P}$ -MinHash and BagMinHash respectively. BagMinHash is even slower than  $\mathcal{P}$ -MinHash on these datasets. On dataset News20 we note that FastGM is 26 times faster than  $\mathcal{P}$ -MinHash.

Fig. 6 shows the estimation error of FastGM and  $\mathcal{P}$ -MinHash on datasets Real-sim and MovieLens. Due to a large number of vector pairs, we here randomly select 100,000 pairs of vectors from each dataset and report the average RMSE. We note that both algorithms give similar accuracy, which is coincident with our analysis. We omit similar results on other datasets.

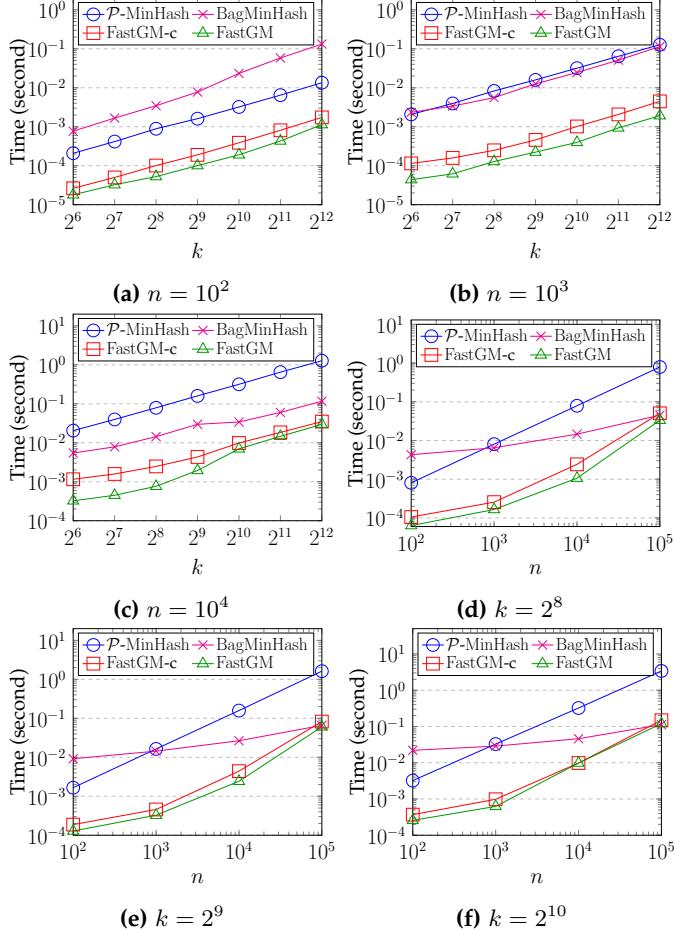


Fig. 4: (Task 1) The efficiency of FastGM compared with  $\mathcal{P}$ -MinHash, BagMinHash, and the conference version of FastGM (FastGM-c) on synthetic vectors, where each element in the vector is randomly selected from  $UNI(0,1)$ .

#### 4.5 Weighted Cardinality Estimation

In this task, we compare our FastGM with Lemiesz's sketch on both effectiveness and efficiency. The experimental results show that our FastGM sketch has the same accuracy as Lemiesz's sketch and is orders of magnitude faster in producing sketches. In the following, we detail the experiments on both synthetic datasets and a simulated scenario obtained from real-world problems in wireless sensor networks.

**Results on synthetic datasets.** To evaluate the weighted cardinality estimation accuracy of our method, we generate a variety of data examples with different cardinalities. We vary the number of elements in the data examples and generate the weights of elements according to the uniform distribution  $UNI(0, 1)$  and the normal distribution  $N(1, 0.1)$ . We report the RMSEs between the true cardinalities  $c$  of data examples and estimations  $\hat{c}$  from the sketches. As shown in Fig. 7, our FastGM sketch has the same performance as Lemiesz's sketch on each dataset, because the  $\vec{y}$  part of FastGM and Lemiesz's sketch have the same results but are computed in different ways. The efficiency of generating the two sketches is totally the same as the results reported in Fig. 4, where Lemiesz's sketch has the same running time as  $\mathcal{P}$ -MinHash. Therefore, in terms of efficiency, our FastGM

