Optimal Dynamic Subset Sampling: Theory and Applications

Lu Yi
yilu@ruc.edu.cn
Renmin University of China
Beijing, China

Hanzhi Wang
hanzhi_wang@ruc.edu.cn
Renmin University of China
Beijing, China

Zhewei Wei*
zhewei@ruc.edu.cn
Renmin University of China
Beijing, China

ABSTRACT

We study the fundamental problem of sampling independent events, called *subset sampling*. Specifically, consider a set of n distinct events $S = \{x_1, \ldots, x_n\}$, in which each event x_i has an associated probability $p(x_i)$. The subset sampling problem aims to sample a subset $T \subseteq S$, such that every x_i is independently included in S with probability p_i. A naive solution is to flip a coin for each event, which takes $O(n)$ time. However, the specific goal is to develop data structures that allow drawing a subset sample in time proportional to the expected output size $\mu = \sum_{i=1}^{n} p(x_i)$, which can be significantly smaller than n in many applications. The subset sampling problem serves as an important building block in many tasks and has been the subject of various research for more than a decade.

However, most of the existing subset sampling approaches are conducted in a *static* setting, where the events or their associated probability in set S is not allowed to be changed over time. These algorithms incur either large query time or update time in a *dynamic setting* despite the ubiquitous time-evolving events with changing probability in real life. Therefore, it is a pressing need, but still, an open problem, to design efficient dynamic subset sampling algorithms.

In this paper, we propose ODSS, the first optimal dynamic subset sampling algorithm. The expected query time and update time of ODSS are both optimal, matching the lower bounds of the subset sampling problem. We present a nontrivial theoretical analysis to demonstrate the superiority of ODSS. We also conduct comprehensive experiments to empirically evaluate the performance of ODSS. Moreover, we apply ODSS to a concrete application: influence maximization. We empirically show that our ODSS can improve the complexities of existing influence maximization algorithms on large real-world evolving social networks.

CCS CONCEPTS

• **Theory of computation → Sketching and sampling**.

KEYWORDS

subset sampling, dynamic probabilities, optimal time cost

1 INTRODUCTION

In the past decade, we have been experiencing a huge "Big Data" movement fueled by our ever-increasing ability and desire to gather, store, and share data. Driven by the exponential blowup in data volumes, efficient algorithms are now in higher demand than ever before. In particular, *sampling* is one of the most powerful techniques in algorithm design and analysis, which can effectively reduce the problem size and is often a necessary tool for achieving high scalability. On the other hand, randomly chosen samples also serve as robust estimators for critical quantities in many cases.

In this paper, we study the fundamental problem of sampling independent events, called *Subset Sampling*. Specifically, consider a set S with n elements $S = \{x_1, \ldots, x_n\}$. All of the n elements represent n distinct events, each of which (e.g., the i-th element x_i) is associated with a probability $p(x_i) \in [0, 1]$. A query for the subset sampling problem returns a subset $T \subseteq S$, in which the i-th event x_i is independently included with probability $p(x_i)$. A trivial bound for the subset sampling problem is $O(n)$ that we flip a biased coin for every $p(x_i)$. However, such complexity can be significantly larger than the expected output size $\mu = \sum_{i=1}^{n} p(x_i)$, leading to excessive time cost. A well-adopted goal for the subset sampling problem is to draw a subset sample in time roughly proportional to the expected output size $\mu = \sum_{i=1}^{n} p(x_i)$.

The subset sampling problem has long been a crucial building block in many tasks. Various applications are in dire need of efficient subset sampling techniques, especially of the implementations in *dynamic* settings. To be more specific, in a dynamic setting, the probability that an event happens can be updated dynamically. And the elements in set S are allowed to be changed over time. Such dynamic settings are extremely common in real life, as events are usually time-evolving in practice. Nonetheless, designing dynamic subset sampling algorithms is a more challenging task due to their hardness. In the following, we will present three concrete examples to demonstrate the wide applications of the subset sampling problem and the dynamic settings of these applications.

1.1 Concrete Applications

Dynamic Influence Maximization. The influence maximization (IM) problem aims to find a set of k users in social networks which

Table 1: Comparison of subset sampling algorithms.

Algorithm	Expected Generation Time	Update Time
The Naive Method	$O(n)$	$O(1)$
HybridSS [25]	$O\left(1 + n\sqrt{\min\{\bar{p}, 1 - \bar{p}\}}\right)$	$O(n)$
BringmannSS [2]	$O(1 + \mu)$	$O(\log^2 n)$
ODSS (Ours)	$O(1 + \mu)$	$O(1)$

can infect the largest number of users in the network. At the heart of existing influence maximization algorithms is generating random reverse reachable (RR) sets efficiently. The state-of-the-art IM algorithm, SUBSIM [11], reduces the complexity of IM by leveraging a subset sampling approach [2] to accelerate the generation of RR sets. However, the subset sampling approach can only attain the optimality in a static setting, presenting a significant drawback due to its demanding $O(\log^2 n)$ update time when handling update operations. This complexity poses a substantial obstacle to the practical application of SUBSIM in real-world scenarios, where social influence is inherently dynamic. Take, for instance, the rapid surge in the influence of celebrities resulting from scandals or rumors within a short span of time. Such fluctuations can swiftly render the user influence rankings obsolete. Consequently, there exists an urgent imperative to devise an efficient subset sampling algorithm specifically tailored for dynamic settings.

Approximate Graph Propagation. In recent Graph Neural Network (GNN) research, an emerging trend is to employ node proximity queries to build scalable GNN models, including prominent examples such as SGC, APPNP, PPRGo, and GBP. These proximity-based GNNs, in contrast to the original GCN, decouple prediction and propagation and thus enable the mini-batch training, leading to significant enhancements in model scalability. To model various proximity measures, Wang [26] proposes the unified graph propagation formulas and introduces a UNIFIED randomized algorithm to calculate the formulas efficiently. Notably, the algorithm frequently employs subset sampling techniques to enhance computation efficiency.

Computational Epidemiology. Particle-based simulation models have become a prevalent choice in the field of computational epidemiology [9]. In these models, each infector I possesses the ability to independently infect susceptible individuals who have come into contact with I during a specific time period T. Consequently, the infection process associated with each infector inherently presents itself as a subset sampling problem.

1.2 Motivations and Contributions

Despite the importance and wide-spread applications of the dynamic subset sampling problem, it remains an open problem to design the optimal dynamic subset sampling algorithm. To be more specific, it has been established that any subset sampling algorithm cannot run faster than $\Omega(1 + \mu)$, where $\mu = \sum_{i=1}^{n} p(x_i)$ denotes the expected output size of the problem. And any algorithm needs an update time of $\Omega(1)$ to handle an update operation (e.g., inserting/deleting an element or modifying a probability). This implies that in the scenario of subset sampling, $\Omega(1 + \mu)$ and $\Omega(1)$ are the lower bounds for query and update time complexity, respectively. Thus, an optimal dynamic subset sampling algorithm is required

to achieve an $O(1 + \mu)$ expected query time per sample with $O(1)$ update time per update operation. In other words, we look for the dynamic subset sampling algorithm that is able to derive the sampling result in time roughly proportional to the output size and only requires a constant number of operations to support real-time updates.

Motivations. The subset sampling problem has been the subject of extensive research for more than a decade [2, 11, 25]. However, to the best of our knowledge, none of them can achieve optimality in a dynamic setting. More specifically, we summarize the complexities of existing subset sampling algorithms in Table 1. Notably, the Naive method supports real-time updates, but its drawback lies in the excessive query time, making it challenging to handle large-scale datasets effectively. Conversely, the BringmannSS method [2] builds dedicated index structures to reduce query time. Unfortunately, such index structures struggle to accommodate dynamic changes, leading to significant update time. As a consequence, the subset sampling problem remains open in a dynamic setting despite years of effort yet.

Our Contributions. Motivated by the need to design efficient dynamic subset sampling algorithms, we make the following contributions.

- We propose ODSS, the first optimal dynamic subset sampling algorithm. We theoretically prove that our ODSS only costs $O(1 + \mu)$ expected query time and supports $O(1)$ update time per update operation. The two complexities are *both* optimal, matching the lower bounds of subset sampling.
  We conduct comprehensive experiments to empirically evaluate the performance of our ODSS. The experimental results show that our ODSS consistently outperforms all existing subset sampling algorithms on all datasets.
- To further demonstrate the effectiveness of our ODSS, we apply our ODSS to a concrete application: Influence Maximization. We empirically show that our ODSS can improve the complexities of existing influence maximization algorithms on large real-world evolving social networks.

2 PRELIMINARY

Considering the subset sampling problem, we note that the optimality can be trivially achieved when all probabilities in set S are identical, i.e., $p(x_1) = p(x_2) = \ldots = p$. This is because, in the special case, the index j_1 of the first sampled element follows the geometric distribution: $\Pr[j_1 = i] = (1 - p)^{i-1} \cdot p$. Given the memoryless property of geometric distribution, we can iterate the process of generating $i \sim \text{Geo}(p)$ where $\text{Geo}(p)$ is the geometric distribution with parameter p, and return x_{j_1+i} as the second element sampled in set S. We repeat the above process until the index of the sampled element exceeds n. It is known that the random number $i \sim (1 - p)^{i-1} \cdot p$ can be generated in $O(1)$ time by setting $i = \left\lfloor \frac{\log \text{rand}()}{\log(1-p)} \right\rfloor$, where $\text{rand}()$ denotes a uniform random number in $[0, 1]$. Thus, in this special case, the expected query time is bounded by $O(np) = O(1 + \mu)$ with $O(1)$ update time per update operation [6, 16]. For reference, we call such method the GeoSS method.

In the general case, a trivial bound of query time complexity is $O(n)$ since we can flip a biased coin for every $p(x_i)$. We call such method the Naive method, which offers an $O(n)$ query time with an $O(1)$ update time per update operation. To further reduce the time cost, a series of studies have been devoted to this problem over the past decade. In the following, we will review the state-of-the-art algorithms and briefly analyze why these methods fall short of achieving optimality in a dynamic setting.

The HybridSS Method. Tsai et al. [25] proposes a subset sampling method called HybridSS, which achieves $O(1 + n\sqrt{\min\{\bar{p}, 1 - \bar{p}\}})$ expected query time with $O(n)$ update time in the worst case. Here $\bar{p} = \frac{1}{n} \sum_{i=1}^{n} p(x_i)$ denotes the mean of all probabilities. The idea of the HybridSS method is to divide the original set S into two disjoint sets X and Y, where $X = \{x_i \leq \sqrt{\bar{p}} \mid x_i \in S\}$ and $Y = S - X$. The HybridSS method invokes the Naive method to sample elements in set Y. For the elements in set X, the HybridSS method first treats all probabilities in X as $\sqrt{\bar{p}}$ and applies the GeoSS method in the special case to sample some candidates. Since $p(x_i) \leq \sqrt{\bar{p}}$ for every $x_i \in X$, the HybridSS method then accepts each candidate (e.g., x_i) with probability $\frac{p(x_i)}{\sqrt{\bar{p}}}$. By this strategy, every element x_i in set X is still guaranteed to be included in the final subset T with probability $\sqrt{\bar{p}} \cdot \frac{p(x_i)}{\sqrt{\bar{p}}} = p(x_i)$.

The BringmannSS Method. The BringmannSS method [2] is proposed by Bringmann and Panagiotou, which is the first algorithm achieving the optimal $O(1 + \mu)$ expected query time complexity in a *static* setting. The core of the BringmannSS method is a bucket sort operation accompanied by the Alias method for weighted sampling. Here weighted sampling is another crucial sampling schema that aims to sample an element from a probability distribution. The Alias method is the state-of-the-art weighted sampling approach (in a static setting, however), which achieves the optimal $O(1)$ time per sample by building a dedicated alias table. The BringmannSS method revealed an interesting interplay between the *subset sampling* problem and the *weighted sampling* problem.

We describe the BringmannSS method in detail here since our ODSS is partially inspired by the BringmannSS method. Specifically, the BringmannSS method first partitions all elements in set S into $(\lceil \log n \rceil + 1)$ buckets. The k-th bucket ($k = 1, 2, \ldots, \lceil \log n \rceil$) consists of the element x of which the associated probability $p(x) \in (2^{-k}, 2^{-k+1}]$. And the last bucket (i.e., the ($\lceil \log n \rceil + 1$)-th bucket) consists of all the elements x with $p(x) < \frac{1}{n}$. For each element x in the k-th bucket, the BringmannSS method sets $\bar{p}(x) = 2^{-k+1}$, which is actually an upper bound on $p(x)$. As a result, after partitioning elements into buckets, all elements $x \in S$ have been sorted in descending order of $\bar{p}(x)$. The BringmannSS method further considers the elements ranked in $[2^k, 2^{k+1})$ as a group, denoted as B_k. Then the BringmannSS method invokes the SOTA weighted sampling method, the Alias method, to find the groups in which at least one element is sampled. Specifically, starting at the first group B_1, the BringmannSS method invokes the Alias method to sample the group index j from the probability distribution $\Pr[X = j] = q_j \cdot \Pi_{k=1}^{j-1}(1 - q_k)$. Here q_k denotes the probability that at least one potential element (sampled with probability $\bar{p}(x)$) is sampled in B_k (i.e., $q_k = 1 - (1 - \bar{p}(x))^{|B_k|}$). The BringmannSS method iterates the above process starting at B_{j+1} to sample the next group until $j > \lceil \log n \rceil + 1$. In each sampled group, the BringmannSS method adopts the idea given in the HybridSS method. That is, we can first sample a candidate x with probability $\bar{p}(x)$, then accept x with probability $\frac{p(x)}{\bar{p}(x)}$.

The BringmannSS method achieves $O(1 + \mu)$ expected query time but incurs an $O(\log^2 n)$ update time per update operation. Despite the limitations, the BringmannSS method provides two key insights for reducing complexity: (i) We can partition the elements in set S into groups and sample them at the group level first. By such a strategy, we can effectively reduce the problem size. (ii) There is a rich interplay between different sampling schemas. Some techniques adopted in other sampling problems may also contribute to the algorithm design for the subset sampling problem.

2.1 Other Related Work

In this subsection, we briefly introduce a useful trick named *table lookup*, which is widely adopted in algorithms for weighted sampling [12, 13, 21]. At the heart of the table lookup trick is a table structure. Each row index refers to a probability distribution, and cells in a row stores the sampling outcomes corresponding to the probability distribution. Consider a weighted sampling problem defined on a finite domain with bounded size of possible probability distributions. We can first use the given distribution to index into the table, and uniformly sample a cell in the corresponding row to derive the sampling outcome. Since all possible outcomes under all possible probability distributions have been included in the table, we can thus achieve the optimal $O(1)$ query time with $O(1)$ update time.

3 ALGORITHM

We first propose *Basic Dynamic Subset Sampling* in Section 3.1. In Section 3.2, we improve this basic algorithm by the table lookup method and propose the optimal algorithm *Optimal Dynamic Subset Sampling (ODSS)*, which achieves the optimal query time and the optimal update time in the meanwhile.

3.1 Basic ODSS Algorithm

In the basic algorithm, the elements are divided into $O(\log n)$ groups, ensuring the probabilities of the elements within each group differ by a factor of at most 2. Rather than querying within each group, we first perform queries at the group level and subsequently sample elements within the sampled groups. Note that the group-level querying itself constitutes a subset sampling problem with only $O(\log n)$ elements. By repeating this process iteratively, we continue to reduce the number of elements until it reaches a constant value, thereby obtaining the basic algorithm.

Group Partition. We divide all the elements into $(\lceil \log n \rceil + 1)$ groups. Define the k-th group $G_k = \{x_i | 2^{-k} < p(x_i) \leq 2^{-k+1}\}$ for $k \in \{1, \ldots, K - 1\}$, and $G_K = \{x_i | p(x_i) \leq 2^{-K+1}\}$, $K = \lceil \log n \rceil + 1$. Note that 2^{-k+1} is the upper bound on $\{p(x_i) | x_i \in G_k\}$. Within the group G_k, we first sample elements as *candidates* with probability 2^{-k+1} and then use rejection for each candidate to yield a correctly distributed sample. Hence, the probability that there exists at least one candidate in G_k can be calculated as $p(G_k) =$

$1 - (1 - 2^{-k+1})^{|G_k|}$. We first sample each group G_k with probability $p(G_k)$ and then sample elements within each sampled group. Take the rightmost part of Figure 1 as an example. We divide seven elements into K groups, where $K = \lceil \log n \rceil + 1 = 4$. To distinguish with other symbols, we denote element x_i as $x_i^{(0)}$ and group G_i as $G_i^{(0)}$. $G_k^{(0)}$ contains elements with probabilities in $(2^{-k}, 2^{-k+1}]$ for $0 \le k < K$. The last group $G_4^{(0)}$ contains elements with probabilities less than 2^{-3}. After the division, $G_1^{(0)}, G_2^{(0)}, G_3^{(0)}, G_4^{(0)}$ contain $\{x_1^{(0)}, x_4^{(0)}, x_6^{(0)}\}, \{x_7^{(0)}\}, \{x_2^{(0)}\}, \{x_3^{(0)}, x_5^{(0)}\}$, respectively.

Sampling within a Group. We first consider the problem of sampling elements within a group G_k given that G_k is sampled. Let Y_k be an indicator random variable for the event that G_k is sampled, that is, $\Pr[Y_k = 1] = p(G_k)$. Note that the probability of each element must be converted to a conditional probability due to $Y_k = 1$. We present Algorithm 1 to illustrate the details of sampling element within a group G_k conditioned on $Y_k = 1$. Since we sample the elements as candidates with probability 2^{-k+1} first, the index X_1 of the first candidate in G_k is geometrically distributed. That is, $\Pr[X_1 = j] = 2^{-k+1}(1 - 2^{-k+1})^{j-1}$. Then the conditional probability of the first candidate can be calculated as

$$\Pr[X_1 = j | Y_k = 1] = \frac{\Pr[X_1 = j \cap Y_k = 1]}{\Pr[Y_k = 1]}$$

for $j \in \{1, \dots, |G_k|\}$. Note that $\Pr[X_1 = j \cap Y_k = 1]$ is the same as $\Pr[X_1 = j]$ since $X_1 = j$ implies that G_k contains at least one candidate. Thus, $\Pr[X_1 = j | Y_k = 1] = 2^{-k+1}(1 - 2^{-k+1})^{j-1}/p(G_k)$. We generate a random number r, distributed as X_1 conditioned on $Y_k = 1$, as the index of the first candidate of G_k (Algorithm 1 Line 3). Subsequently, we proceed to sample the second candidate from the remaining group members. Note that the sampling of the second candidate is contingent upon two conditions: (1) $Y_k = 1$ and (2) the r-th element is selected as the first candidate. Let X_2 be the index of the second candidate. Then we have $\Pr[X_2 = j + r | Y_k = 1 \cap X_1 = r] = \Pr[X_2 = j + r | X_1 = r] = 2^{-k+1}(1 - 2^{-k+1})^{j-1}$ since $X_1 = r$ implies $Y_k = 1$. Note that $X_2 = j + r$ is the index in the whole group, while the index of the second candidate in the remainder of the group is j. Therefore, the index of the second candidate in the remainder of the group is distributed geometrically. We generate a new r, which is a geometric random variable, and select the r-th element in the remainder of the group as the second candidate (Algorithm 1 Line 8). Iterate the above process in G_k to select the further candidates until the index exceeds $n_k = |G_k|$. After sampling candidates, we accept each candidate x_i with probability $p(x_i)/2^{-k+1}$. By this strategy, x_i is still guaranteed to be sampled with probability $2^{-k+1} \cdot \frac{p(x_i)}{2^{-k+1}} = p(x_i)$. The theoretical property of Algorithm 1 will be shown in Lemma 1 in Section 4. Take the rightmost part of Figure 1 as an example again. Given that $G_1^{(0)}$ is sampled, we query in $G_1^{(0)}$ and obtain a sample $x_1^{(0)}, x_6^{(0)}$. Since $G_2^{(0)}, G_3^{(0)}, G_4^{(0)}$ fail to be sampled, we do not query in these groups and thus no elements in these groups can be sampled.

The Algorithm Structure. Recall that querying at the group level is also a subset sampling problem with $O(\log n)$ elements. Each group G_k is associated with probability $p(G_k)$. Partitioning the groups again can break down this subset sampling problem into

---

Algorithm 1: SampleWithinGroup

Input: a group G_k
Output: a drawn sample T

1   $n_k \leftarrow |G_k|, T \leftarrow \emptyset, h \leftarrow 0$;
2   Let $G_k[i]$ be the i-th element of G_k;
3   Generate a random r s.t. $\Pr[r = j] = \frac{2^{-k+1}(1 - 2^{-k+1})^{j-1}}{p(G_k)}$, $j \in \{1, \dots, n_k\}$;
4   **while** $r + h \le n_k$ **do**
5      $h \leftarrow r + h$;
6   **if** rand() $< p(G_k[h])/2^{-k+1}$ **then**
7         $T \leftarrow T \cup \{G_k[h]\}$;
8   Generate a random $r \sim \text{Geo}(2^{-k+1})$;
9   **return** T

---

a new subset sampling problem but with $O(\log \log n)$ elements. Continuing in this manner until the remaining subset sampling problem is of constant size and hence trivial to solve, we could derive the *Basic Dynamic Subset Sampling* algorithm as demonstrated in Algorithm 2. To distinguish the subset sampling problems at various levels, we denote the set of elements at level ℓ as $S^{(\ell)}$ and the set of groups at level ℓ as $G^{(\ell)}$. Let $S^{(0)} = S = \{x_1, \dots, x_n\}$. We first partition $S^{(0)}$ into a set of groups $G^{(0)} = \{G_1^{(0)}, \dots, G_K^{(0)}\}$, $K = \lceil \log n \rceil + 1$. The subset sampling problem for $G^{(0)}$ is defined at level 1. Let $S^{(1)} = \{x_1^{(1)}, \dots, x_K^{(1)}\}$, $x_i^{(1)}$ with probability $p(G_i^{(0)})$. Thus, a sampled $x_i^{(1)}$ indicates that $G_i^{(0)}$ is sampled. Continuing in this fashion, we maintain $\{S^\ell\}$ and $\{G^\ell\}$, $\ell \in \{0, \dots, L\}$, until the size of $S^{(L)}$ is a constant, at which $L = \log^* n$. For querying, we first sample each element $x_k^{(L)} \in S^{(L)}$ with probability $p(x_k^{(L)})$ using the Naive method and enqueue the sampled elements into $Q^{(L)}$ (Algorithm 2 Line 2). Note that each element at level L corresponds to a group at level $L - 1$, and the sampled element $x_k^{(L)}$ indicates that the group $G_k^{(L-1)}$ is sampled. Thus, for each sampled $x_k^{(L)}$ in $Q^{(L)}$, we sample the elements within $G_k^{(L-1)}$ by Algorithm 1 and thus draw a sample of $S^{(L-1)}$. Iterating the above process from level $L - 1$ down to level 1, we finally obtain $Q^{(0)}$, which is exactly a sample of $S^{(0)}$.