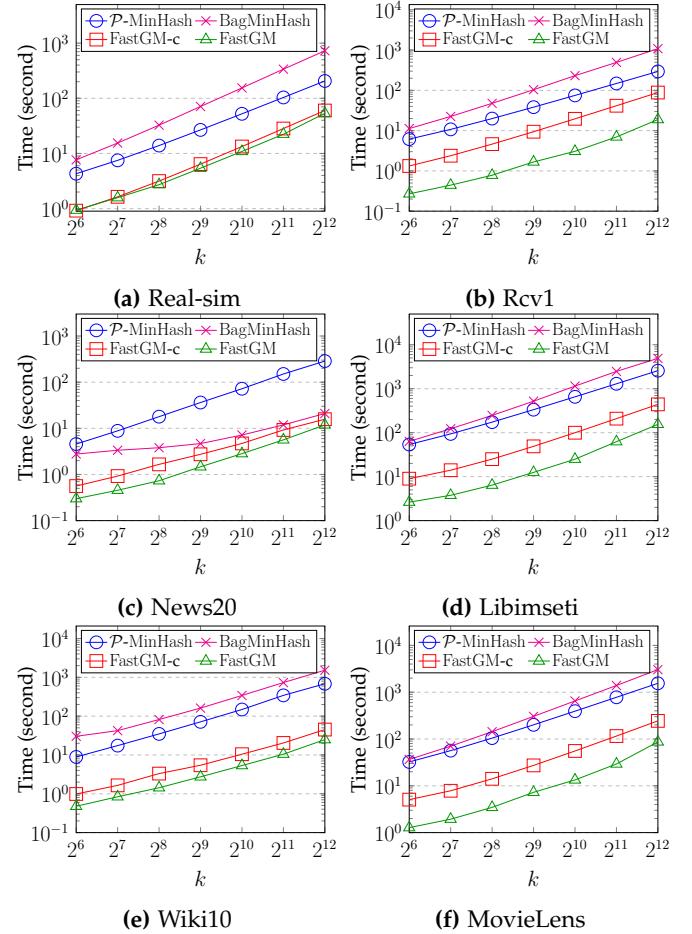


Fig. 5: (Task 1) Efficiency of FastGM compared with  $\mathcal{P}$ -MinHash, BagMinHash, and the conference version of FastGM (FastGM-c) for different  $k$  on real-world datasets.

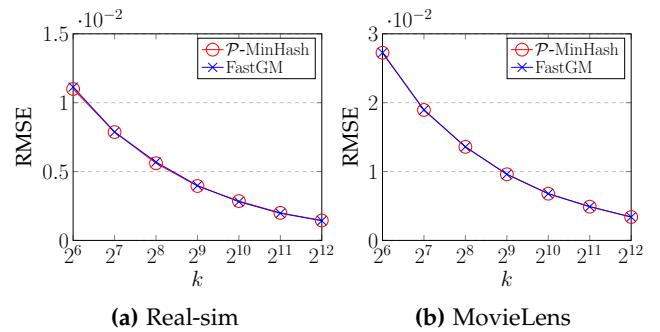


Fig. 6: (Task 1) The accuracy of FastGM compared with  $\mathcal{P}$ -MinHash for different  $k$ .

sketch outperforms Lemiesz's sketch by as much as FastGM outperforms  $\mathcal{P}$ -MinHash. Hence we omit similar results. Moreover, in Fig. 8a we show the running time of computing the sketches by using our Stream-FastGM compared with Lemiesz's sketch, and our Stream-FastGM is 23 times faster than Lemiesz's sketch on average when  $n = 1,000$ . In Fig. 8b we report the running time of generating the sketches of length  $k = 1024$  for data examples with different objects  $n$ , our Stream-FastGM is about 120 times faster than Lemiesz's sketch at  $n = 10^6$ .

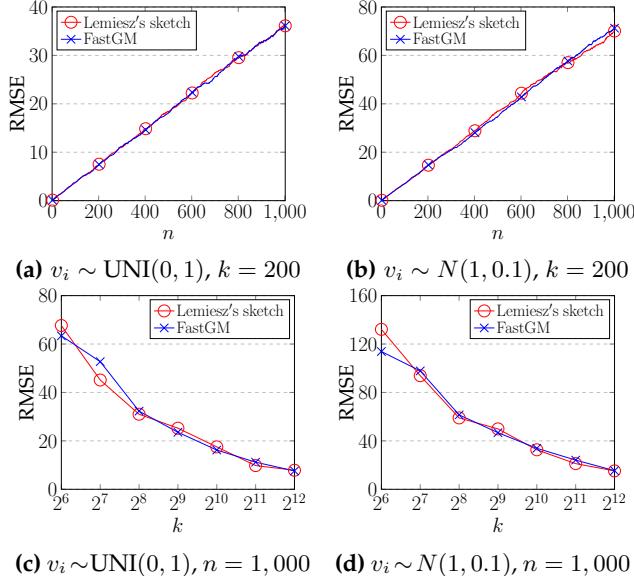


Fig. 7: (Task 2) The weighted cardinality estimation errors on synthetic datasets, where the lengths  $k$  of both Lemiesz's and FastGM sketches are the same. For each data example, we generate  $\vec{v}^{\Pi}$  with  $n$  objects of which weights  $\vec{v}_j^{\Pi}$  are derived according to the uniform distribution  $\text{UNI}(0, 1)$  and the normal distribution  $N(1, 0.1)$  respectively.