The theoretical property of Algorithm 2 will be shown in Theorem 1 in Section 4.

3.2 Optimal Dynamic Subset Sampling

Although the basic algorithm described above demonstrates competitive query time, it falls short of meeting the lower bound. To ensure that the number of elements at level L is a constant, we repeat the reduction $\log^* n$ times. It is remarkable that the number of groups becomes so small after a constant number of reduction steps. We present the table lookup method for solving the subset sampling problem with so few elements. Subsequently, we replace the Naive method with this table lookup method in the basic algorithm and reduce the reduction steps to 2 times, thereby giving rise to the *Optimal Dynamic Subset Sampling* algorithm (ODSS).
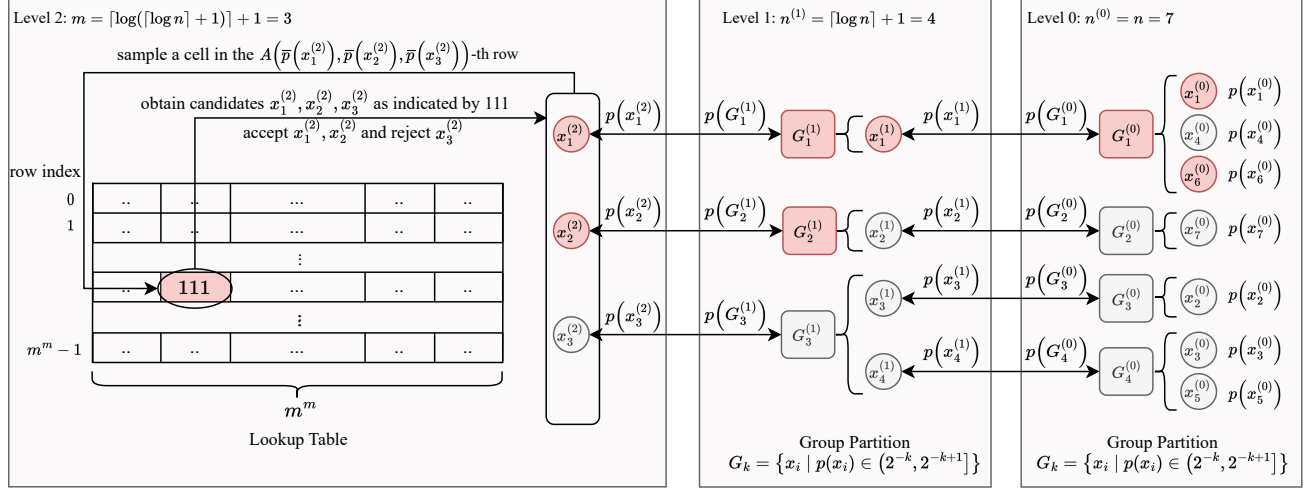
Figure 1: An example of Optimal Dynamic Subset Sampling.

Algorithm 2: BasicDynamicSubsetSampling

Input: the maximum level L, the set of elements $S^{(\ell)}$ and
 the set of groups $G^{(\ell)}$ at level l for $0 \leq \ell \leq L$

Output: a drawn sample T

1 $Q^{(\ell)} \leftarrow \emptyset, 1 \leq \ell \leq L$;

2 Sample each element $x_k^{(L)} \in S^{(L)}$ with the Naive method
 and enqueue $x_k^{(L)}$ into $Q^{(L)}$ if $x_k^{(L)}$ is sampled;

3 **for** $\ell \leftarrow L$ *to 1* **do**

4   **while** $Q^{(\ell)} \neq \emptyset$ **do**

5         $x_k^{(\ell)} \leftarrow$ deQueue($Q^{(\ell)}$);

6   Enqueue each element in the return of
         SampleWithinGroup($G_k^{(\ell-1)}$) into $Q^{(\ell-1)}$;

7 $T \leftarrow Q^{(0)}$;

8 **return** T

Table Lookup. We denote the total number of elements as m to differentiate it from the original problem size, denoted as n. Consider a set S with m elements, $S = \{x_1, \ldots, x_m\}$. The subset sampling problem returns a subset of S as a drawn sample, that is, each subset of S is chosen as the drawn sample with a certain probability. We encode each subset of S as a bit array B with m bits. $B[i]$ is the i-th (from right to left) bit indicating if x_i is included in the subset. For instance, when $m = 3$, $B = 011$ represents a subset $\{x_2, x_1\}$. Since the elements are independently included in B, the probability of B being the drawn sample is given by $p(B) = \prod_{x_i \in B} p(x_i) \cdot \prod_{x_j \notin B}(1 - p(x_j))$. Notably, the sum of $p(B)$ over all distinct subsets $B \subseteq S$ is equal to 1. An important observation is that independently sampling each element x_i with $p(x_i)$ and then returning the sampled elements is equivalent to returning a subset B with probability $p(B)$. Note that the latter is a weighted sampling problem that is entirely distinct from subset sampling. The elements of this weighted sampling problem are subsets of S and only one subset is returned as a sample. Based on this observation, we apply the table lookup trick to solve the subset

sampling problem for a special case, where $p(x_i) \in \{\frac{1}{m}, \frac{2}{m}, \ldots, \frac{m}{m}\}$. Note that $p(B)$ is a multiple of $\frac{1}{m^m}$ since $p(x_i)$ is a multiple of $\frac{1}{m}$, that is, $m^m p(B) \in \{1, \ldots, m^m\}$ for each subset B of S. We create a *lookup row* with m^m entries and fill $m^m p(B)$ entries with B for each subset B. The m^m entries in the row are all filled with a subset since $p(B)$ for all subsets B sum up to 1 as mentioned above. When querying, we select an entry of the row uniformly and return the subset in the entry. There are $m^m p(B)$ entries filled with the subset B, so the probability of returning B as a drawn sample is $m^m p(B)/m^m = p(B)$. For a general case without any limitation on $p(x_i)$, let $\bar{p}(x_i) = \lceil m p(x_i) \rceil / m$ for each element x_i. We maintain the lookup row with respect to $\bar{p}(x_i)$ instead of $p(x_i)$. Thus, each entry in the row contains x_i with probability $\bar{p}(x_i)$. Denote the elements in the subset as candidates. We accept each candidate with probability $p(x_i)/\bar{p}(x_i)$ to ensure that x_i is sampled with $p(x_i)$.

Table Lookup for All Possible Distributions. If the probability of an element x_i is altered, so too is $\bar{p}(x_i)$. To accommodate the modification, we also construct such a lookup row for each possible distribution of $(\bar{p}(x_1), \ldots, \bar{p}(x_m))$. Note that $\bar{p}(x_i) \in \{\frac{1}{m}, \ldots, \frac{m}{m}\}$, so there are exactly m^m different distributions of $(\bar{p}(x_1), \ldots, \bar{p}(x_m))$. To help with indexing into the table, let

$$A(\bar{p}(x_1), \ldots, \bar{p}(x_m)) = \sum_{i=1}^{m} (m\bar{p}(x_i) - 1)m^{i-1}.$$

Note that $A(\bar{p}(x_1), \ldots, \bar{p}(x_m)) \in \{0, \ldots, m^m - 1\}$. For each distribution of $(\bar{p}(x_1), \ldots, \bar{p}(x_m))$, we fill in the $A(\bar{p}(x_1), \ldots, \bar{p}(x_m))$-th row with the sampling outcomes. When querying, we generate a random r uniformly from $\{0, \ldots, m^m - 1\}$, and return the subset in the entry in the $A(\bar{p}(x_1), \ldots, \bar{p}(x_m))$-th row and the r-th column.

The Optimal Algorithm Structure. Replacing the Naive method in the basic algorithm with the table lookup trick, we finally derive the optimal algorithm, Optimal Dynamic Subset Sampling (ODSS). We provide the pseudocode in Algorithm 3. The subset sampling problems at level 0 and level 1 are solved by querying within the sampled groups, while a table is used for querying at level 2. When querying, we first sample $S^{(2)}$ using the table lookup method, which plays a role as the Naive method in the basic algorithm. The remains of ODSS are the same as Lines 3 to 6 in Algorithm 2, but with $L = 2$.

Algorithm 3: Optimal Dynamic Subset Sampling

Input: the set of elements $S^{(0)}, S^{(1)}, S^{(2)}$ and the set of groups $G^{(0)}, G^{(1)}$, the table for $S^{(2)}$

Output: a drawn sample T

1 $L \leftarrow 2, T \leftarrow \emptyset$;

2 $Q^{(\ell)} \leftarrow \emptyset, 1 \leq \ell \leq L$;

3 Draw a sample of $S^{(2)}$ by the table lookup method and enqueue the sampled elements into $Q^{(2)}$;

4 The steps are the same as Algorithm 2 Line 3 to 6 with $L = 2$;

5 $T \leftarrow Q^{(0)}$;

6 **return** T

Figure 1 provides an example of Optimal Dynamic Subset Sampling. There are seven elements $S = \{x_1, \ldots, x_7\}$. Let $S^{(0)} = S$. After the group partitions and maintaining the lookup table, we obtain the structure with three levels. At level 2, we have $m = 3$ and $S^{(2)} = \{x_1^{(2)}, x_2^{(2)}, x_3^{(2)}\}$. When querying, we first index into the $A(\bar{p}(x_1^{(2)}), \bar{p}(x_2^{(2)}), \bar{p}(x_3^{(2)}))$-th row. By selecting an entry uniformly from m^m entries in the row, we obtain a subset $B = 111$, which indicates $x_3^{(2)}, x_2^{(2)}, x_1^{(2)}$ are candidates. Then we accept each candidate $x_i^{(2)}$ with probability $p(x_i^{(2)})/\bar{p}(x_i^{(2)})$. It comes out that $x_1^{(2)}, x_2^{(2)}$ are accepted, which implies that $G_1^{(1)}, G_2^{(1)}$ are sampled. Thus, we query within $G_1^{(1)}$ and $G_2^{(1)}$ using Algorithm 1 and only $x_1^{(1)}$ in $G_1^{(1)}$ is sampled. Then, by querying within $G_1^{(0)}$, we accept $x_1^{(0)}, x_6^{(0)}$ and reject $x_4^{(0)}$. Therefore, we draw a sample of S, $\{x_1, x_6\}$.

3.3 Update Operations

In this subsection, we demonstrate how ODSS handles element insertion, element deletion, and probability modification. Let's consider the example of inserting an element x with probability $p(x)$ into the set S. We start by adding x to the corresponding group $G_k^{(0)}$ based on $p(x)$. Subsequently, we recalculate the probability $p(G_k^{(0)})$ for this group. The modification of $p(G_k^{(0)})$ may necessitate transferring $x_k^{(1)}$ from one group at level 1 to another, resulting in a change in the probabilities of two groups at level 1. In other words, the probabilities of two elements in $S^{(2)}$ are altered. Denote the two elements at level 2 as x_i and x_j. After modifying their probabilities, denoted as $p'(x_i)$ and $p'(x_j)$ respectively, we recalculate the row index. This involves adding $(m\bar{p}'(x_i) - 1)m^{i-1} - (m\bar{p}(x_i) - 1)m^{i-1} + (m\bar{p}'(x_j) - 1)m^{j-1} - (m\bar{p}(x_j) - 1)m^{j-1}$ to the previous value of $A(\bar{p}(x_0), \ldots, \bar{p}(x_m))$. This step completes the insertion of x_i. This step completes the insertion of x_i.

To achieve the update operations above, we maintain an array for each group so that accessing the element at a specified location and assigning an element to a specified location can be done in constant time. We also maintain an array for storing the group index k and the position j for each element x_i, indicating that x_i is the j-th element in G_k. Let $\text{Gidx}(x_i)$ be the group index of x_i and $\text{Gpos}(x_i)$ be the position of x_i in the group.

When inserting an element into a group, we append it to the end of the array. When it comes to deletion, we remove x_i from the $\text{Gidx}(x_i)$-th group by replacing the value at position $\text{Gpos}(x_i)$ with the value at the last position in the group and then delete the last cell. That is, we move the last element to the position of x_i, and then the array contracts by one cell. Thus, inserting or deleting an element within a group can be done in constant time.

With the above techniques, each element insertion can be solved in constant time. Note that each element deletion also involves an adjustment of one group at level 0. It is convenient to solve element deletions with a similar solution. A probability modification can be achieved by an element insertion following an element deletion. Note that we are left with the problem that the number of groups in each level can be altered with the increase or decrease of $n^{(\ell)}$. We provide a technical trick to maintain the correct number of groups in constant time. The details of the trick are deferred to the proof of the update time cost in Appendix A.6.

Take Figure 1 as an example again. We update the set of elements by inserting x_8 to $S^{(0)}$ with $p(x_8) = 0.3$. Then, x_8 is assigned to $G_2^{(0)}$ since $2^{-2} \leq p(x_8) \leq 2^{-1}$. The size of $G_2^{(0)}$ is enlarged to two elements, so $p(G_2^{(0)})$ is revised to $1 - (1 - 2^{-1})^2 = 3/4$. Thus, $x_1^{(1)}$ has to be moved from $G_2^{(1)}$ to $G_1^{(1)}$. As a result, we have to revise $p(G_1^{(1)}), p(G_2^{(1)})$. Subsequently, at level 2, we recalculate $\bar{p}(x_1)$ and $\bar{p}(x_2)$, and revise $A(\bar{p}(x_1), \bar{p}(x_2), \bar{p}(x_3))$ accordingly. Then we are done with the insertion.

4 THEORETICAL ANALYSIS

In this section, we analyze the theoretical properties of our algorithms. Theorem 1 and Theorem 2 illustrate the correctness and the expected cost of Basic Dynamic Subset Sampling and Optimal Dynamic Subset Sampling, respectively.

Our theoretical analysis is conducted under the standard word RAM model proposed by Fredman et al. [8]. The word RAM model is a commonly adopted computational model which offers a simple yet accurate abstraction of real-world computers. In particular, we will assume that any basic arithmetical operations on a single word of $\log n$ bits take constant time, where n matches the problem size. The basic arithmetical operations include addition, multiplication, comparison, logical shifts, $\exp(x)$, and $\log(x)$ (the binary logarithm of x). The operation rand() (to generate a random number uniformly from $[0, 1]$) takes constant time, too.

THEOREM 1 (BASIC DYNAMIC SUBSET SAMPLING). *Using Algorithm 2, the* SUBSETSAMPLING *problem can be solved in $O(2^{\log^* n} \cdot \mu + 2^{\log^* n} + \log^* n)$ expected query time, $O(n \log^* n)$ preprocessing time, and $O(n \log^* n)$ space.*

THEOREM 2. *Using Algorithm 3, the* SUBSETSAMPLING *problem can be solved in $O(1 + \mu)$ query time, constant update time, $O(n)$ preprocessing time, and $O(n)$ space.*

To prove Theorem 1 and Theorem 2, we need several lemmas. In particular, we first prove that Algorithm 1 returns an unbiased sample within G_k in $O(\mu_k + 1)$ expected query time, where $\mu_k = \sum_{x_i \in G_k} p(x_i)$.

LEMMA 1. *Given that a group $G_k = \{x_i | 2^{-k} < p(x_i) \leq 2^{-k+1}\}$ for $1 \leq k < K$ is successfully sampled with probability $p(G_k) =$*

$1 - (1 - 2^{-k+1})^{n_k}$, *Algorithm 1 returns a sample T of G_k in $O(\mu_k + 1)$ expected query time, where $K = \lceil \log n \rceil + 1$, $\mu_k = \sum_{x_i \in G_k} p(x_i)$, $n_k = |G_k| \leq n$. Each element $x_i \in G_k$ is included in T independently with probability $p(x_i)$. A similar conclusion holds for $G_K = \{x_i | p(x_i) \leq 2^{-K+1}\}$.*

Next, we bound $\mu^{(\ell)}$, the sum of the probabilities for all elements at level ℓ.

LEMMA 2. *Let $\mu = \sum_{i=1}^{n} p(x_i)$. Denote the sum of the probabilities for all elements in level ℓ as $\mu^{(\ell)}$. We have*

$$\mu^{(\ell)} \leq 2^\ell \mu + 2^\ell - 1 \tag{1}$$

The following lemma analyzes the theoretical property of the table lookup method.

LEMMA 3. *With the table lookup method, the subset sampling problem with $S = \{x_1, \ldots, x_m\}$ can be solved with $O(1 + \mu)$ query time, $O(2^m \cdot m + m^m)$ preprocessing time, and $O(m^{2m})$ space.*

With the help of Lemma 1, Lemma 2 and Lemma 3, we are able to prove Theorem 1 and Theorem 2. For the sake of readability, we defer all proofs to the appendix.

5 EXPERIMENTS

In this section, we experimentally evaluate the performance of our ODSS against alternatives.

Experiment Environment. We conduct all experiments on a machine with an Intel(R) Xeon(R) Silver 4114 CPU @ 2.20GHz CPU and 1007GB memory in Linux OS. All the methods are implemented in C++ compiled by g++ with O3 turned on.

Algorithms. We compared our ODSS against the Naive method, the HybridSS method, and the BringmannSS method. For each method, we report the total time of 100 queries as the query time. And we report the time cost of inserting and deleting elements 1000 times as the update time.

The Distributions Of Probabilities. To examine the effectiveness of our ODSS, we carefully set the probabilities of elements distributed with different *skewnesses*. In particular, we choose the normal distribution, the half-normal distribution, the exponential distribution, and the log-normal distribution. The normal distribution is a symmetric distribution with skewness as 0. We set the mean of the normal distribution as 0 and the variance as 10. For the half-normal distribution, the *probability density (PDF)* is $f(x) = \frac{\sqrt{2}}{\sigma \sqrt{\pi}} \exp{-\frac{x^2}{2\sigma^2}}$ and the skewness of it is just below 1. We set the σ^2 as 10 for it. For the exponential distribution, the PDF is $f(x) = \lambda e^{-\lambda x}$ with the skewness as 2, and we set $\lambda = 1$. For the log-normal distribution, the PDF is $f(x) = \frac{1}{x\sigma \sqrt{2\pi}} \exp{-\frac{(\ln x - \mu)^2}{2\sigma^2}}$. By setting $\mu = 0$ and $\sigma = \sqrt{\ln 2}$, the skewness of this log-normal distribution is 4. To generate the probabilities of elements, we first sample a value for each element with the distribution settings. To ensure each value is greater than 0, we subtract *min* from each value, where *min* is the minimum of all the generated values. Then we scale the range of the values to $[0, 1]$ and scale the sum of them to a specified μ according to the needs of various experiments.

Query Time v.s. Update time. In light of the pressing needs for simultaneously achieving sample and update efficiency in the subset sampling problem, we draw the trade-off plots between *query time* and *update time* of each method in Figure 2. We set the number of elements $n = 100,000$, and the sum of the probabilities is $\mu = 1$. From Figure 2, we observe that ODSS consistently achieves the best of both query time and update time on all datasets. The Naive method performs efficient updates, while it requires a large query time, which is 1000× slower than ODSS. The query time of the BringmannSS method is significantly smaller than the Naive method, however, the update time is 1000× larger than our ODSS, respectively. The HybridSS method performs much better than the Naive method on querying, however, the update time of the HybridSS method is $10^5×$ slower than that of the Naive method. These results concur with our analysis for query time and update time. For update time, the Naive method and ODSS are the best with only $O(1)$ time per element insertion/deletion, while the BringmannSS method needs $O(\log n)$ time and the HybridSS methods need $O(n)$ time. For query time, the BringmannSS method, and our ODSS are the best with $O(1 + \mu)$ optimal query time. The HybridSS method needs $O\left(1 + n\sqrt{\min\{\bar{p}, 1 - \bar{p}\}}\right)$ query time, and the mean of the probabilities $\bar{p}$ is so small here, and thus the HybridSS performs better than the Naive method, which needs $O(n)$ time in any case.

Effectiveness Of Query. To further examine the query time of these algorithms, we vary μ, the sum of the probabilities of elements, and show the query time in Figure 3. We set $n = 1,000,000$, and vary μ in $\{1, 10, 10^2, 10^3, 10^4, 10^5, 10^6\}$. From Figure 3, we observe that with the increase of μ, the query time of all algorithms except the Naive method becomes larger. This concurs with the theoretical analysis that the query time of the BringmannSS method and ODSS is $O(1 + \mu)$, and the increase of the mean $\bar{p}$ of the probabilities incurs the increase of the query time of the HybridSS method. Additionally, we note that when $\mu = n = 10^6$, the query time of the BringmannSS method is larger than that of the Naive method due to the complicated index structure. In comparison, the HybridSS method performs well when $\mu = n$ since it will degenerate into the Naive method. ODSS costs the least query time with any μ in this experiment.

Effectiveness Of Update. To further examine the update time of these algorithms, we set $\mu = 1$ and vary n in $\{10^4, 5 \times 10^4, 10^5, 5 \times 10^5, 10^6, 5 \times 10^6, 10^7\}$. Figure 4 shows the update time of the algorithms on the four distributions. We observe that the Naive method and ODSS consistently outperform other algorithms on all the distributions with different skewnesses. The update time of the BringmannSS method is 50× ~ 100× larger than that of the Naive method and our ODSS. The update time has little change with the increase of n since the $O(\log n)$ term increases little when n varies from 10^4 to 10^7. The update of the HybridSS method becomes slower with the increase of n, which concurs with its update time complexity $O(n)$.