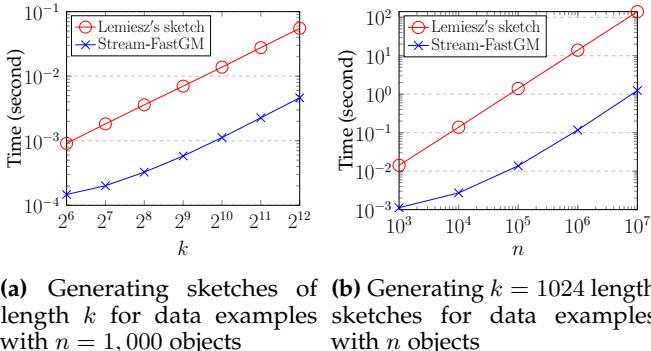


Fig. 8: (Task 2) Average running time of Stream-FastGM and Lemiesz's sketch on synthetic data.

**Results on the simulated scenario.** Following the experimental setting in [8], we conduct experiments on simulated multi-hop wireless sensor networks where sensors use a braid chain strategy to guarantee the robustness of communication. In Fig. 9, we show the topology of simulated networks. A braided chain consists of two sequences of nodes (sensors)  $\mathbb{S}^A = [s_1^A, \dots, s_d^A]$  and  $\mathbb{S}^B = [s_1^B, \dots, s_d^B]$ . Nodes with the same position in the sequences, such as  $s_1^A$  and  $s_1^B$ , are considered as nodes in the same layer. Because the transfer path (edges in the network topology) is unstable. To guarantee the transfer of traffic packets, the node in the previous layer **redundantly** transfers traffic packets to all nodes in the next layer. Specifically, the transfer path between nodes in the same sensors sequence and between different sensors sequences (e.g., the edge between  $s_1^A$  and  $s_2^A$ , the edge between  $s_1^A$  and  $s_2^B$ ) work well in chance  $p_1$  and  $p_2$ , respectively. For example, a traffic packet in node  $s_1^A$

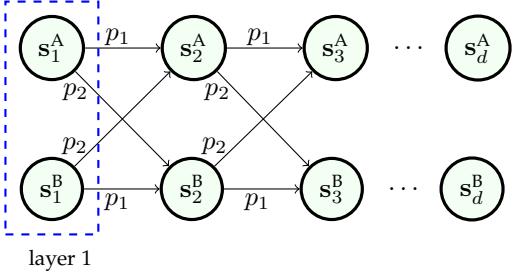
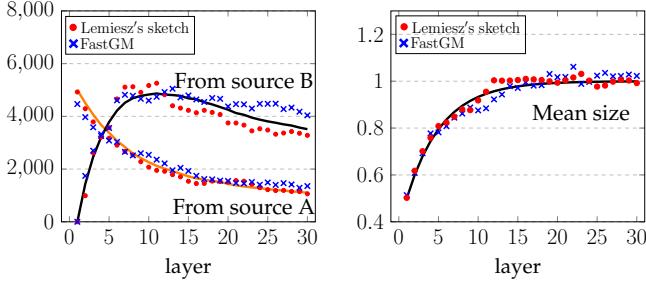


Fig. 9: An example to show a simulated sensor network with  $d$  layers using the braid chain strategy to transfer traffic packets, where each node and each edge represent a sensor in the network and a traffic transfer path, respectively. The  $p_1$  and  $p_2$  denote the probability of a successful transmission between two nodes.

successfully sent to  $s_2^A$  in chance  $p_1$ , meanwhile a copy of this traffic packet also has  $p_2$  chance to be successfully sent to node  $s_2^B$ . Note that  $p_1$  does not necessarily equal  $1 - p_2$ .

In the experiment setting, the first node in each sequence is considered as the source that generates traffic packet  $i$  with size  $v_i$  in sequence. After each source  $s_1$  generates a sequence  $\Pi$  consisting of  $n$  traffic packets, we have a vector  $\vec{v}^{\Pi}$  of length  $n$  from this sequence  $\Pi$ . In our experiment, we follow the setting in [8] and set  $p_1 = 0.9$ ,  $p_2 = 0.1$ ,  $d = 30$ ,  $n = 10,000$  and the sizes of packets  $v_i$  are generated according to a Beta distribution with parameters  $\alpha = \beta = 5$ . Take a node in the second layer  $s_2^A$  as an example, traffic packet sequence received by  $s_2^A$  is a mixture of some traffic packets in sequences  $\Pi_{s_1^A}$  and  $\Pi_{s_1^B}$  of both sources  $s_1^A$  and  $s_1^B$ . For the traffic packet sequence that passes through each node in the network, we build a sketch for it and use the sketch to estimate the total size of **distinct** packets appearing in this sequence. In this case, the weighted cardinality of the sequence represents the sum of distinct packets' sizes in the sequence. The reasons to build a sketch rather than simply use a counter are: 1) the traffic packet sequences passing through nodes in layers behind the second layer contain repetitive traffic packets, which causes the double-counting problem; 2) aggregating sketches rather than packets will not cause an explosion of packets even when the network follows a flooding strategy while guaranteeing the certain accuracy of network communication [8]. More than that, based on the sketches we are able to obtain more useful information about the network and we conduct the following experiments to demonstrate this. Ground truth results are shown as solid lines, and estimations obtained from sketches are shown as dashed lines. For clarity, we use the symbol  $N_{\mathbf{s}}$  to represent the weighted set of packets that occurred in traffic packet sequence  $\Pi_{\mathbf{s}}$  of a node rather than  $N_{\Pi_{\mathbf{s}}}$ .