6 EMPIRICAL STUDY ON INFLUENCE MAXIMIZATION

This section discusses how our algorithm ODSS improves the influence maximization (IM) algorithms for evolving graphs. We will
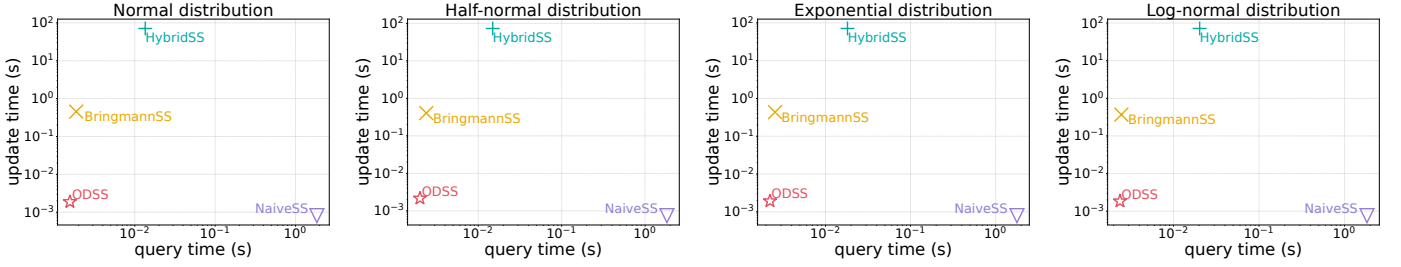
Figure 2: query time v.s. update time overhead on distributions with different skewnesses. ($n = 10^5, \mu = 1$)
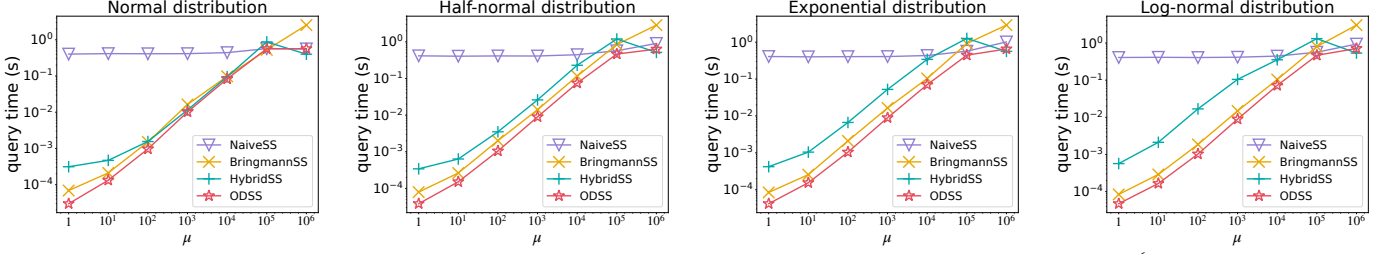

Figure 3: Varying μ: query time (s) on distributions with different skewnesses. ($n = 10^6$)
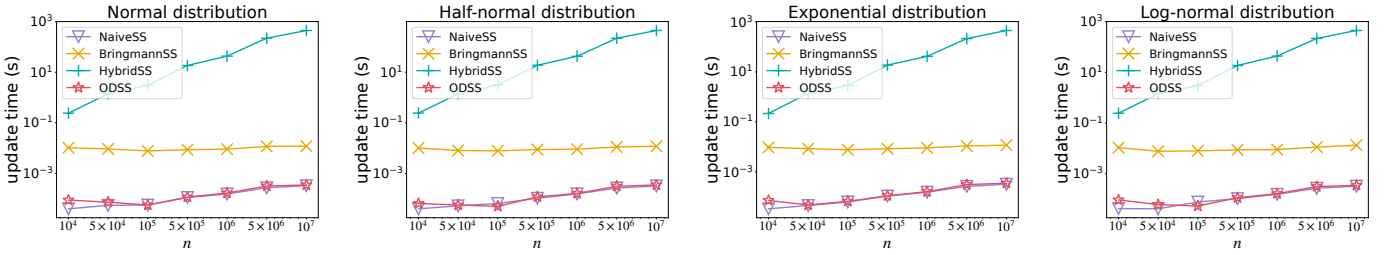

Figure 4: Varying n: update time (s) on distributions with different skewnesses. ($\mu = 1$)

first briefly introduce the IM problem and the dynamic IM problem as follows.

With the booming of online social networks, information diffusion in networks has attracted extensive research. As a key problem in information diffusion research, IM finds important applications in viral marketing [7], network monitoring [17], social recommendation [28]. IM aims to find a set of seed users in the social networks with the maximum *influence spread*. Specifically, consider a social graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, where $\mathbb{V}$ is the set of nodes in $\mathbb{G}$ and $\mathbb{E}$ is the set of edges in $\mathbb{G}$. Each edge $e = (u, v) \in \mathbb{E}$ is associated with a propagation probability $p(u, v) \in [0, 1]$. A specified *diffusion model* M is defined to model the stochastic process of spreading influence on $\mathbb{G}$. The *influence spread* of a set of nodes S, denoted as $\sigma_{\mathbb{G},M}(S)$, is the expected number of users influenced by S under the diffusion model M.

DEFINITION 1 (INFLUENCE MAXIMIZATION [15]). *Given a graph $\mathbb{G}$, a diffusion model M, and a positive integer k, the influence maximization problem selects a set S_k of k nodes from $\mathbb{G}$ as the seed set to maximize the influence spread $\sigma_{\mathbb{G},M}(S_k)$, i.e., $S_k = \arg\max_{S:|S| \le k} \sigma_{\mathbb{G},M}(S)$.*

The IC model. In this work, we focus on a widely adopted model, the *Independent Cascade (IC)* model. Under the IC model, a seed set $S \in \mathbb{V}$ spreads its influence as follows. At time step 0, all nodes in S are activated. Each active node u in step t activates each of its outgoing neighbors v that is inactive in step $t - 1$ with probability

$p(u, v)$. Note that u has only a chance to activate its outgoing neighbors. After the time step t, u stays active and do not activate any nodes again. The spreading process terminates when no more active nodes can activate other nodes. We observe that the process that an active node u activates its outgoing neighbors is exactly a subset sampling problem. Each outgoing neighbor v of u is independently sampled with probability $p(u, v)$.

Research for static IM. It has been proved that IM is NP-hard under the Independent Cascade (IC) model [15]. Computing the influence spread function $\sigma(S)$ of a seed set S is #P-hard under the IC model [3]. Due to the theoretical hardness of IM, extensive research has tried to design efficient IM algorithms for a few decades. According to [19], the existing algorithms can be classified into three categories: the sketch based [1, 23, 24] solutions, the simulation based solutions [10, 15, 17, 27, 29], and the proxy based solutions [4, 5, 14, 20]. The sketch based solution and the simulation based solution both solve a huge number of subset sampling problems.

Research for dynamic IM. Considering the highly dynamic nature of social influence, the *dynamic IM* problem has received much attention in the past decade. Dynamic IM hopes to support real-time influential users tracking on evolving social networks. According to Peng [22], the evolution model of dynamic IM can be classified into two categories, the *incremental model* and the *fully dynamic model*.
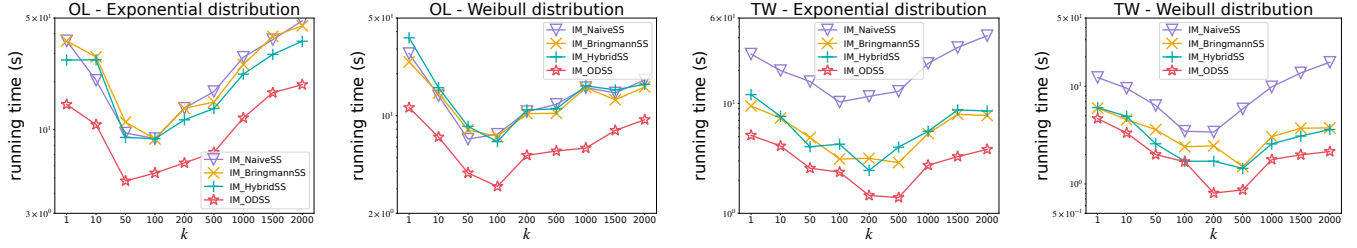
Figure 5: running time of dynamic IM algorithms based on various subset sampling structures.
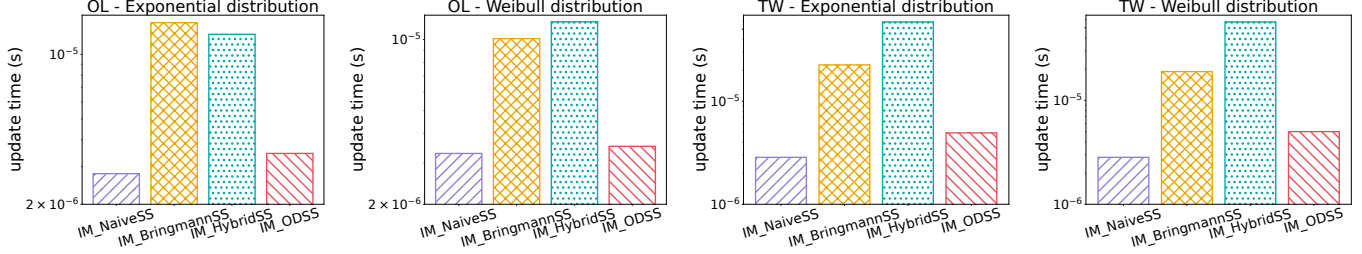

Figure 6: update time of dynamic IM algorithms based on various subset sampling structures.

Table 2: Datasets

Dataset	n	m	m/n
Orkut-Links	3,072,441	117,185,083	38
Twitter	41,652,230	1,468,365,182	35

In the incremental model, the social network only enlarges over time, while in the fully dynamic model, users can join in or leave the networks, and the propagation probabilities between users can be altered over time. Unfortunately, Peng has proven that no algorithm can achieve any meaningful approximation guarantee in the fully dynamic model, that is, re-running an IM algorithm upon every update can achieve the lower bound of the running time. However, the existing static IM algorithms focus on static social networks and do not support evolving networks. This motivates us that by replacing the subset sampling module in the static IM algorithms with the structure of ODSS, we can obtain a new dynamic IM algorithm for the fully dynamic model. The query time of each subset sampling problem in the new IM algorithm is reduced to $O(1 + \mu)$. In the meanwhile, since our ODSS performs $O(1)$ update time, the new IM algorithms can support the insertion/deletion of user nodes or the modification of propagation probabilities in $O(1)$ time.

In the following, we will apply our ODSS and alternatives to static RR-sketch based solutions to obtain new dynamic IM algorithms for the fully dynamic model in Section 6.1. The experiments show that the new dynamic IM algorithm based on our ODSS can reduce the running time of IM and support real-time updates simultaneously. We also examine how our ODSS improves the complexities of simulation based solutions. Due to the simplicity of the simulation based solutions, we defer the detail in Appendix A.1.