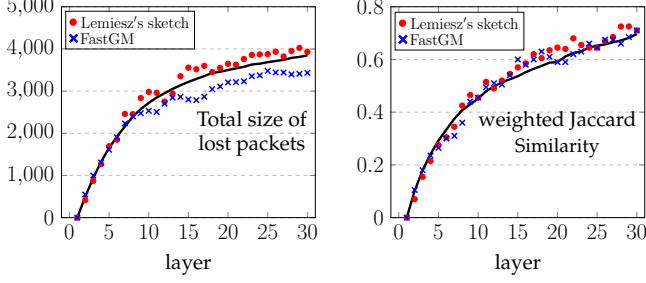
In Fig. 10a, black and orange lines represent the size of packets from source  $s_1^A$  and  $s_1^B$  respectively. The size of distinct packets received by a node  $s_{\ell}^A$ ,  $1 \leq \ell \leq d$  is defined as  $|N_{s_{\ell}^A}|_w = \sum_{i \in N_{s_{\ell}^A}} v_i$ , where  $i$  represents a traffic packet and  $v_i$  is the size of packet  $i$ . For a node  $s_{\ell}^A$  in sequence  $\mathbb{S}^A$ , the sizes of distinct packets sent from source  $s_1^A$  and source  $s_1^B$  are computed as  $|N_{s_1^A} \cap N_{s_{\ell}^A}|_w$  and  $|N_{s_1^B} \cap N_{s_{\ell}^A}|_w$ , respectively. In Fig. 10b, we show the results of estimating the average



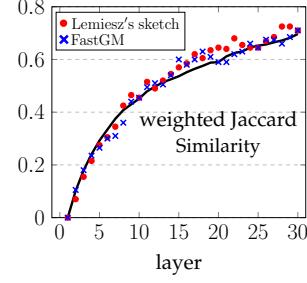
(a) The total size of distinct packets from sources A and B at node  $s_{\ell}^A$  in each layer.



(b) The mean size of distinct packets passing through node  $s_{\ell}^A$  in each layer.



(c) The total size of lost packets sent from source  $s_1^A$  in each layer.



(d) The weighted Jaccard similarity between two nodes in each layer.

Fig. 10: (Task 2) In Fig. 10a-10d solid lines represent ground truth results, red circle and blue cross points respectively represent the estimation results based on Lemiesz's sketches and FastGM sketches with length  $k = 200$ . All results are obtained from a simulated sensor network with  $d = 30$  layers where each data source generates  $n = 10,000$  packets.

mixture of some packets in traffic packets from both sources  $s_1^A$  and  $s_1^B$ . Therefore, we can use the weighted Jaccard similarity  $\mathcal{J}_W$  between traffic packets sets  $N_{s_{\ell}^A}$  and  $N_{s_{\ell}^B}$ , i.e.,

$$\mathcal{J}_W(N_{s_{\ell}^A}, N_{s_{\ell}^B}) = \frac{|N_{s_{\ell}^A} \cap N_{s_{\ell}^B}|_w}{|N_{s_{\ell}^A} \cup N_{s_{\ell}^B}|_w},$$

to measure the proportion of the total size of identical packets passing through the two nodes. We show the results in Fig. 10d. Given the Gumbel-Max sketches of two arbitrary sets  $N_A$  and  $N_B$ , Lemiesz [8] proposed a series of methods to estimate the weighted cardinality of both union and intersection  $|N_A \cup N_B|_w$  and  $|N_A \cap N_B|_w$ , the weighted Jaccard similarity  $\mathcal{J}_W(N_A, N_B)$ , the weighted cardinality of relative complement  $|N_A \setminus N_B|_w$  from these sketches, and these methods can be extended to multiple sets. In our experiments, we use the same methods to compute the total size of packets from sources A and B at each sensor node, the total size of lost packets at each sensor node, and the weighted Jaccard similarity between two nodes in each layer. As we analyzed above, the  $\vec{y}$  part of FastGM sketch is the same as Lemiesz's sketch, so they have the same performance in each experiment.

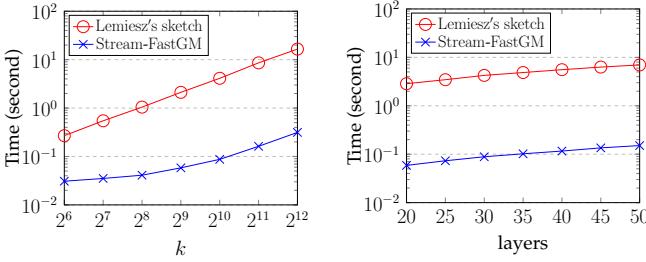
To demonstrate the efficiency of our Stream-FastGM, in Fig. 11a we report the running time of generating sketches with different lengths  $k$ . When  $k = 2048$ , our Stream-FastGM is 52 times faster than Lemiesz's sketch, and the results show that our Stream-FastGM gets faster than Lemiesz's sketch when  $k$  gets larger. We also conduct experiments on simulated sensor networks with different depths of layers, as shown in Fig. 11b, our Stream-FastGM is 47 times faster than Lemiesz's sketch on average.

## 5 RELATED WORK

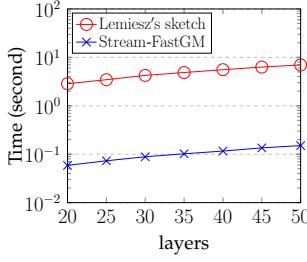
### 5.1 Jaccard Similarity Estimation

Broder et al. [14] proposed the first sketch method *MinHash* to compute the Jaccard similarity of two sets (or binary vectors). MinHash builds a sketch consisting of  $k$  registers for each set. Each register uses a hash function to keep track of the set's element with the minimal hash value. To further improve the performance of MinHash, [31], [12], [32] developed several memory-efficient methods. Li et al. [33] proposed *One Permutation Hash* (OPH) to reduce the time complexity of processing each element from  $O(k)$  to  $O(1)$  but this method may exhibit large estimation errors because of the empty buckets. To solve this problem, several densification methods [17], [18], [19], [34] were developed to set the registers of empty buckets according to the values of non-empty buckets' registers.

Besides binary vectors, a variety of methods have also been developed to estimate generalized Jaccard similarity on weighted vectors. For vectors consisting of only nonnegative integer weights, Haveliwala et al. [35] proposed to add a corresponding number of replications of each element in order to apply the conventional MinHash. To handle more general real weights, Haeupler et al. [36] proposed to generate another additional replication with a probability that equals the floating part of an element's weight. These two algorithms are computationally intensive when computing hash values of massive replications for elements with large weights. To solve this problem, [37], [38] proposed to compute hash values only for a few necessary replications



(a) Generating sketches of different lengths  $k$  on each node of simulated sensor networks with  $d = 30$  layers.



(b) Generating sketches of length  $k = 1024$  for nodes of simulated networks with different depth of layers.

Fig. 11: (Task 2) Average running time of Stream-FastGM and Lemiesz's sketch on simulated sensor networks.

size of distinct packets on each node in sensor sequence  $S^A$ . In Fig. 10c, we use the sketches to estimate the total size of lost packets from source  $s_1^A$  in each layer of the braided chain. The set of lost packets from source  $s_1^A$  in a layer  $N_{L_{\ell}^A}$  can be obtained from  $N_{L_{\ell}^A} = N_{s_1^A} \setminus (N_{s_{\ell}^A} \cup N_{s_{\ell}^B})$ , where  $N_{s_{\ell}^A} \cup N_{s_{\ell}^B}$  denotes the set of the distinct packets passing through at least one of nodes  $s_{\ell}^A$  and  $s_{\ell}^B$ , and set  $N_{s_1^A} \setminus (N_{s_{\ell}^A} \cup N_{s_{\ell}^B})$  represents the set of packets generated by source  $s_1^A$  but not received by node  $s_{\ell}^A$  or node  $s_{\ell}^B$ . Note that each node in a layer receives a

(i.e., "active indices"). ICWS [39] and its variations such as 0-bit CWS [40], CCWS [41], PCWS [42], I<sup>2</sup>CWS [43] were proposed to improve the performance of CWS [38]. The CWS algorithm and its variants all have the time complexity of  $O(n^+k)$ , where  $n^+$  is the number of elements with positive weights. Recently, Otmar [30] proposed another efficient algorithm *BagMinHash* for handling high-dimensional vectors. *BagMinHash* is faster than ICWS when the vector has a large number of positive elements, e.g.,  $n^+ > 1,000$ , which may not hold for many real-world datasets. The above methods all estimate the weighted Jaccard similarity. Ryan et al. [6] proposed a Gumbel-Max Trick based sketching method,  $\mathcal{P}$ -MinHash, to estimate another novel Jaccard similarity metric, *probability Jaccard similarity*  $\mathcal{J}_{\mathcal{P}}$ . They also demonstrated that the proposed probability Jaccard similarity  $\mathcal{J}_{\mathcal{P}}$  is scale-invariant and more sensitive to changes in vectors. However, the time complexity of  $\mathcal{P}$ -MinHash processing a weighted vector is  $O(n^+k)$ , which is infeasible for high-dimensional vectors.