6.1 Applying ODSS to RR-Sketch Solutions

The Reverse Reachable Sketch (RR-Sketch) solutions are the current mainstream of the sketch based solutions for IM [1, 19, 24]. The solutions are based on the concept of *random Reverse Reachable (RR)*

set. Generating a random RR set starts at a uniformly selected node v from $\mathbb{V}$, and then reversely samples the nodes that can activate v, until no more nodes can be sampled. The set of the sampled nodes is a random RR set, denoted as R. To reverse sample the nodes that can activate v, we sample each ingoing neighbor u of v with respect to $p(u, v)$ and then repeat the above process by starting at the sampled ingoing neighbors. It is also a subset sampling problem, but with the ingoing neighbors of a node as elements. After generating a sufficient number of random RR sets, a greedy algorithm is applied to select a seed set[15]. The error of the RR-Sketch solutions is bounded due to the submodularity of the influence spread function $\sigma(\cdot)$ [15].

We notice that Guo et al. [11] propose a framework called SUB-SIM (Subset Sampling with influence maximization) based on the OPIM-C algorithm [23] to accelerate the generation of random RR sets by modifying the subset sampling module. However, SUBSIM focuses on static IM problems and can not support the evolving graphs. We can obtain various dynamic IM algorithms for the fully dynamic model by replacing the subset sampling module with various dynamic subset sampling structures (our ODSS and alternatives). We conduct experiments for these IM algorithms to evaluate the IM running time and the update time for the edge insertions/deletions.

The experiments are conducted on two real-world graphs, Orkut-Links (OL) and Twitter (TW). The two graphs are publicly available at [18]. We give the summary of the two graphs in Table 2. Following previous studies [11, 23], we test the case when the possibilities of edges follow two skewed distributions: exponential distribution and Weibull distribution. For exponential distribution, We set $\lambda = 1$. For each node, we scale the sum of the probabilities of its outgoing neighbors to 1. For Weibull distribution, the PDF is $f(x) = \frac{a}{b} \cdot \left(\frac{x}{b}\right)^{(a-1)} \cdot e^{-(x/b)^a}$. The parameters a and b are drawn

uniformly from $[0, 10]$ for each edge. For each node, the sum of the probabilities of its outgoing neighbors is scaled to 1.

To fully explore the efficiency of the algorithms, we vary the size k of the seed set in $\{1, 10, 50, 100, 200, 500, 1000, 2000\}$ in the experiments. We repeat each IM algorithm to generate the seed set 5 times and report the average running time. To simulate the evolution of social networks, we uniformly choose 10^6 edges from each graph and report the average time for the insertions and deletions of these 10^6 edges. Figure 5 presents the running time of the dynamic IM algorithms based on various subset sampling structures. The IM_ODSS, the IM algorithm based on our ODSS, outperforms alternatives in all tested graphs and all seed sizes. In particular, the running time of IM_ODSS is 10× smaller than the IM_NaiveSS on the TW graph when $k \geq 500$. Figure 6 presents the update time of the algorithms. IM_ODSS performs per update in less than 4×10^{-6} s. IM_BringmanSS and IM_HybridSS both suffer the long update time. In particular, IM_HybridSS is 10× slower than IM_ODSS on the TW graph.

7 CONCLUSION

Subset sampling is a fundamental problem in both data mining and theoretical computer science. Its dynamic version finds various applications in Influence Maximization, Graph Neural Networks, and Computational Epidemiology. This paper proposes ODSS, the first optimal dynamic subset sampling algorithm. We present a theoretical analysis to demonstrate the optimal complexities of ODSS. We also conduct extensive experiments to evaluate the performance of ODSS and give an empirical study on the Influence Maximization problem. For future work, it is interesting to see the applications of ODSS in Graph Neural Networks (GNN) and Computational Epidemiology.

ACKNOWLEDGEMENTS

REFERENCES

[1] Christian Borgs, Michael Brautbar, Jennifer Chayes, and Brendan Lucier. 2014. Maximizing social influence in nearly optimal time. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*. SIAM, 946–957.

[2] Karl Bringmann and Konstantinos Panagiotou. 2012. Efficient sampling methods for discrete distributions. In *International colloquium on automata, languages, and programming*. Springer, 133–144.

[3] Wei Chen, Chi Wang, and Yajun Wang. 2010. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1029–1038.

[4] Wei Chen, Yajun Wang, and Siyu Yang. 2009. Efficient influence maximization in social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. 199–208.

[5] Suqi Cheng, Huawei Shen, Junming Huang, Wei Chen, and Xueqi Cheng. 2014. Imrank: influence maximization via finding self-consistent ranking. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*. 475–484.

[6] Luc Devroye. 2006. Nonuniform random variate generation. *Handbooks in operations research and management science* 13 (2006), 83–121.

[7] Pedro Domingos and Matt Richardson. 2001. Mining the network value of customers. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. 57–66.

[8] Michael L Fredman and Dan E Willard. 1993. Surpassing the information theoretic bound with fusion trees. *Journal of computer and system sciences* 47, 3 (1993), 424–436.

[9] Timothy C Germann, Kai Kadau, Ira M Longini Jr, and Catherine A Macken. 2006. Mitigation strategies for pandemic influenza in the United States. *Proceedings of the National Academy of Sciences* 103, 15 (2006), 5935–5940.

[10] Amit Goyal, Wei Lu, and Laks VS Lakshmanan. 2011. Celf++ optimizing the greedy algorithm for influence maximization in social networks. In *Proceedings of the 20th international conference companion on World wide web*. 47–48.

[11] Qintian Guo, Sibo Wang, Zhewei Wei, and Ming Chen. 2020. Influence maximization revisited: Efficient reverse reachable set generation with bound tightened. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 2167–2181.

[12] Torben Hagerup, Kurt Mehlhorn, and J Ian Munro. 1993. Maintaining discrete probability distributions optimally. In *International Colloquium on Automata, Languages, and Programming*. Springer, 253–264.

[13] Torben Hagerup, Kurt Mehlhorn, and James Ian Munro. 1993. Optimal algorithms for generating discrete random variables with changing distributions. *Lecture Notes in Computer Science* 700 (1993), 253–264.

[14] Kyomin Jung, Wooram Heo, and Wei Chen. 2012. Irie: Scalable and robust influence maximization in social networks. In *2012 IEEE 12th international conference on data mining*. IEEE, 918–923.

[15] David Kempe, Jon Kleinberg, and Éva Tardos. 2003. Maximizing the spread of influence through a social network. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. 137–146.

[16] Donald Knuth. 1981. Seminumerical algorithms. *The art of computer programming* 2 (1981).

[17] Jure Leskovec, Andreas Krause, Carlos Guestrin, Christos Faloutsos, Jeanne Van-Briesen, and Natalie Glance. 2007. Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. 420–429.

[18] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

[19] Yuchen Li, Ju Fan, Yanhao Wang, and Kian-Lee Tan. 2018. Influence maximization on social graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering* 30, 10 (2018), 1852–1872.

[20] Qi Liu, Biao Xiang, Enhong Chen, Hui Xiong, Fangshuang Tang, and Jeffrey Xu Yu. 2014. Influence maximization over large-scale social networks: A bounded linear approach. In *Proceedings of the 23rd ACM international conference on conference on information and knowledge management*. 171–180.

[21] Yossi Matias, Jeffrey Scott Vitter, and Wen-Chun Ni. 2003. Dynamic generation of discrete random variates. *Theory of Computing Systems* 36 (2003), 329–358.

[22] Binghui Peng. 2021. Dynamic influence maximization. *Advances in Neural Information Processing Systems* 34 (2021), 10718–10731.

[23] Jing Tang, Xueyan Tang, Xiaokui Xiao, and Junsong Yuan. 2018. Online processing algorithms for influence maximization. In *Proceedings of the 2018 International Conference on Management of Data*. 991–1005.

[24] Youze Tang, Xiaokui Xiao, and Yanchen Shi. 2014. Influence maximization: Near-optimal time complexity meets practical efficiency. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. 75–86.

[25] Meng-Tsung Tsai, Da-Wei Wang, Churn-Jung Liau, and Tsan-sheng Hsu. 2010. Heterogeneous subset sampling. In *Computing and Combinatorics: 16th Annual International Conference, COCOON 2010, Nha Trang, Vietnam, July 19-21, 2010. Proceedings 16*. Springer, 500–509.

[26] Hanzhi Wang, Mingguo He, Zhewei Wei, Sibo Wang, Ye Yuan, Xiaoyong Du, and Ji-Rong Wen. 2021. Approximate graph propagation. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 1686–1696.

[27] Yu Wang, Gao Cong, Guojie Song, and Kunqing Xie. 2010. Community-based greedy algorithm for mining top-k influential nodes in mobile social networks. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*. 1039–1048.

[28] Mao Ye, Xingjie Liu, and Wang-Chien Lee. 2012. Exploring social influence for recommendation: a generative model approach. In *Proceedings of the 35th*

international ACM SIGIR conference on Research and development in information retrieval. 671–680.

[29] Chuan Zhou, Peng Zhang, Wenyu Zang, and Li Guo. 2015. On the upper bounds of spread for greedy algorithms in social network influence maximization. *IEEE Transactions on Knowledge and Data Engineering* 27, 10 (2015), 2770–2783.

A   APPENDIX

A.1   Applying ODSS to simulation based solutions

The simulation based solutions apply the *Monte Carlo* method for estimating the influence function $\sigma(\cdot)$. An MC simulation under IC model to estimate $\sigma(S)$ starts from the set S, traverses on a *possible world* of $\mathbb{G}$, and returns the number of reachable nodes as an estimation of $\sigma(S)$. To generate a possible world of $\mathbb{G}$, we remove each edge $e = (u, v)$ with probability $1 - p(u, v)$. It is a subset sampling problem with $S = \mathbb{E}$, in which each element $e = (u, v)$ is associated by a reserving probability $p(u, v)$. When it comes to the dynamic IM problems, edges are inserted into the graph or removed from the graph over time, resulting in the element insertion/deletion of the subset sampling problem.

We conduct experiments on the two real-world graphs (OL and TW) with exponential distribution and Weibull distribution and show the query time for possible world generations and the update time.

For each graph with various skewed distributions, we query for 100 possible worlds and conduct 1000 edge insertions and 1000 edge deletions of the graph. We report the tradeoff between the query time and the update time for all algorithms in Figure 7. Our ODSS outperforms alternatives in both query time and update time on all the tested datasets. The update time of the Naive method approaches our ODSS while the query time is $3\times \sim 5\times$ larger than ODSS. Note that the ratio of the query time of the Naive method to that of ODSS is associated with m/n since the query time of ODSS in this setting is $O(n)$. Recall that we scale the sum of the probabilities of the outgoing neighbors of each node to 1, thus, the sum of the probabilities μ here is n. Therefore, the query time of ODSS is $O(1 + \mu) = O(n)$, while the query time of the Naive method is $O(m)$. As shown in Table 2, the m/n term is 38, 35 for OL and TW, respectively. Therefore, the empirical result indicates that the constant of the μ term in the $O(1 + \mu)$ query time of ODSS is a bit larger than the constant of the n term in the $O(n)$ query time of the Naive method. It concurs with the empirical results in Figure 3. The BringmanSS method performs $2\times$ faster than the Naive method for query, however, it needs $100\times$ update time larger than the Naive method in the OL graph.