## 5.2 Cardinality Estimation

The regular problem of cardinality estimation aims to compute the number of distinct elements in the set of interest, which is typically given as a sequence containing duplicated elements [44]. To address this problem, a number of sketch methods such as LPC [45], LogLog [46], HyperLogLog [47], RoughEstimator [48], HLL-TailCut+ [49], and HLL++ [50] build a sketch consisting of  $m$  bits/counters for a set. The sketch is small (e.g.,  $m = 1,000$ ) and can be efficiently updated, which handles each element with few operations. The generated sketch is finally used to estimate the set's cardinality. In addition, [51], [52] exploit martingale estimation and maximum likelihood estimation to improve the estimation accuracy of the above methods. For some applications, there may exist many sets of which sizes vary significantly. To reduce the memory cost of building a sketch for each set, a number of works [53], [54], [55], [56], [57], [58], [59] propose to implement  $m$  independent hash functions to randomly map each sketch into a large shared bit/counter array, where each sketch can be rebuilt by randomly sampling  $m$  bits/counters from the shared array.

Recently, [60], [8] generalized the problem of cardinality estimation to a weighted version, where each element is associated with a fixed positive weight. The goal of weighted cardinality estimation is to estimate the total sum of weights for all distinct elements in the stream of interest. The drawback of the sketch methods in [60], [8] is their high computational costs.

## 6 CONCLUSION

In this paper, we develop an efficient algorithm *FastGM* to compute a non-negative vector's  $k$ -length Gumbel-Max sketch. We propose a novel model, *Queuing model with  $k$ -servers and  $n$ -queues*, to model the procedure of computing the Gumbel-Max sketch in a brief and practical way. Based on the proposed model, we optimize the procedure of generating  $k$  random variables  $-\frac{\ln a_{i,j}}{v_i}$  of an element  $v_i$  in a vector. We theoretically prove that our *FastGM* reduces the time complexity of generating a  $k$ -length Gumbel-Max

sketch from  $O(n^+k)$  to  $O(k \ln k + n^+)$ , where  $n^+$  is the number of the vector's positive elements. We conduct two tasks probability Jaccard similarity estimation and weighted cardinality estimation to demonstrate the efficiency and effectiveness of *FastGM*. Experimental results show that our *FastGM* is around 10 times faster than state-of-the-art methods, without losing any estimation accuracy.

## ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their comments and suggestions. This work was supported in part by National Natural Science Foundation of China (U22B2019, 62272372, 61902305), MoE-CMCC "Artificial Intelligence" Project (MCM20190701).

## REFERENCES

- [1] Y. Qi, P. Wang, Y. Zhang, J. Zhao, G. Tian, and X. Guan, "Fast generating A large number of gumbel-max variables," in *WWW*, 2020, pp. 796–807.
- [2] R. D. Luce, *Individual choice behavior: A theoretical analysis*. Courier Corporation, 1959.
- [3] D. Yang, B. Li, and P. Cudré-Mauroux, "Poisketch: Semantic place labeling over user activity streams," Université de Fribourg, Tech. Rep., 2016.
- [4] D. Yang, B. Li, L. Rettig, and P. Cudré-Mauroux, "Histosketch: Fast similarity-preserving sketching of streaming histograms with concept drift," in *IEEE ICDM*. IEEE, 2017, pp. 545–554.
- [5] ———, "D2 histosketch: discriminative and dynamic similarity-preserving sketching of streaming histograms," *IEEE TKDE*, pp. 1–1, 2018.
- [6] R. Moulton and Y. Jiang, "Maximally consistent sampling and the jaccard index of probability distributions," *arXiv preprint arXiv:1809.04052*, 2018.
- [7] D. Yang, P. Rosso, B. Li, and P. Cudré-Mauroux, "Nodesketch: Highly-efficient graph embeddings via recursive sketching," in *SIGKDD*, 2019.
- [8] J. Lemiesz, "On the algebra of data sketches," *Proc. VLDB Endow.*, vol. 14, no. 9, pp. 1655–1667, may 2021.
- [9] M. Henzinger, "Finding near-duplicate web pages: a large-scale evaluation of algorithms," in *SIGIR*. ACM, 2006, pp. 284–291.
- [10] G. S. Manku, A. Jain, and A. Das Sarma, "Detecting near-duplicates for web crawling," in *WWW*. ACM, 2007, pp. 141–150.
- [11] Y. Bachrach, E. Porat, and J. S. Rosenschein, "Sketching techniques for collaborative filtering," in *IJCAI*, 2009.
- [12] M. Mitzenmacher, R. Pagh, and N. Pham, "Efficient estimation for high similarities using odd sketches," in *WWW*, 2014, pp. 109–118.
- [13] A. Gionis, P. Indyk, and R. Motwani, "Similarity search in high dimensions via hashing," in *PVLDB*, 1999, pp. 518–529.
- [14] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher, "Min-wise independent permutations," *J. Comput. Syst. Sci.*, vol. 60, no. 3, pp. 630–659, Jun. 2000.
- [15] M. S. Charikar, "Similarity estimation techniques from rounding algorithms," in *STOC*, 2002, pp. 380–388.
- [16] E. Buchnik, E. Cohen, A. Hasidim, and Y. Matias, "Self-similar epochs: Value in arrangement," in *ICML*, 2019, pp. 841–850.
- [17] A. Shrivastava and P. Li, "Improved densification of one permutation hashing," in *UAI*, 2014, pp. 732–741.
- [18] ———, "Densifying one permutation hashing via rotation for fast near neighbor search," in *ICML*, 2014, pp. 557–565.
- [19] A. Shrivastava, "Optimal densification for fast and accurate minwise hashing," in *ICML*, 2017, pp. 3154–3163.
- [20] R. Motwani and P. Raghavan, "3.6 the coupon collector's problem, randomized algorithms," 1995.
- [21] A. Rényi, "On the theory of order statistics," *Acta Mathematica Hungarica*, 1953.
- [22] R. A. Fisher and F. Yates, *Statistical tables for biological, agricultural and medical research*. Hafner Publishing Company, 1953.
- [23] "Variance of the maximum of  $n$  independent exponentials," <https://math.stackexchange.com/questions/3175307/variance-of-the-maximum-of-n-independent-exponentials>.