A.2   Proof of Lemma 1

PROOF. We first consider the correctness of the algorithm. Define a set of random variables $X_1, \ldots,$ where X_i is the index of the i-th candidate. Since G_k is sampled, we have $Y_k = 1$. Note that $X_i = j$ implies that there is at least one candidate in G_k, so $\Pr[X_i = j] = \Pr[X_i = j | Y_k = 1] \cdot \Pr[Y_k = 1]$. Let R_i be the i-th random number generated in Algorithm 1, that is, $\Pr[R_1 = j] = 2^{-k}(1 - 2^{-k})^{j-1}/p(G_k)$ and $\Pr[R_i = j] = 2^{-k}(1 - 2^{-k})^{j-1}$, $1 < i \le n_k$. For simplicity's sake, we denote the j-th element in G_k as e_j. Note that the algorithm runs on the condition that G_k is sampled. $\Pr[X_i = j | Y_k = 1] = \Pr[\sum_{h=1}^{i} R_h = j]$. Thus, the j-th element e_j is sampled as the i-th candidate with probability $\Pr[X_i = j] = \Pr[\sum_{h=1}^{i} R_h = j] \cdot \Pr[Y_k = 1]$, $1 \le i \le j$. Let $Z_i = \sum_{h=1}^{i} R_h$. Note that e_j can be sampled as a i-th candidate, where $1 \le i \le j$. Then, the probability that e_j is sampled as a

candidate is

$$\sum_{i=1}^{j} \Pr[X_i = j] = \Pr[Y_k = 1] \sum_{i=1}^{j} \Pr[Z_i = j]$$

Let

$$\hat{p}(e_j) = \sum_{i=1}^{j} \Pr[Z_i = j] \qquad (2)$$

We will prove that $\hat{p}(e_j) = 2^{-k}/p(G_k)$ for all j using Mathematical Induction. For the initial step, we have $\hat{p}(e_1) = \Pr[Z_1 = 1] = 2^{-k}/p(G_k)$ for the first element e_1. For the inductive step, we prove that $\hat{p}(e_j) = 2^{-k}/p(G_k)$ if the proposition is true for any $1 \le j' < j$. The situation for $Z_1 = j$ is simple,

$$\Pr[Z_1 = j] = \frac{2^{-k}(1 - 2^{-k})^{j-1}}{p(G_k)} \qquad (3)$$

Consider Z_{i-1} when $i > 2$ and $Z_i = j$. Firstly, $Z_{i-1} \le j - 1$ if $Z_i = j$. And the index of the $(i-1)$-th candidate Z_{i-1} is at least $i - 1$. Hence, $\Pr[Z_i = j]$ can be written as $\sum_{h=i-1}^{j-1} \Pr[Z_{i-1} = h \cap R_i = j - h]$ for $i > 2$. Note that the random variables $R_1, \ldots, R_{n_k}$ are mutually independent. Thus, we have

$$\Pr[Z_i = j] = \sum_{h=i-1}^{j-1} \Pr[Z_{i-1} = h] \Pr[R_i = j - h] \qquad (4)$$

Applying Equation 4 to Equation 2, we have

$$\hat{p}(e_j) = \Pr[Z_1 = j] + \sum_{i=2}^{j} \sum_{h=i-1}^{j-1} \Pr[Z_{i-1} = h] \cdot 2^{-k}(1 - 2^{-k})^{j-h-1}$$

We interchange the order of the summations and obtain

$$\hat{p}(e_j) = \Pr[Z_1 = j] + \sum_{h=1}^{j-1} \sum_{i=1}^{h} \Pr[Z_i = h] \cdot 2^{-k}(1 - 2^{-k})^{j-h-1}$$

Note that $\sum_{i=1}^{h} \Pr[Z_i = h]$ is exactly $\hat{p}(e_h)$. Using the inductive hypothesis, we have $\hat{p}(e_h) = 2^{-k}/p(G_k)$ for $h < j$. Thus,

$$\hat{p}(e_j) = \Pr[Z_1 = j] + \sum_{h=1}^{j-1} \frac{2^{-k}}{p(G_k)} \cdot 2^{-k}(1 - 2^{-k})^{j-h-1}$$

Applying Equation 3, we obtain that $\hat{p}(e_j) = 2^{-k}/p(G_k)$. This completes the inductive step. Hence, e_j is sampled with probability

$$\sum_{i=1}^{j} \Pr[X_i = j] = p(G_k) \cdot 2^{-k}/p(G_k) = 2^{-k}$$

We accept each candidate e_j with probability $p(e_j)/2^{-k}$. Thus, the element e_j is sampled with probability $2^{-k} \cdot p(e_j)/2^{-k} = p(e_j)$. In a word, given that G_k is sampled with probability $p(G_k)$, Algorithm 1 return an unbiasedness sample of G_k.

Next we consider the expected query time of Algorithm 1. The expected query time only depends on the number of times we generate random numbers, which equals the number of candidates plus one (one time for the index out of the group). For $0 \le k < K$, the expected number of candidates in G_k is $\sum_{i=1}^{n_k} 2^{-k} \le \sum_{x_i \in G_k} 2p(x_i) = 2\mu_k$ since $p(x_i) > 2^{-k-1}$. For G_K, the expected number of candidates is $2^{-K} \cdot n_K \le 2^{-K} \cdot n \le 1$. Therefore, the expected query time for G_k is $2\mu_k + 1 = O(\mu_k + 1)$ for $0 \le k < K$, and $O(1)$ for $k = K$.
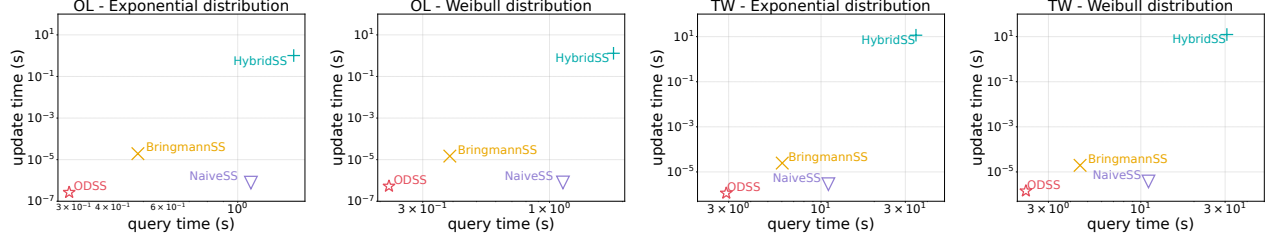
Figure 7: query time v.s. update time for generating possible worlds

For simplicity, we use $O(\mu_k + 1)$ as the expected time for $0 \le k \le K$ in the later part of the paper. □

A.3 Proof of Lemma 2

PROOF. $\mu^{(0)} = \mu = \sum_{i=1}^{n} p(x_i)$. Let $n^{(\ell)} = \left| S^{(\ell)} \right|$ be the number of elements at level ℓ, $n_k^{(\ell)} = \left| G_k^{(\ell)} \right|$ be the number of elements contained in $G_k^{(\ell)}$, $K^{(\ell)} = \lceil \log n^{(\ell)} \rceil$ be the largest index of the groups at level ℓ. Note that $p(x_k^{(\ell)}) = p(G_k^{(\ell-1)})$, so $\mu^{(\ell)} = \sum_{k=1}^{K^{(\ell-1)}} p(G_k^{(\ell-1)})$. And $p(G_k^{(\ell-1)}) = 1 - (1 - 2^{-k})^{n_k^{(\ell-1)}}$ is the probability that $G_k^{(\ell-1)}$ contains at least one candidate. Thus, we have

$$\mu^{(\ell)} = \sum_{k=1}^{K^{(\ell-1)}} 1 - (1 - 2^{-k})^{n_k^{(\ell-1)}} \le \sum_{k=1}^{K^{(\ell-1)}} 2^{-k} n_k^{(\ell-1)}$$

since $(1 - x)^y \ge 1 - xy$ for any $0 \le x < 1$ and $y \ge 1$. Note that $p(x_i^{(\ell)}) > 2^{-k-1}$ if $x_i^{(\ell)} \in G_k^{(\ell)}$ for $0 \le k < K^{(\ell)}$ at any level ℓ. Thus, we have

$$2^{-k} n_k^{(\ell-1)} \le \sum_{x_i \in G_k^{(\ell-1)}} 2p(x_i^{(\ell-1)})$$

for $0 \le k < K^{(\ell-1)}$. For $k = K^{(\ell-1)}$, $2^{-k} = 2^{-\lceil \log n^{(\ell-1)} \rceil} \le 1/n^{(\ell-1)}$. Note that $n_k^{(\ell-1)} \le n$, so $2^{-k} n_k^{(\ell-1)} \le 1$ for $k = K^{(\ell-1)}$. Therefore, we have

$$\mu^{(\ell)} \le \sum_{x_i^{(\ell-1)} \in S^{(\ell-1)}} 2p(x_i^{(\ell-1)}) + 1 = 2\mu^{(\ell-1)} + 1$$

Thus,

$$\mu^{(\ell)} \le 2\mu^{(\ell-1)} + 1 \le \ldots \le 2^\ell \mu + 2^\ell - 1$$

□

A.4 Proof of Theorem 1

PROOF. **Unbiasedness.** We first show that Algorithm 2 draw an unbiased sample of S. Firstly, the Naive method gives an unbiased sample of $S^{(L)}$. Each sampled element $x_k^{(L)}$ indicates that $G_k^{(L-1)} \in G^{(L-1)}$ is sampled with $p(G_k^{(L-1)})$. According to Lemma 1, given that a group G_k^ℓ is sampled with $p(G_k^\ell)$, Algorithm 1 returns a sample of G_k^ℓ. Thus, we obtain an unbiased sample of $S^{(L-1)}$ by combined the outcomes of *SamplingWithinGroup*$(G_k^{(L-1)})$ for all sampled groups $G_k^{(L-1)}$. Repeating in this fashion, we finally obtain an unbiased sample of $S^{(0)}$. Thus, Algorithm 2 returns an unbiased sample of the subset sampling problem.

Expected Query Time. Note that L equals $\log^* n$ to guarantee that the number of items at level L is a constant. Sampling elements at level L (Algorithm 2 Line 2) costs constant time since $n^{(L)}$ is a constant. Then we examine the cost of sampling elements at level ℓ, denoting as $C^{(\ell)}$, given that some groups of level ℓ is sampled (Algorithm 2 Line 3 to line 6). Note that the expected number of sampled groups at level ℓ is exactly the expected number of elements sampled at level $\ell + 1$, which equals $\mu^{(\ell+1)}$. According to Lemma 1, sampling with $G_k^{(\ell)}$ costs $2\mu_k + 1$ expected time for $0 \le k \le K^{(\ell)}$, where μ_k is the sum of probabilities for the elements in $G_k^{(\ell)}$. Let T be the set of the indexes of the sampled groups. We have

$$\mathbb{E}[C^{(\ell)}] \le O(1) + \mu^{(\ell+1)} + \mathbb{E}\left[\sum_{k \in T} (2\mu_k + 1) \right]$$

in which the $O(1)$ term for miscellaneous overhead, the $\mu^{(\ell+1)}$ term for iterating in the $\mu^{(\ell+1)}$ sampled groups. According to the linearity of expectation, we have $\mathbb{E}[\sum_{k \in T} (2\mu_k + 1)] = \mathbb{E}[\sum_{k \in T} 2\mu_k] + \mathbb{E}[\sum_{k \in T} 1]$. The first term $\mathbb{E}[\sum_{k \in T} 2\mu_k]$ is less than the sum of $2\mu_k$ for all $0 \le k \le K^{(\ell)}$, which equals $2\mu^{(\ell)}$. The second term $\mathbb{E}[\sum_{k \in T} 1] = \mu^{(\ell+1)}$. Thus, $\mathbb{E}[C^{(\ell)}] \le 2\mu^{(\ell+1)} + 2\mu^{(\ell)} + O(1)$. The total query cost

$$\mathbb{E}[C] = \mathbb{E}\left[\sum_{\ell=0}^{L} C^{(\ell)} \right] \le O(1) + \sum_{\ell=0}^{L-1} (2\mu^{(\ell+1)} + 2\mu^{(\ell)} + O(1))$$

Applying Equation (1), we derive the expected query time

$$E[C] = O(2^{\log^* n} \mu + 2^{\log^* n} + \log^* n)$$

Preprocessing Time and Memory space. In the preprocessing phase, we check the probability of each element and then assign it to an appropriate group at each level ℓ (except for L as $G^{(L)}$ is not necessary). Then, the probabilities of the groups are computed to obtain the probabilities of elements at the next level. Note that $p(G_k^{(\ell)}) = 1 - (1 - 2^{-k})^{n_k^{(\ell)}}$ can be calculated in constant time since $a^b = \exp(b \log a)$. And the number of the groups at level ℓ is $\lceil \log n^{(\ell)} \rceil + 1$, which equals $n^{(\ell+1)}$. Thus, the cost of each level ℓ is $O(n^{(\ell)} + n^{(\ell+1)}) = O(n)$. The total cost of preprocessing is $\sum_{\ell=0}^{L-1} n^{(\ell)} + n^{(\ell)} \le O(nL) = O(n \log^* n)$. Similary, the cost of memory space at level ℓ is $O(n^{(\ell)} + \lceil \log n^{(\ell)} \rceil + 1)$ for $n^{(\ell)}$ elements and $\lceil \log n^{(\ell)} \rceil + 1$) groups. The total memory space is $O(n \log^* n)$.

□

A.5 Proof of Lemma 3

PROOF. We first show that the table lookup method gives an unbiased sample of S. Denote the subset as B that we obtain by

indexing into the table with $A(\bar{p}(x_1), \ldots, \bar{p}(x_m))$ and a uniformly distributed random r. B is selected with $p(B)$ as we fill in $m^m p(B)$ entries with B in the total m^m entries of the row. For each element x_i, it is included in B with probability $\hat{p}(x_i) = \sum_{x_i \in B, B \subseteq S} p(B)$. Note that $\{B|x_i \in B, B \subseteq S\}$ are the subsets containing B and no matter if other subsets are contained or not. Thus, $\hat{p}(x_i) = \bar{p}(x_i)$. If x_i is included in B, that is, x_i is sampled as a candidate, we accept it with $p(x_i)/\bar{p}(x_i)$. Thus, x_i is sampled with $\bar{p}(x_i) \cdot p(x_i)/\bar{p}(x_i) = p(x_i)$. The unbiasedness of the table lookup method follows.

Next, we consider the expected query time of the table lookup method. When querying, we generate a random r uniformly from $\{0, \ldots, m^m - 1\}$ and obtain the bit array stored in the entry in row $A(\bar{p}(x_1), \ldots, \bar{p}(x_m))$ and column r. To decode the bit array, we first get the position idx of rightmost 1-bit of B by $idx = \log((\sim B+1)\&B)$. x_{idx} is the sampled candidate with the smallest index. Then, let $B = B - 2^{idx}$. By repeating the technique above with B, we will get all the sampled candidates. For each candidate, we then use rejection to get the probability of candidate x_i down to $p(x_i)$. Thus, the query time depends on the number of sampled candidates. Note that $\bar{p}(x_i) - p(x_i) \leq \frac{1}{m}$ since $\bar{p}(x_i) = \lceil mp(x_i) \rceil / m$. Thus we have

$$\sum_{i=1}^{m} \bar{p}(x_i) \leq \sum_{i=1}^{m} \frac{1}{m} + p(x_i) \leq \mu + 1$$

where $\mu = \sum_{i=1}^{m} p(x_i)$. Therefore, the expected query time is $O(\mu + 1)$.

At the preprocessing phase, we maintain the table and calculate $A(\bar{p}(x_1), \ldots, \bar{p}(x_m))$ for the current $(p(x_1), \ldots, p(x_m))$. For each table row, we calculate the probabilities of 2^m subsets. Note that each probability of a subset is the product of m probabilities, which costs $O(m)$ time to calculate. The time for filling in the table is $O(m^m \cdot m^m)$. Calculating $A(\bar{p}(x_1), \ldots, \bar{p}(x_m))$ costs $O(m)$ time. Thus, the total preprocessing time is $O(2^m \cdot m + m^{2m})$.

The memory space depends on the size of the table, which is $O(m^{2m})$. Note that the word length is at least $\log m^m$ bits to store the row index in a single memory word. We will show in the proof of Theorem 2 that the word length under the standard word RAM model is sufficient since the number of elements m is so few. □

A.6 Proof of Theorem 2

Proof. **Unbiasedness.** We derive the optimal algorithm by replacing the Naive method in Algorithm 2 with the table lookup method at the maximum level. The table lookup method allows for a drop in the maximum level L from $log * n$ to 2. Since we have proved the unbiasedness of the table lookup method in Lemma 3 and the unbiasedness of the basic algorithm in Theorem 1, the unbiasedness of Algorithm 3 follows.

Expected Query Time. As proved in Theorem 1, the expected cost of sampling elements within the sampled groups at level ℓ is $\mathbb{E}[C^{(\ell)}] \leq 2\mu^{(\ell+1)} + 2\mu^{(\ell)} + O(1)$. According to Lemma 3, the expected cost of table lookup at level 2 is $\mathbb{E}[C^{(2)}] = 1 + \mu^{(2)}$. Thus, the expected query time of Algorithm 3 is

$$\mathbb{E}[C] = \mathbb{E}[C^{(0)}] + \mathbb{E}[C^{(1)}] + \mathbb{E}[C^{(2)}] \leq 3\mu^{(2)} + 4\mu^{(1)} + 2\mu + 3$$

According to Lemma 2, $\mu^{(\ell)} \leq 2^\ell \mu + 2^\ell - 1$. Thus, we derive the expected query time of Algorithm 3 as

$$\mathbb{E}[C] = O(1 + \mu)$$

Technical Trick to Maintain Groups. Before we illustrate the update cost, we give a technical trick to maintain the groups correctly. The number of groups may vary over time with the increasing or decreasing number of elements n. We consider how to maintain $O(1)$ update time in this scenario. First of all, we make an assumption that the upper bound of n is known in advance. With the assumption, we maintain the lookup table with $m = \lceil \log(\lceil \log n \rceil + 1) \rceil + 1$ where n is the upper bound of the number of elements over time. Next, we consider how to maintain the correct group partitions if the number of groups $K^{(\ell)}$ varies. If $K^{(\ell)}$ is reduced to $K^{(\ell)} - 1$, we first conceptually combine the two groups into a new $G_{K-1}^{(\ell)}$, that is, we recalculate $p(G_{K-1}^{(\ell)})$ with the new group size and set a flag to indicate the combination. In each of the following query or update operations, we move two elements in the old $G_K^{(\ell)}$ to the old $G_{K-1}^{(\ell)}$ in practice. Before the number of groups is reduced by one again, all the elements can be moved into $G_{k-1}^{(\ell)}$. To deal with the increasing of K, we implicitly maintain the partition in $G_K^{(\ell)}$, i.e., the group is implicitly partitioned into two groups: $G'_K^{(\ell)}$ and $G'_{K+1}^{(\ell)}$. The implicit group $G'_{K+1}^{(\ell)}$ contains elements with probabilities no greater than $2^{-(K+1)+1}$. When K is increased to $K + 1$, the implicit $G'_K^{(\ell)}$ becomes a new group and the implicit $G'_{K+1}^{(\ell)}$ becomes the last group. In each of the following query or update operations, we partition an element in the new $G_{K+1}^{(\ell)}$ into the new implicit $G'_{K+1}^{(\ell)}$ or into the new implicit $G'_{K+1}^{(\ell)}$ according to its probability so that the partition is finished before the number of groups is increased to $K + 2$. In this way, the groups are maintained correctly in constant time.

Update Cost. As we illustrated in Section 3.3, an insertion of an element can be updated in constant time. The element deletion is the same as element insertion since it modifies the probability of one group at level 0. Note that probability modification brings a revision of the probabilities of at most two groups. Thus, the cost of probability modification is twice that of adding or removing an element but is still constant. Therefore, an update (element insertion/deletion or probability modification) can be done in constant time.

Preprocessing Time. The preprocessing time for level 1 and level 0 is $O(n)$ as proved in Theorem 2. To support efficient updates, we insertionally maintain an array for storing the position in the corresponding group for each element x_i. The total preprocessing time remains $O(n)$ for level 1 and level 0. According to Lemma 3, the preprocessing time for the table lookup method for level 2 is $O(2^m \cdot m + m^m)$ when there are m elements at level 2. Note that $m = \lceil \log(\lceil \log n \rceil + 1) \rceil + 1$, and hence $O(2^m \cdot m + m^m) = O(n)$. Thus, the preprocessing time for each level $0 \leq \ell \leq 2$ is $O(n)$, so as the total preprocessing time.

Memory Space. At level 0 and level 1, we maintain an array for each group to store the containing elements. To help with locating the position of the elements in the groups, we also maintain another

array at level 0 and level 1 to store the position of each element. Thus, the total memory space actually in use is $O(n)$. Note that the size of each array will grow and shrink with the insertions and deletions of elements. To support the changes of the arrays and bound the total memory in the meantime, we implement the arrays using the *doubling technique*. Specifically, for each array A, we maintain an insertional arrays $A+$. Denote the size of A as c, initially $c = 2$. Let n' be the number of elements in A. The size of $A+$ is initialized as $2c$. At the beginning, $A+$ is empty. When n' starts to exceed $c/2$, each time we insert a new element into A, the new element is inserted into $A+$ as well and one element contained in A is copied to $A+$. Thus, when A is full, $A+$ contains all elements in A. Then, we release the memory space of A, make $A+$ as the new A, and create a new $A+$ of size $4c$. When n' becomes smaller than $\frac{c}{4}$,

we delete $A+$ and contract A by $\frac{c}{2}$. Then we create a new $A+$ of size c. Note that the time cost for each insertion/deletion is still bounded by $O(1)$, and the memory space of $A+$ is asymptotically the same as that of A. Thus, the total memory space of level 0 and level 1 only depends on the number of elements at the level, which is $O(n)$. According to Lemma 3, it costs $O(m^{2m})$ space for level 2. Since $m = \lceil \log(\lceil \log n \rceil + 1) \rceil + 1$, we have $O(m^{2m}) = O(n)$. As mentioned in the proof of Lemma 3, the table lookup method requires the word length of at least $\log m^m$ bits. Since $m = O(\log \log n)$, the word length of $O(\log n)$ bits in the word RAM model is sufficient. Therefore, we conclude that the total memory space for the optimal algorithm is $O(n)$.

□