- [24] W. Wei, B. Li, C. Ling, and C. Zhang, "Consistent weighted sampling made more practical," in *WWW*, 2017, pp. 1035–1043.
- [25] D. D. Lewis, Y. Yang, T. G. Rose, and L. Fan, "Rcv1: A new benchmark collection for text categorization research," *JMLR*, vol. 5, no. 2, pp. 361–397, 2004.
- [26] S. S. Keerthi and D. DeCoste, "A modified finite newton method for fast solution of large scale linear svms," *Journal of Machine Learning Research*, no. 6, pp. 341–361, 2005.
- [27] "Libimseti.cz network dataset – KONECT," Apr. 2017. [Online]. Available: <http://konect.uni-koblenz.de/networks/libimseti>
- [28] A. Zubiaga, "Enhancing navigation on wikipedia with social tags," in *Wikimania*, 2009.
- [29] "Movielens 10m network dataset – KONECT," Apr. 2017. [Online]. Available: [http://konect.uni-koblenz.de/networks/movielens-10m\\_rating](http://konect.uni-koblenz.de/networks/movielens-10m_rating)
- [30] O. Ertl, "Bagminhash-minwise hashing algorithm for weighted sets," in *SIGKDD*. ACM, 2018, pp. 1368–1377.
- [31] P. Li and A. C. König, "b-bit minwise hashing," in *WWW*, 2010, pp. 671–680.
- [32] P. Wang, Y. Qi, Y. Zhang, Q. Zhai, C. Wang, J. C. S. Lui, and X. Guan, "A memory-efficient sketch method for estimating high similarities in streaming sets," in *SIGKDD*, 2019, pp. 25–33.
- [33] P. Li, A. B. Owen, and C. Zhang, "One permutation hashing," in *NIPS*, 2012, pp. 3122–3130.
- [34] S. Dahlgaard, M. B. T. Knudsen, and M. Thorup, "Fast similarity sketching," in *FOCS*. IEEE, 2017, pp. 663–671.
- [35] T. Haveliwala, A. Gionis, and P. Indyk, "Scalable techniques for clustering the web," 2000.
- [36] B. Haeupler, M. Manasse, and K. Talwar, "Consistent weighted sampling made fast, small, and easy," *arXiv preprint arXiv:1410.4266*, 2014.
- [37] S. Gollapudi and R. Panigrahy, "Exploiting asymmetry in hierarchical topic extraction," in *CIKM*. ACM, 2006, pp. 475–482.
- [38] M. Manasse, F. McSherry, and K. Talwar, "Consistent weighted sampling," Tech. Rep., June 2010.
- [39] S. Ioffe, "Improved consistent sampling, weighted minhash and L1 sketching," in *ICDM*, 2010, pp. 246–255.
- [40] P. Li, "0-bit consistent weighted sampling," in *SIGKDD*, 2015, pp. 665–674.
- [41] W. Wu, B. Li, L. Chen, and C. Zhang, "Canonical consistent weighted sampling for real-value weighted min-hash," in *ICDM*, 2016, pp. 1287–1292.
- [42] ——, "Consistent weighted sampling made more practical," in *WWW*, 2017, pp. 1035–1043.
- [43] W. Wu, B. Li, L. Chen, C. Zhang, and P. Yu, "Improved consistent weighted sampling revisited," *IEEE TKDE*, 2018.
- [44] H. Lan, Z. Bao, and Y. Peng, "A survey on advancing the dbms query optimizer: Cardinality estimation, cost model, and plan enumeration," *Data Science and Engineering*, vol. 6, no. 1, pp. 86–101, 2021.
- [45] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *TODS*, vol. 15, no. 2, pp. 208–229, 1990.
- [46] M. Durand and P. Flajolet, "Loglog counting of large cardinalities," in *ESA*, 2003, pp. 605–617.
- [47] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm," in *DMTCS*, 2007, pp. 137–156.
- [48] D. M. Kane, J. Nelson, and D. P. Woodruff, "An optimal algorithm for the distinct elements problem," in *PODS*, 2010, pp. 41–52.
- [49] Q. Xiao, Y. Zhou, and S. Chen, "Better with fewer bits: Improving the performance of cardinality estimation of large data streams," in *INFOCOM*, 2017, pp. 1–9.
- [50] S. Heule, M. Nunkesser, and A. Hall, "Hyperloglog in practice: Algorithmic engineering of a state of the art cardinality estimation algorithm," in *EDBT*, 2013, pp. 683–692.
- [51] D. Ting, "Streamed approximate counting of distinct elements: Beating optimal batch methods," in *SIGKDD*, 2014, pp. 442–451.
- [52] O. Ertl, "New cardinality estimation algorithms for hyperloglog sketches," *arXiv preprint arXiv:1702.01284*, 2017.
- [53] Q. Zhao, A. Kumar, and J. J. Xu, "Joint data streaming and sampling techniques for detection of super sources and destinations," in *IMC*, 2005, pp. 77–90.
- [54] M. Yoon, T. Li, S. Chen, and J.-K. Peir, "Fit a spread estimator in small memory," in *INFOCOM*, 2009, pp. 504–512.
- [55] P. Wang, X. Guan, T. Qin, and Q. Huang, "A data streaming method for monitoring host connection degrees of high-speed links," *TIFS*, vol. 6, no. 3, pp. 1086–1098, 2011.
- [56] Q. Xiao, S. Chen, M. Chen, and Y. Ling, "Hyper-compact virtual estimators for big network data based on register sharing," in *SIGMETRICS*, 2015, pp. 417–428.
- [57] P. Wang, P. Jia, X. Zhang, J. Tao, X. Guan, and D. Towsley, "Utilizing dynamic properties of sharing bits and registers to estimate user cardinalities over time," in *ICDE*, 2019, pp. 1094–1105.
- [58] P. Jia, P. Wang, Y. Zhang, X. Zhang, J. Tao, J. Ding, X. Guan, and D. Towsley, "Accurately estimating user cardinalities and detecting super spreaders over time," *TKDE*, vol. 34, no. 1, pp. 92–106, 2020.
- [59] D. Ting, "Approximate distinct counts for billions of datasets," in *SIGMOD*, 2019, pp. 69–86.
- [60] R. Cohen, L. Katzir, and A. Yehezkel, "A unified scheme for generalizing cardinality estimators to sum aggregation," *Inf. Process. Lett.*, vol. 115, no. 2, pp. 336–342, 2015.

**Yuanming Zhang** received a B.S. degree in automation from Chongqing University, Chongqing, China, in 2017. He is currently working toward a graduate degree at the MOE Key Laboratory for Intelligent Networks and Network Security, Xi'an Jiaotong University, Xi'an, China. His research interests include anomaly detection, encrypted traffic analysis, and Internet traffic measurement and modeling.



**Pinghui Wang** (Senior Member, IEEE) is currently a Professor with the MOE Key Laboratory for Intelligent Networks and Network Security, Xi'an Jiaotong University, Xi'an, China, and also with the Shenzhen Research Institute, Xi'an Jiaotong University, Shenzhen, China. His research interests include internet traffic measurement and modeling, traffic classification, abnormal detection, and online social network measurement.



**Yiyan Qi** received a B.S. in automation engineering and a Ph.D. degree in automatic control from Xi'an Jiaotong University, Xi'an, China, in 2014 and 2021 respectively. He is currently a Researcher at the International Digital Economy Academy (IDEA). Prior to joining IDEA, he was working at Tencent. His current research interests include abnormal detection, graph mining and embedding, and recommender systems.



**Kuankuan Cheng** is currently working toward an undergraduate degree at Xi'an Jiaotong University, Xi'an, China. His research interests include streaming data processing and encrypted traffic analysis.





**Junzhou Zhao** received B.S. (2008) and Ph.D. (2015) degrees in control science and engineering from Xi'an Jiaotong University. He is currently an associate professor at the School of Cyber Science and Engineering, Xi'an Jiaotong University. His research interests include graph data mining and streaming data processing.



**Guangjian Tian** received a Ph.D. degree in computer science and technology from Northwestern Polytechnical University, Xi'an, China, in 2006. He is currently a principal researcher in Huawei Noah's Ark Lab. Before that, he was a postdoctoral research fellow with the Department of Electronic and Information Engineering, The Hong Kong Polytechnic University, Hong Kong. His research interests include temporal data analysis, deep learning, and data mining with a specific focus on different industry applications.



**Xiaohong Guan** (Fellow, IEEE) received a Ph.D. degree in electrical engineering from the University of Connecticut, Storrs, in 1993. Since 1995, he has been with the Department of Automation, Tsinghua National Laboratory for Information Science and Technology, and the Center for Intelligent and Networked Systems, Tsinghua University. He is currently with the MOE Key Laboratory for Intelligent Networks and Network Security, Faculty of Electronic and Information Engineering, Xi'an Jiaotong University, Xi'an, China, where he is also the Dean of the Faculty of Electronic and Information Engineering. He is an Academician of the Chinese Academy of Sciences.